The GNU C Library:
Application Fundamentals

For GNU C Libraries version 2.3.x

by Sandra Loosemore
with Richard M. Stallman, Roland McGrath,
Andrew Oram, and Ulrich Drepper

This manual documents the GNU C Libraries version 2.3.x.
ISBN 1-882114-22-1, First Printing, March 2004.

Published by:

GNU Press
a division of the
Free Software Foundation
51 Franklin St, Fifth Floor
Boston, MA 02110-1301 USA

Website: www.gnupress.org
General: press@gnu.org
Orders: sales@gnu.org
Tel: 617-542-5942
Fax: 617-542-2652

Cover art by Etienne Suvasa. Cover design by Jonathan Richard. Printed in USA.

Short Contents

Table of Contents

The GNU C Library: Application Fundamentals

1 Introduction

The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation and the like. Instead, these facilities are defined in a standard *library*, which you compile and link with your programs.

The GNU C Library, described in this document, defines all of the library functions that are specified by the ISO C standard, as well as additional features specific to POSIX and other derivatives of the Unix operating system, and extensions specific to the GNU system.

The purpose of this manual is to tell you how to use the facilities of the GNU library. We have mentioned which features belong to which standards to help you identify things that are potentially nonportable. But the emphasis in this manual is not on strict portability.

1.1 Getting Started

This manual is written with the assumption that you are at least somewhat familiar with the C programming language and basic programming concepts. Specifically, familiarity with ISO standard C (see Section 1.2.1 [ISO C], page 2), rather than "traditional" pre-ISO C dialects, is assumed.

The GNU C Library includes several *header files*, each of which provides definitions and declarations for a group of related facilities; this information is used by the C compiler when processing your program. For example, the header file '`stdio.h`' declares facilities for performing input and output, and the header file '`string.h`' declares string-processing utilities. The organization of this manual generally follows the same division as the header files.

If you are reading this manual for the first time, you should read all of the introductory material and skim the remaining chapters. There are a *lot* of functions in the GNU C Library and it is not realistic to expect that you will be able to remember exactly *how* to use each and every one of them. It is more important to become generally familiar with the kinds of facilities that the library provides, so that when you are writing your programs you can recognize *when* to make use of library functions, and *where* in this manual you can find more specific information about them.

1.2 Standards and Portability

This section discusses the various standards and other sources that the GNU C Library is based upon. These sources include the ISO C and POSIX standards, and the System V and Berkeley Unix implementations.

The primary focus of this manual is to tell you how to make effective use of the GNU library facilities. But if you are concerned about making your programs compatible with these standards, or portable to operating systems other than GNU,

this can affect how you use the library. This section gives you an overview of these standards, so that you will know what they are when they are mentioned in other parts of the manual.

See Appendix A [Summary of Library Facilities], page 521, for an alphabetical list of the functions and other symbols provided by the library. This list also states which standards each function or symbol comes from.

1.2.1 ISO C

The GNU C Library is compatible with the C standard adopted by the American National Standards Institute (ANSI)as *American National Standard X3.159-1989—"ANSI C"* and later by the International Standardization Organization (ISO) as ISO/IEC *9899:1990, "Programming languages—C"*. In this manual, we refer to the standard as ISO C since this is the more general standard with respect to ratification. The header files and library facilities that make up the GNU library are a superset of those specified by the ISO C standard.

If you are concerned about strict adherence to the ISO C standard, you should use the '`-ansi`' option when you compile your programs with the GNU C Compiler. This tells the compiler to define *only* ISO standard features from the library header files, unless you explicitly ask for additional features. See Section 1.3.4 [Feature-Test Macros], page 8, for information on how to do this.

Being able to restrict the library to include only ISO C features is important because ISO C puts limitations on what names can be defined by the library implementation, and the GNU extensions don't fit these limitations. See Section 1.3.3 [Reserved Names], page 6, for more information about these restrictions.

This manual does not attempt to give you complete details on the differences between ISO C and older dialects. It gives advice on how to write programs to work portably under multiple C dialects, but does not aim for completeness.

1.2.2 POSIX (The Portable Operating System Interface)

The GNU library is also compatible with the ISO POSIX family of standards, known more formally as the *Portable Operating System Interface for Computer Environments* (ISO/IEC 9945). They were also published as ANSI/IEEE Std 1003. POSIX is derived mostly from various versions of the Unix operating system.

The library facilities specified by the POSIX standards are a superset of those required by ISO C; POSIX specifies additional features for ISO C functions, as well as specifying new additional functions. In general, the additional requirements and functionality defined by the POSIX standards are aimed at providing lower-level support for a particular kind of operating system environment, rather than general programming language support that can run in many diverse operating system environments.

The GNU C Library implements all of the functions specified in ISO/IEC *9945-1:1996, the* POSIX *System Application Program Interface*, commonly referred to as POSIX.1. The primary extensions to the ISO C facilities specified by this standard

include file-system interface primitives[1], device-specific terminal control functions[2] and process control functions.[3]

Some facilities from ISO/IEC *9945-2:1993, the* POSIX *Shell and Utilities standard* (POSIX.2) are also implemented in the GNU library. These include utilities for dealing with regular expressions and other pattern-matching facilities (see Chapter 13 [Pattern Matching], page 355).

1.2.3 Berkeley Unix

The GNU C Library defines facilities from some versions of Unix that are not formally standardized, specifically from the 4.2 BSD, 4.3 BSD and 4.4 BSD Unix systems (also known as *Berkeley Unix*) and from SunOS (a popular 4.2 BSD derivative that includes some Unix System V functionality). These systems support most of the ISO C and POSIX facilities, and 4.4 BSD and newer releases of SunOS in fact support them all.

The BSD facilities include symbolic links[4], the `select` function[5], the BSD signal functions[6] and sockets.[7]

1.2.4 SVID (The System V Interface Description)

The *System V Interface Description* (SVID) is a document describing the AT&T Unix System V operating system. It is to some extent a superset of the POSIX standard.

The GNU C Library defines most of the facilities required by the SVID that are not also required by the ISO C or POSIX standards, for compatibility with System V Unix and other Unix systems (such as SunOS) that include these facilities. However, many of the more obscure and less generally useful facilities required by the SVID are not included. (In fact, Unix System V itself does not provide them all.)

The supported facilities from System V include the methods for inter-process communication and shared memory, the `hsearch` and `drand48` families of functions, `fmtmsg` and several of the mathematical functions.

---

[1] See Sandra Loosemore et al., "File-System Interface", *GNU C Library: Systems & Network Applications* (Boston: GNU Press, 2004), available online at `http://www.gnu.org/manual/manual.html`.

[2] Ibid., "Low-Level Terminal Interface".

[3] Ibid., "Processes".

[4] Ibid., "Symbolic Links".

[5] Ibid., "Waiting for Input or Output".

[6] Ibid., "BSD Signal Handling".

[7] Ibid., "Sockets".

1.2.5 XPG (The X/Open Portability Guide)

The *X/Open Portability Guide*[8] is a more general standard than POSIX. X/Open owns the Unix copyright and the XPG specifies the requirements for systems that are intended to be Unix systems.

The GNU C Library complies with the *X/Open Portability Guide*, Issue 4.2, with all extensions common to XSI (X/Open System Interface) compliant systems and also all X/Open Unix extensions.

The additions on top of POSIX are mainly derived from functionality available in System V and BSD systems, though some of the really bad mistakes in System V systems were corrected. Since fulfilling the XPG standard with the Unix extensions is a precondition for getting the Unix brand, chances are good that the functionality is available on commercial systems.

1.3 Using the Library

This section describes some of the practical issues involved in using the GNU C Library.

1.3.1 Header Files

Libraries for use by C programs really consist of two parts: *header files* that define types and macros and declare variables and functions, and the actual library or *archive* that contains the definitions of the variables and functions.

(Recall that in C, a *declaration* merely provides information that a function or variable exists and gives its type. For a function declaration, information about the types of its arguments might be provided as well. The purpose of declarations is to allow the compiler to correctly process references to the declared variables and functions. A *definition*, on the other hand, actually allocates storage for a variable or says what a function does.)

In order to use the facilities in the GNU C Library, you should be sure that your program source files include the appropriate header files. This is so that the compiler has declarations of these facilities available and can correctly process references to them. Once your program has been compiled, the linker resolves these references to the actual definitions provided in the archive file.

Header files are included into a program source file by the '`#include`' preprocessor directive. The C language supports two forms of this directive; the first,

```
#include "header"
```

is typically used to include a header file *header* that you write yourself; this would contain definitions and declarations describing the interfaces between the different parts of your particular application. By contrast,

[8] X/Open Company, *X/Open Portability Guide,* Issue 4 (Reading, UK: X/Open Company, Ltd., 1992).

```
#include <file.h>
```

is typically used to include a header file '`file.h`' that contains definitions and declarations for a standard library. This file would normally be installed in a standard place by your system administrator. You should use this second form for the C library header files.

Typically, '`#include`' directives are placed at the top of the C source file, before any other code.[9] If you begin your source files with some comments explaining what the code in the file does (a good idea), put the '`#include`' directives immediately afterward, following the feature-test macro definition (see Section 1.3.4 [Feature-Test Macros], page 8).

The GNU C Library provides several header files, each of which contains the type and macro definitions and variable and function declarations for a group of related facilities. This means that your programs may need to include several header files, depending on exactly which facilities you are using.

Some library header files include other library header files automatically. However, as a matter of programming style, you should not rely on this; it is better to explicitly include all the header files required for the library facilities you are using. The GNU C Library header files have been written in such a way that it doesn't matter if a header file is accidentally included more than once; including a header file a second time has no effect. Likewise, if your program needs to include multiple header files, the order in which they are included doesn't matter.

Compatibility Note: Inclusion of standard header files in any order and any number of times works in any ISO C implementation. However, this has traditionally not been the case in many older C implementations.

Strictly speaking, you don't *have to* include a header file to use a function it declares; you could declare the function explicitly yourself, according to the specifications in this manual. But it is usually better to include the header file because it may define types and macros that are not otherwise available and because it may define more efficient macro replacements for some functions. It is also a sure way to have the correct declaration.

1.3.2 Macro Definitions of Functions

If we describe something as a function in this manual, it may have a macro definition as well. This normally has no effect on how your program runs—the macro definition does the same thing as the function would. In particular, macro equivalents for library functions evaluate arguments exactly once, in the same way that a function call would. The main reason for these macro definitions is that sometimes they can produce an in-line expansion that is considerably faster than an actual function call.

---

[9] For more information about the use of header files and '`#include`' directives, see Richard M. Stallman and the GCC Developer Community, "Header Files" in *The GNU C Preprocessor Manual* (2003), `http://gcc.gnu.org/onlinedocs/gcc-3.3.2/cpp/`.

Taking the address of a library function works even if it is also defined as a macro. This is because, in this context, the name of the function isn't followed by the left parenthesis that is syntactically necessary to recognize a macro call.

You might occasionally want to avoid using the macro definition of a function—perhaps to make your program easier to debug. There are two ways you can do this:

1. You can avoid a macro definition in a specific use by enclosing the name of the function in parentheses. This works because the name of the function does not appear in a syntactic context where it is recognizable as a macro call.
2. You can suppress any macro definition for a whole source file by using the '`#undef`' preprocessor directive, unless otherwise stated explicitly in the description of that facility.

For example, suppose the header file '`stdlib.h`' declares a function named `abs` with:

```
extern int abs (int);
```

and also provides a macro definition for `abs`. Then, in:

```
#include <stdlib.h>
int f (int *i) { return abs (++*i); }
```

the reference to `abs` might refer to either a macro or a function. On the other hand, in each of the following examples, the reference is to a function and not a macro:

```
#include <stdlib.h>
int g (int *i) { return (abs) (++*i); }

#undef abs
int h (int *i) { return abs (++*i); }
```

Since macro definitions that double for a function behave in exactly the same way as the actual function version, there is usually no need for any of these methods. In fact, removing macro definitions usually just makes your program slower.

1.3.3 Reserved Names

The names of all library types, macros, variables and functions that come from the ISO C standard are reserved unconditionally; your program *may not* redefine these names. All other library names are reserved if your program explicitly includes the header file that defines or declares them. There are several reasons for these restrictions:

- Other people reading your code could get very confused if, for example, you were using a function named `exit` to do something completely different from what the standard `exit` function does. Preventing this situation helps to make your programs easier to understand and contributes to modularity and maintainability.

- It avoids the possibility of a user accidentally redefining a library function that is called by other library functions. If redefinition were allowed, those other functions would not work properly.
- It allows the compiler to do whatever special optimizations it pleases on calls to these functions, without the possibility that they may have been redefined by the user. Some library facilities, such as those for dealing with variadic arguments[10] and nonlocal exits[11], actually require a considerable amount of cooperation on the part of the C compiler, and with respect to the implementation, it might be easier for the compiler to treat these as built-in parts of the language.

In addition to the names documented in this manual, reserved names include all external identifiers (global functions and variables) that begin with an underscore ('_') and all identifiers regardless of use that begin with either two underscores or an underscore followed by a capital letter. This is so that the library and header files can define functions, variables, and macros for internal purposes without risk of conflict with names in user programs.

Some additional classes of identifier names are reserved for future extensions to the C language or the POSIX.1 environment. While using these names for your own purposes right now might not cause a problem, there is the possibility of conflict with future versions of the C or POSIX standards, so you should avoid using them:

- Names beginning with a capital '`E`' followed by a digit or uppercase letter may be used for additional error-code names (see Chapter 2 [Error Reporting], page 17).
- Names that begin with either '`is`' or '`to`' followed by a lowercase letter may be used for additional character testing and conversion functions (see Chapter 4 [Character Handling], page 79).
- Names that begin with '`LC_`' followed by an uppercase letter may be used for additional macros specifying locale attributes (see Chapter 7 [Locales and Internationalization], page 181).
- Names of all existing mathematics functions (see Chapter 8 [Mathematics], page 203) suffixed with '`f`' or '`l`' are reserved for corresponding functions that operate on `float` and `long double` arguments, respectively.
- Names that begin with '`SIG`' followed by an uppercase letter are reserved for additional signal names.[12]
- Names that begin with '`SIG_`' followed by an uppercase letter are reserved for additional signal actions.[13]
- Names beginning with '`str`', '`mem`', or '`wcs`' followed by a lowercase letter are reserved for additional string and array functions (see Chapter 5 [String and Array Utilities], page 89).

---

[10] See Loosemore et al., "Variadic Functions".

[11] Ibid., "Nonlocal Exits".

[12] Ibid., "Standard Signals".

[13] Ibid., "Basic Signal Handling".

- Names that end with '_t' are reserved for additional type names.

In addition, some individual header files reserve names beyond those that they actually define. You only need to worry about these restrictions if your program includes that particular header file.

- The header file 'dirent.h' reserves names prefixed with 'd_'.
- The header file 'fcntl.h' reserves names prefixed with 'l_', 'F_', 'O_', and 'S_'.
- The header file 'grp.h' reserves names prefixed with 'gr_'.
- The header file 'limits.h' reserves names suffixed with '_MAX'.
- The header file 'pwd.h' reserves names prefixed with 'pw_'.
- The header file 'signal.h' reserves names prefixed with 'sa_' and 'SA_'.
- The header file 'sys/stat.h' reserves names prefixed with 'st_' and 'S_'.
- The header file 'sys/times.h' reserves names prefixed with 'tms_'.
- The header file 'termios.h' reserves names prefixed with 'c_', 'V', 'I', 'O', and 'TC'; and names prefixed with 'B' followed by a digit.

1.3.4 Feature-Test Macros

The exact set of features available when you compile a source file is controlled by which *feature-test macros* you define.

If you compile your programs using 'gcc -ansi', you get only the ISO C library features, unless you explicitly request additional features by defining one or more of the feature macros.[14]

You should define these macros by using '#define' preprocessor directives at the top of your source code files. These directives *must* come before any #include of a system header file. It is best to make them the very first thing in the file, preceded only by comments. You could also use the '-D' option to GCC, but it is better if you make the source files indicate their own meaning in a self-contained way.

This system exists to allow the library to conform to multiple standards. Although the different standards are often described as supersets of each other, they are usually incompatible because larger standards require functions with names that smaller ones reserve to the user program. This is not mere pedantry—it has been a problem in practice. For instance, some non-GNU programs define functions named getline that have nothing to do with this library's getline. They would not be compilable if all features were enabled indiscriminately.

[14] See Richard M. Stallman and the GCC Developer Community, "Invoking GCC" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/, for more information about GCC options.

This should not be used to verify that a program conforms to a limited standard. It is insufficient for this purpose, as it will not protect you from including header files outside the standard, or relying on semantics undefined within the standard.

_POSIX_SOURCE Macro

If you define this macro, then the functionality from the POSIX.1 standard (IEEE Standard 1003.1) is available, as well as all of the ISO C facilities.

The state of `_POSIX_SOURCE` is irrelevant if you define the macro `_POSIX_C_SOURCE` to a positive integer.

_POSIX_C_SOURCE Macro

Define this macro to a positive integer to control which POSIX functionality is made available. The greater the value of this macro, the more functionality is made available.

If you define this macro to a value greater than or equal to `1`, then the functionality from the 1990 edition of the POSIX.1 standard (IEEE Standard 1003.1-1990) is made available.

If you define this macro to a value greater than or equal to `2`, then the functionality from the 1992 edition of the POSIX.2 standard (IEEE Standard 1003.2-1992) is made available.

If you define this macro to a value greater than or equal to `199309L`, then the functionality from the 1993 edition of the POSIX.1b standard (IEEE Standard 1003.1b-1993) is made available.

Greater values for `_POSIX_C_SOURCE` will enable future extensions. The POSIX standards process will define these values as necessary, and the GNU C Library should support them some time after they become standardized. The 1996 edition of POSIX.1 (ISO/IEC 9945-1: 1996) states that if you define `_POSIX_C_SOURCE` to a value greater than or equal to `199506L`, then the functionality from the 1996 edition is made available.

_BSD_SOURCE Macro

If you define this macro, functionality derived from 4.3 BSD Unix is included as well as the ISO C, POSIX.1, and POSIX.2 material.

Some of the features derived from 4.3 BSD Unix conflict with the corresponding features specified by the POSIX.1 standard. If this macro is defined, the 4.3 BSD definitions take precedence over the POSIX definitions.

Due to the nature of some of the conflicts between 4.3 BSD and POSIX.1, you need to use a special BSD *compatibility library* when linking programs compiled for BSD compatibility. This is because some functions must be defined in two different ways, one in the normal C library, and one in the compatibility library. If your program defines `_BSD_SOURCE`, you must give the option '`-lbsd-compat`' to the compiler or linker when linking the program, to tell it to find functions in this special compatibility library before looking for them in the normal C library.

_SVID_SOURCE Macro

If you define this macro, functionality derived from SVID is included as well as the ISO C, POSIX.1, POSIX.2 and X/Open material.

_XOPEN_SOURCE Macro
_XOPEN_SOURCE_EXTENDED Macro

If you define this macro, functionality described in the *X/Open Portability Guide*[15] is included. This is a superset of the POSIX.1 and POSIX.2 functionality and in fact `_POSIX_SOURCE` and `_POSIX_C_SOURCE` are automatically defined.

As the unification of all Unices, functionality only available in BSD and SVID is also included.

If the macro `_XOPEN_SOURCE_EXTENDED` is also defined, even more functionality is available. The extra functions will make all functions available that are necessary for the X/Open Unix brand.

If the macro `_XOPEN_SOURCE` has the value 500, this includes all functionality described so far plus some new definitions from the Single Unix Specification, version 2.

_LARGEFILE_SOURCE Macro

If this macro is defined, some extra functions are available that rectify a few shortcomings in all previous standards. Specifically, the functions `fseeko` and `ftello` are available. Without these functions, the difference between the ISO C interface (`fseek`, `ftell`) and the low-level POSIX interface (`lseek`) would lead to problems.

This macro was introduced as part of the Large File Support extension (LFS).

_LARGEFILE64_SOURCE Macro

If you define this macro, an additional set of functions is made available that enables 32-bit systems to use files of sizes beyond the usual limit of 2GB. This interface is not available if the system does not support files that large. On systems where the natural file size limit is greater than 2GB (i.e., on 64-bit systems), the new functions are identical to the replaced functions.

The new functionality is made available by a new set of types and functions that replace the existing ones. The names of these new objects contain `64` to indicate the intention, e.g., `off_t` vs. `off64_t` and `fseeko` vs. `fseeko64`.

This macro was introduced as part of the Large File Support extension (LFS). It is a transition interface for the period when 64-bit offsets are not generally used (see `_FILE_OFFSET_BITS`).

---

[15] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

_FILE_OFFSET_BITS Macro

This macro determines which file-system interface will be used, one replacing the other. Whereas _LARGEFILE64_SOURCE makes the 64-bit interface available as an additional interface, _FILE_OFFSET_BITS allows the 64-bit interface to replace the old interface.

If _FILE_OFFSET_BITS is undefined, or if it is defined to the value 32, nothing changes. The 32-bit interface is used and types like off_t have a size of 32 bits on 32-bit systems.

If the macro is defined to the value 64, the large file interface replaces the old interface. The functions are not made available under different names (as they are with _LARGEFILE64_SOURCE); instead, the old function names now reference the new functions, e.g., a call to fseeko now indeed calls fseeko64.

This macro should only be selected if the system provides mechanisms for handling large files. On 64-bit systems this macro has no effect since the *64 functions are identical to the normal functions.

This macro was introduced as part of the Large File Support extension (LFS).

_ISOC99_SOURCE Macro

Until the revised ISO C standard is widely adopted the new features are not automatically enabled. The GNU libc nevertheless has a complete implementation of the new standard. To enable the new features the macro _ISOC99_SOURCE should be defined.

_GNU_SOURCE Macro

If you define this macro, everything is included: ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS, and GNU extensions. In the cases where POSIX.1 conflicts with BSD, the POSIX definitions take precedence.

If you want to get the full effect of _GNU_SOURCE but make the BSD definitions take precedence over the POSIX definitions, use this sequence of definitions:

```
#define _GNU_SOURCE
#define _BSD_SOURCE
#define _SVID_SOURCE
```

If you do this, you must link your program with the BSD compatibility library by passing the '-lbsd-compat' option to the compiler or linker. If you forget, you may get very strange errors at run time.

_REENTRANT Macro
_THREAD_SAFE Macro

If you define one of these macros, reentrant versions of several functions get declared. Some of the functions are specified in POSIX.1c, but many others are only available on a few other systems or are unique to GNU libc. The problem is the delay in the standardization of the thread safe C library interface.

Unlike on some other systems, no special version of the C library must be used for linking. There is only one version—but while compiling this, it must have been specified to compile as thread safe.

We recommend you use _GNU_SOURCE in new programs. If you don't specify the '-ansi' option to GCC and do not define any of these macros explicitly, the effect is the same as defining _POSIX_C_SOURCE to 2 and _POSIX_SOURCE, _SVID_SOURCE and _BSD_SOURCE to 1.

When you define a feature-test macro to request a larger class of features, it is harmless to define, in addition, a feature-test macro for a subset of those features. For example, if you define _POSIX_C_SOURCE, then defining _POSIX_SOURCE as well has no effect. Likewise, if you define _GNU_SOURCE, defining either _POSIX_SOURCE, _POSIX_C_SOURCE, or _SVID_SOURCE as well has no effect.

Note, however, that the features of _BSD_SOURCE are not a subset of any of the other feature-test macros supported. This is because it defines BSD features that take precedence over the POSIX features that are requested by the other macros. For this reason, defining _BSD_SOURCE in addition to the other feature-test macros does have an effect—it causes the BSD features to take priority over the conflicting POSIX features.

1.4 Road Map to the Manual

Here is an overview of the contents of the remaining chapters of this manual.

The following chapters are found in the first volume, Sandra Loosemore et al., *GNU C Library: Application Fundamentals* (Boston: GNU Press, 2004), available online at http://www.gnu.org/manual/manual.html.

- "Error Reporting" describes how errors detected by the library are reported.
- "Virtual Memory Allocation and Paging" describes the GNU library's facilities for managing and using virtual and real memory, including dynamic allocation of virtual memory. If you do not know in advance how much memory your program needs, you can allocate it dynamically instead, and manipulate it via pointers.
- "Character Handling" contains information about character-classification functions (such as isspace) and functions for performing case conversion.
- "String and Array Utilities" has descriptions of functions for manipulating strings (null-terminated character arrays) and general byte arrays, including operations such as copying and comparison.
- "Character-Set Handling" contains information about manipulating characters and strings using character sets larger than will fit in the usual char data type.
- "Locales and Internationalization" describes how selecting a particular country or language affects the behavior of the library. For example, the locale affects collation sequences for strings and how monetary values are formatted.
- "Mathematics" contains information about the math library functions. These include things like random-number generators and remainder functions on integers as well as the usual trigonometric and exponential functions on floating-point numbers.

- "Arithmetic Functions" describes functions for simple arithmetic, analysis of floating-point values, and reading numbers from strings.
- "Date and Time" describes functions for measuring both calendar time and CPU time, as well as functions for setting alarms and timers.
- "Message Translation" describes how to write programs that are capable of delivering messages in whatever language the user selects without filling the source code with sets of translations.
- "Searching and Sorting" contains information about functions for searching and sorting arrays. You can use these functions on any kind of array by providing an appropriate comparison function.
- "Pattern Matching" presents functions for matching regular expressions and shell file-name patterns, and for expanding words as the shell does.
- "The Basic Program/System Interface" tells how your programs can access their command-line arguments and environment variables.
- "Input/Output Overview" gives an overall look at the input and output facilities in the library, and contains information about basic concepts such as file names.
- "Debugging Support" describes functions provided by the library to make the debugging process easier, whether or not a dedicated debugger program is being used.
- "Input/Output on Streams" describes I/O operations involving streams (or `FILE *` objects). These are the normal C library functions from '`stdio.h`'.
- "Summary of Library Facilities" gives a summary of all the functions, variables, and macros in the library, with complete data types and function prototypes, and says what standard or system each is derived from. This section is also found in the second volume, for convenient reference.

The following chapters are found in the second volume, Sandra Loosemore et al., *GNU C Library: System & Network Applications* (Boston: GNU Press, 2004), available online at `http://www.gnu.org/manual/manual.html`.

- "Low-Level Input/Output" contains information about I/O operations on file descriptors. File descriptors are a lower-level mechanism specific to the Unix family of operating systems.
- "File-System Interface" has descriptions of operations on entire files, such as functions for deleting and renaming them and for creating new directories. This chapter also contains information about how you can access the attributes of a file, such as its owner and file-protection modes.
- "Pipes and FIFOs" contains information about simple interprocess-communication mechanisms. Pipes allow communication between two related processes (such as between a parent and child), while FIFOs allow communication between processes sharing a common file-system on the same machine.
- "Sockets" describes a more complicated interprocess-communication mechanism that allows processes running on different machines to communicate

over a network. This chapter also contains information about Internet host-addressing and how to use the system network databases.

- "Low-Level Terminal Interface" describes how you can change the attributes of a terminal device. If you want to disable echo of characters typed by the user, for example, read this chapter.
- "Processes" contains information about how to start new processes and run programs.
- "Job Control" describes functions for manipulating process groups and the controlling terminal. This material is probably only of interest if you are writing a shell or other program that handles job control specially.
- "System Databases and Name-Service Switch" describes the services that are available for looking up names in the system databases, how to determine which service is used for which database, and how these services are implemented so that contributors can design their own services.
- "Users and Groups" tells you how to access the system user- and group-databases.
- "System Management" describes functions for controlling and getting information about the hardware and software configuration your program is executing under.
- "System Configuration Parameters" tells you how you can get information about various operating system limits. Most of these parameters are provided for compatibility with POSIX.
- "DES Encryption and Password Handling" discusses the legal and technical issues related to password encryption and security, as well as the functions necessary to implement effective encryption.
- "Resource Usage and Limitation" tells you how to monitor the memory and other resource usage totals of processes, and how to regulate this usage. It also covers prioritization and scheduling.
- "Syslog" describes facilities for issuing and logging messages of system administration interest.
- "Nonlocal Exits" contains descriptions of the `setjmp` and `longjmp` functions. These functions provide a facility for `goto`-like jumps that can jump from one function to another.
- "Signal Handling" tells you all about signals—what they are, how to establish a handler that is called when a particular kind of signal is delivered, and how to prevent signals from arriving during critical sections of your program.
- "POSIX Threads" describes the pthreads (POSIX threads) library. This library provides support functions for multithreaded programs: thread primitives, synchronization objects, etc. It also implements POSIX 1003.1b semaphores.
- "C Language Facilities in the Library" contains information about library support for standard parts of the C language, including things like the `sizeof` operator and the symbolic constant `NULL`, how to write functions accepting variable numbers of arguments, and constants describing the ranges and other

properties of the numerical types. There is also a simple debugging mechanism that allows you to put assertions in your code and have diagnostic messages printed if the tests fail.

- "Installing the GNU C Library" provides a detailed reference for installing, compiling and configuring the GNU C Library. Configuration and optimization command-line options are covered here.
- "Library Maintenance" explains how to port and enhance the GNU C Library and how to report any bugs you might find.

If you already know the name of the facility you are interested in, you can look it up in Appendix A [Summary of Library Facilities], page 521. This gives you a summary of its syntax and a pointer to where you can find a more detailed description. This appendix is particularly useful if you just want to verify the order and type of arguments to a function, for example. It also tells you what standard or system each function, variable, or macro is derived from.

2 Error Reporting

Many functions in the GNU C Library detect and report error conditions, and sometimes your programs need to check for these error conditions. For example, when you open an input file, you should verify that the file was actually opened correctly, and print an error message or take other appropriate action if the call to the library function failed.

This chapter describes how the error-reporting facility works. Your program should include the header file ‘errno.h’ to use this facility.

2.1 Checking for Errors

Most library functions return a special value to indicate that they have failed. The special value is typically -1, a null pointer, or a constant such as EOF that is defined for that purpose. But this return value tells you only that an error has occurred. To find out what kind of error it was, you need to look at the error code stored in the variable errno. This variable is declared in the header file ‘errno.h’.

volatile int **errno** Variable

The variable errno contains the system error number. You can change the value of errno.

Since errno is declared volatile, it might be changed asynchronously by a signal handler.[1] However, a properly written signal handler saves and restores the value of errno, so you generally do not need to worry about this possibility except when writing signal handlers.

The initial value of errno at program start-up is zero. Many library functions are guaranteed to set it to certain nonzero values when they encounter certain kinds of errors. These error conditions are listed for each function. These functions do not change errno when they succeed; thus, the value of errno after a successful call is not necessarily zero, and you should not use errno to determine *whether* a call failed. The proper way to do that is documented for each function. *If* the call failed, you can examine errno.

Many library functions can set errno to a nonzero value as a result of calling other library functions that might fail. You should assume that any library function might alter errno when the function returns an error.

Portability Note: ISO C specifies errno as a "modifiable lvalue" rather than as a variable, permitting it to be implemented as a macro. For example, its expansion might involve a function call, like *_errno (). In fact, that is what it is on the GNU system itself. The GNU library, on non-GNU systems, does whatever is right for the particular system.

There are a few library functions, like sqrt and atan, that return a perfectly legitimate value in case of an error, but also set errno. For these functions, if

[1] See Loosemore et al., “Defining Handlers” (see chap. 1, n. 1).

you want to check to see whether an error occurred, the recommended method is to set `errno` to zero before calling the function, and then check its value afterward.

All the error codes have symbolic names; they are macros defined in '`errno.h`'. The names start with '`E`' and an uppercase letter or digit; you should consider names of this form to be reserved names (see Section 1.3.3 [Reserved Names], page 6).

The error code values are all positive integers and are all distinct, with one exception: `EWOULDBLOCK` and `EAGAIN` are the same. Since the values are distinct, you can use them as labels in a `switch` statement; just do not use both `EWOULDBLOCK` and `EAGAIN`. Your program should not make any other assumptions about the specific values of these symbolic constants.

The value of `errno` doesn't necessarily have to correspond to any of these macros, since some library functions might return other error codes of their own for other situations. The only values that are guaranteed to be meaningful for a particular library function are the ones that this manual lists for that function.

On non-GNU systems, almost any system call can return `EFAULT` if it is given an invalid pointer as an argument. Since this could only happen as a result of a bug in your program, and since it will not happen on the GNU system, we have saved space by not mentioning `EFAULT` in the descriptions of individual functions.

In some Unix systems, many system calls can also return `EFAULT` if given as an argument a pointer into the stack, and the kernel for some obscure reason fails in its attempt to extend the stack. If this ever happens, you should probably try using statically or dynamically allocated memory instead of stack memory on that system.

2.2 Error Codes

The error code macros are defined in the header file '`errno.h`'. All of them expand into integer constant values. Some of these error codes cannot occur on the GNU system, but they can occur using the GNU library on other systems.

`int` **EPERM** Macro

This means the operation is not permitted; only the owner of the file (or other resource) or processes with special privileges can perform the operation.

`int` **ENOENT** Macro

This means there is no such file or directory. This is a *file does not exist* error for ordinary files that are referenced in contexts where they are expected to already exist.

`int` **ESRCH** Macro

This means no process matches the specified process ID.

`int` **EINTR** Macro

This indicates an interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

You can choose to have functions resume after a signal that is handled, rather than failing with `EINTR`.[2]

`int` **EIO** Macro

This indicates an input/output error; usually used for physical read or write errors.

`int` **ENXIO** Macro

This means there is no such device or address. The system tried to use the device represented by a file you specified, and it could not find the device. This can mean that the device file was installed incorrectly, or that the physical device is missing or not correctly attached to the computer.

`int` **E2BIG** Macro

This means that the argument list is too long; it is used when the arguments passed to a new program being executed with one of the `exec` functions[3] occupy too much memory space. This condition never arises in the GNU system.

`int` **ENOEXEC** Macro

This indicates an invalid executable file format. This condition is detected by the `exec` functions.[4]

`int` **EBADF** Macro

This indicates a bad file descriptor; for example, I/O on a descriptor that has been closed or reading from a descriptor open only for writing (or vice versa).

`int` **ECHILD** Macro

There are no child processes. This error happens on operations that are supposed to manipulate child processes when there are not any processes to manipulate.

`int` **EDEADLK** Macro

This means a deadlock was avoided; allocating a system resource would have resulted in a deadlock situation. The system does not guarantee that it will notice all such situations. This error means you got lucky and the system noticed; it might just hang.[5]

---

[2] Ibid., “Primitives Interrupted by Signals”.

[3] Ibid., “Executing a File”.

[4] Ibid.

[5] For an example, see Loosemore et al., “File Locks”.

`int` **ENOMEM** Macro
This means no memory is available. The system cannot allocate more virtual memory because its capacity is full.

`int` **EACCES** Macro
This means that permission is denied; the file permissions do not allow the attempted operation.

`int` **EFAULT** Macro
This indicates a bad address; an invalid pointer was detected. In the GNU system, this error never happens; you get a signal instead.

`int` **ENOTBLK** Macro
A file that is not a block special file was given in a situation that requires one. For example, trying to mount an ordinary file as a file system in Unix gives this error.

`int` **EBUSY** Macro
This means that a resource is busy; a system resource that cannot be shared is already in use. For example, if you try to delete a file that is the root of a currently mounted file system, you get this error.

`int` **EEXIST** Macro
This means a file exists; an existing file was specified in a context where it only makes sense to specify a new file.

`int` **EXDEV** Macro
An attempt to make an improper link across file systems was detected. This happens not only when you use `link`,[6] but also when you rename a file with `rename`.[7]

`int` **ENODEV** Macro
The wrong type of device was given to a function that expects a particular sort of device.

`int` **ENOTDIR** Macro
A file that is not a directory was specified when a directory is required.

`int` **EISDIR** Macro
A file is a directory; you cannot open a directory for writing, or create or remove hard links to it.

---

[6] Ibid., “Hard Links”.

[7] Ibid., “Renaming Files”.

`int` **EINVAL** Macro

This indicates an invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function.

`int` **EMFILE** Macro

The current process has too many files open and cannot open any more. Duplicate descriptors do count toward this limit.

In BSD and GNU, the number of open files is controlled by a resource limit that can usually be increased. If you get this error, you might want to increase the `RLIMIT_NOFILE` limit or make it unlimited.[8]

`int` **ENFILE** Macro

There are too many distinct file openings in the entire system. Note that any number of linked channels count as just one file opening.[9] This error never occurs in the GNU system.

`int` **ENOTTY** Macro

This indicates an inappropriate I/O control operation, such as trying to set terminal modes on an ordinary file.

`int` **ETXTBSY** Macro

This indicates an attempt to execute a file that is currently open for writing, or write to a file that is currently being executed. Often using a debugger to run a program is considered having it open for writing and will cause this error. (The name stands for *text file busy*.) This is not an error in the GNU system; the text is copied as necessary.

`int` **EFBIG** Macro

A file is too big; the size of a file would be larger than allowed by the system.

`int` **ENOSPC** Macro

No space is left on device; a write operation on a file failed because the disk is full.

`int` **ESPIPE** Macro

This indicates an invalid seek operation (such as on a pipe).

`int` **EROFS** Macro

An attempt was made to modify something on a read-only file system.

---

[8] Ibid., “Limiting Resource Usage”.

[9] Ibid., “Linked Channels”.

`int` **EMLINK** Macro

There are too many links; the link count of a single file would become too large. `rename` can cause this error if the file being renamed already has as many links as it can take.[10]

`int` **EPIPE** Macro

This indicates a broken pipe; there is no process reading from the other end of a pipe. Every library function that returns this error code also generates a `SIGPIPE` signal; this signal terminates the program if not handled or blocked. Thus, your program will never actually see `EPIPE` unless it has handled or blocked `SIGPIPE`.

`int` **EDOM** Macro

This indicates a domain error; used by mathematical functions when an argument value does not fall into the domain over which the function is defined.

`int` **ERANGE** Macro

This indicates a range error; used by mathematical functions when the result value is not representable because of overflow or underflow.

`int` **EAGAIN** Macro

Resource temporarily unavailable; the call might work if you try again later. The macro `EWOULDBLOCK` is another name for `EAGAIN`; they are always the same in the GNU C Library.

This error can happen in a few different situations:

- An operation that would block was attempted on an object that has nonblocking mode selected. Trying the same operation again will block until some external condition makes it possible to read, write, or connect (whatever the operation). You can use `select` to find out when the operation will be possible.[11]

  Portability Note: In many older Unix systems, this condition was indicated by `EWOULDBLOCK`, which was a distinct error code different from `EAGAIN`. To make your program portable, you should check for both codes and treat them the same.

- A temporary resource shortage made an operation impossible. `fork` can return this error. It indicates that the shortage is expected to pass, so your program can try the call again later and it may succeed. It is probably a good idea to delay for a few seconds before trying it again, to allow time for other processes to release scarce resources. Such shortages are usually fairly serious and affect the whole system, so usually an interactive program should report the error to the user and return to its command loop.

[10] Ibid., “Renaming Files”.

[11] Ibid., “Waiting for Input or Output”.

`int` **EWOULDBLOCK** Macro

In the GNU C Library, this is another name for `EAGAIN`. The values are always the same, on every operating system.

C libraries in many older Unix systems have `EWOULDBLOCK` as a separate error code.

`int` **EINPROGRESS** Macro

An operation that cannot complete immediately was initiated on an object that has nonblocking mode selected. Some functions that must always block (such as `connect`[12]) never return `EAGAIN`. Instead, they return `EINPROGRESS` to indicate that the operation has begun and will take some time. Attempts to manipulate the object before the call completes return `EALREADY`. You can use the `select` function to find out when the pending operation has completed.[13]

`int` **EALREADY** Macro

An operation is already in progress on an object that has nonblocking mode selected.

`int` **ENOTSOCK** Macro

A file that isn't a socket was specified when a socket is required.

`int` **EMSGSIZE** Macro

The size of a message sent on a socket was larger than the supported maximum size.

`int` **EPROTOTYPE** Macro

The socket type does not support the requested communications protocol.

`int` **ENOPROTOOPT** Macro

You specified a socket option that doesn't make sense for the particular protocol being used by the socket.[14]

`int` **EPROTONOSUPPORT** Macro

The socket domain does not support the requested communications protocol (perhaps because the requested protocol is completely invalid).[15]

`int` **ESOCKTNOSUPPORT** Macro

The socket type is not supported.

---

[12] Ibid., "Connecting".

[13] Ibid., "Waiting for Input or Output".

[14] Ibid., "Socket Options".

[15] Ibid., "Creating a Socket".

`int` **EOPNOTSUPP** Macro
The operation you requested is not supported. Some socket functions don't make sense for all types of sockets, and others may not be implemented for all communications protocols. In the GNU system, this error can happen for many calls when the object does not support the particular operation; it is a generic indication that the server knows nothing to do for that call.

`int` **EPFNOSUPPORT** Macro
The socket communications protocol family you requested is not supported.

`int` **EAFNOSUPPORT** Macro
The address family specified for a socket is not supported; it is inconsistent with the protocol being used on the socket.[16]

`int` **EADDRINUSE** Macro
The requested socket address is already in use.[17]

`int` **EADDRNOTAVAIL** Macro
The requested socket address is not available; for example, you tried to give a socket a name that doesn't match the local host name.[18]

`int` **ENETDOWN** Macro
A socket operation failed because the network was down.

`int` **ENETUNREACH** Macro
A socket operation failed because the subnet containing the remote host was unreachable.

`int` **ENETRESET** Macro
A network connection was reset because the remote host crashed.

`int` **ECONNABORTED** Macro
A network connection was aborted locally.

`int` **ECONNRESET** Macro
A network connection was closed for reasons outside the control of the local host, such as by the remote machine rebooting or an unrecoverable protocol violation.

`int` **ENOBUFS** Macro
The kernel's buffers for I/O operations are all in use. In GNU, this error is always synonymous with `ENOMEM`; you may get one or the other from network operations.

---

[16] Ibid., "Sockets".
[17] Ibid., "Socket Addresses".
[18] Ibid.

`int` **EISCONN** Macro
You tried to connect a socket that is already connected.[19]

`int` **ENOTCONN** Macro
The socket is not connected to anything. You get this error when you try to transmit data over a socket, without first specifying a destination for the data. For a connectionless socket (for datagram protocols, such as UDP), you get `EDESTADDRREQ` instead.

`int` **EDESTADDRREQ** Macro
No default destination address was set for the socket. You get this error when you try to transmit data over a connectionless socket, without first specifying a destination for the data with `connect`.

`int` **ESHUTDOWN** Macro
The socket has already been shut down.

`int` **ETOOMANYREFS** Macro
There are too many references; cannot splice.

`int` **ETIMEDOUT** Macro
A socket operation with a specified time-out received no response during the time-out period.

`int` **ECONNREFUSED** Macro
A remote host refused to allow the network connection (typically because it is not running the requested service).

`int` **ELOOP** Macro
Too many levels of symbolic links were encountered in looking up a file name. This often indicates a cycle of symbolic links.

`int` **ENAMETOOLONG** Macro
The file name is too long (longer than `PATH_MAX`[20]) or the host name is too long (in `gethostname` or `sethostname`[21]).

`int` **EHOSTDOWN** Macro
The remote host for a requested network connection is down.

`int` **EHOSTUNREACH** Macro
The remote host for a requested network connection is not reachable.

---

[19] Ibid., “Making a Connection”.
[20] Ibid., “Limits on File-System Capacity”.
[21] Ibid., “Host Identification”.

`int` **ENOTEMPTY** Macro
A directory was not empty, where an empty directory was expected. Typically, this error occurs when you are trying to delete a directory.

`int` **EPROCLIM** Macro
This means that the per-user limit on new processes would be exceeded by an attempted `fork`.[22]

`int` **EUSERS** Macro
The file quota system is confused because there are too many users.

`int` **EDQUOT** Macro
The user's disk quota was exceeded.

`int` **ESTALE** Macro
There is a stale NFS file handle. This indicates an internal confusion in the NFS system which is due to file system rearrangements on the server host. Repairing this condition usually requires unmounting and remounting the NFS file system on the local host.

`int` **EREMOTE** Macro
An attempt was made to NFS-mount a remote file system with a file name that already specifies an NFS-mounted file. (This is an error on some operating systems, but we expect it to work properly on the GNU system, making this error code impossible.)

`int` **EBADRPC** Macro
RPC struct is bad.

`int` **ERPCMISMATCH** Macro
RPC version is wrong.

`int` **EPROGUNAVAIL** Macro
RPC program is not available.

`int` **EPROGMISMATCH** Macro
RPC program version is wrong.

`int` **EPROCUNAVAIL** Macro
RPC procedure for program is bad.

[22] For details on the `RLIMIT_NPROC` limit, see Loosemore et al., "Limiting Resource Usage".

`int` **ENOLCK** Macro

No locks are available. This is used by the file-locking facilities.[23] This error is never generated by the GNU system, but it can result from an operation to an NFS server running another operating system.

`int` **EFTYPE** Macro

This indicates an inappropriate file type or format. The file was the wrong type for the operation, or a data file had the wrong format.

On some systems `chmod` returns this error if you try to set the sticky bit on a nondirectory file.[24]

`int` **EAUTH** Macro

There was an authentication error.

`int` **ENEEDAUTH** Macro

An authenticator is needed.

`int` **ENOSYS** Macro

Function not implemented. This indicates that the function called is not implemented at all, either in the C library itself or in the operating system. When you get this error, you can be sure that this particular function will always fail with `ENOSYS` unless you install a new version of the C library or the operating system.

`int` **ENOTSUP** Macro

Not supported. A function returns this error when certain parameter values are valid, but the functionality they request is not available. This can mean that the function does not implement a particular command or option value or flag bit at all. For functions that operate on some object given in a parameter, such as a file descriptor or a port, it might instead mean that only *that specific object* (file descriptor, port, etc.) is unable to support the other parameters given; different file descriptors might support different ranges of parameter values.

If the entire function is not available at all in the implementation, it returns `ENOSYS` instead.

`int` **EILSEQ** Macro

While decoding a multibyte character, the function came to an invalid or incomplete sequence of bytes, or the given wide character is invalid.

`int` **EBACKGROUND** Macro

In the GNU system, servers supporting the `term` protocol return this error for certain operations when the caller is not in the foreground process group of the

[23] Ibid., “File Locks”.

[24] Ibid., “Assigning File Permissions”.

terminal. Users do not usually see this error because functions such as `read` and `write` translate it into a `SIGTTIN` or `SIGTTOU` signal.[25]

`int` **EDIED** Macro
In the GNU system, opening a file returns this error when the file is translated by a program and the translator program dies while starting up, before it has connected to the file.

`int` **ED** Macro
The experienced user will know what is wrong.

`int` **EGREGIOUS** Macro
You did **what**?

`int` **EIEIO** Macro
Go home and have a glass of warm, dairy-fresh milk.

`int` **EGRATUITOUS** Macro
This error code has no purpose.

`int` **EBADMSG** Macro
Message was bad.

`int` **EIDRM** Macro
An identifier was removed.

`int` **EMULTIHOP** Macro
A multihop was attempted.

`int` **ENODATA** Macro
No data was available.

`int` **ENOLINK** Macro
A link has been severed.

`int` **ENOMSG** Macro
There is no message of the desired type.

`int` **ENOSR** Macro
There are no more streams resources.

`int` **ENOSTR** Macro
The device is not a stream.

[25] For information on process groups and these signals, see Loosemore et al., “Job Control”.

`int` **EOVERFLOW** Macro
The value is too large for the defined data type.

`int` **EPROTO** Macro
There was a protocol error.

`int` **ETIME** Macro
The timer expired.

`int` **ECANCELED** Macro
Operation canceled; an asynchronous operation was canceled before it completed.[26] When you call `aio_cancel`, the normal result is for the operations affected to complete with this error.[27]

The following error codes are defined by the Linux/i386 kernel. They are not yet documented.

`int` **ERESTART** Macro

`int` **ECHRNG** Macro

`int` **EL2NSYNC** Macro

`int` **EL3HLT** Macro

`int` **EL3RST** Macro

`int` **ELNRNG** Macro

`int` **EUNATCH** Macro

`int` **ENOCSI** Macro

`int` **EL2HLT** Macro

[26] Ibid., "Perform I/O Operations in Parallel".
[27] Ibid., "Cancellation of AIO Operations".

`int` **EBADE** Macro

`int` **EBADR** Macro

`int` **EXFULL** Macro

`int` **ENOANO** Macro

`int` **EBADRQC** Macro

`int` **EBADSLT** Macro

`int` **EDEADLOCK** Macro

`int` **EBFONT** Macro

`int` **ENONET** Macro

`int` **ENOPKG** Macro

`int` **EADV** Macro

`int` **ESRMNT** Macro

`int` **ECOMM** Macro

`int` **EDOTDOT** Macro

`int` **ENOTUNIQ** Macro

`int` **EBADFD** Macro

`int` **EREMCHG** Macro

`int` **ELIBACC** Macro

`int` **ELIBBAD** Macro

`int` **ELIBSCN** Macro

`int` **ELIBMAX** Macro

`int` **ELIBEXEC** Macro

`int` **ESTRPIPE** Macro

`int` **EUCLEAN** Macro

`int` **ENOTNAM** Macro

`int` **ENAVAIL** Macro

`int` **EISNAM** Macro

`int` **EREMOTEIO** Macro

`int` **ENOMEDIUM** Macro

`int` **EMEDIUMTYPE** Macro

2.3 Error Messages

The library has functions and variables designed to make it easy for your program to report informative error messages in the customary format about the failure of a library call. The functions `strerror` and `perror` give you the standard error message for a given error code; the variable `program_invocation_short_name` gives you convenient access to the name of the program that encountered the error.

`char *` **strerror** (`int` *errnum*) Function

The `strerror` function maps the error code (see Section 2.1 [Checking for Errors], page 17) specified by the *errnum* argument to a descriptive error message string. The return value is a pointer to this string.

The value *errnum* normally comes from the variable `errno`.

You should not modify the string returned by `strerror`. Also, if you make subsequent calls to `strerror`, the string might be overwritten. (But it's guaranteed that no library function ever calls `strerror` behind your back.)

The function `strerror` is declared in '`string.h`'.

`char *` **strerror_r** (`int` *errnum*, `char *`*buf*, `size_t` *n*) Function

The `strerror_r` function works like `strerror` but instead of returning the error message in a statically allocated buffer shared by all threads in the process, it returns a private copy for the thread. This might be either some permanent global data or a message string in the user-supplied buffer starting at *buf* with the length of *n* bytes.

At most *n* characters are written (including the NUL byte), so it is up to the user to select the buffer large enough.

This function should always be used in multithreaded programs since there is no way to guarantee that the string returned by `strerror` really belongs to the last call of the current thread.

This function `strerror_r` is a GNU extension and it is declared in '`string.h`'.

`void` **perror** (`const char *`*message*) Function

This function prints an error message to the stream `stderr` (see Section 17.2 [Standard Streams], page 439). The orientation of `stderr` is not changed.

If you call `perror` with a *message* that is either a null pointer or an empty string, `perror` just prints the error message corresponding to `errno`, adding a trailing newline.

If you supply a nonnull *message* argument, then `perror` prefixes its output with this string. It adds a colon and a space character to separate the *message* from the error string corresponding to `errno`.

The function `perror` is declared in '`stdio.h`'.

`strerror` and `perror` produce the exact same message for any given error code; the precise text varies from system to system. On the GNU system, the messages are fairly short; there are no multiline messages or embedded newlines. Each error message begins with a capital letter and does not include any terminating punctuation.

Compatibility Note: The `strerror` function was introduced in ISO C89. Many older C systems do not support this function yet.

Many programs that don't read input from the terminal are designed to exit if any system call fails. By convention, the error message from such a program should start with the program's name, sans directories. You can find that name in the variable `program_invocation_short_name`; the full file name is stored the variable `program_invocation_name`.

`char *` **program_invocation_name** Variable

This variable's value is the name that was used to invoke the program running in the current process. It is the same as `argv[0]`. This is not necessarily a useful file name; often it contains no directory names (see Section 14.1 [Program Arguments], page 379).

`char *` **program_invocation_short_name** Variable

This variable's value is the name that was used to invoke the program running in the current process, with directory names removed—it is the same as `program_invocation_name` minus everything up to the last slash, if any.

The library initialization code sets up both of these variables before calling `main`.

Portability Note: These two variables are GNU extensions. If you want your program to work with non-GNU libraries, you must save the value of `argv[0]` in `main`, and then strip off the directory names yourself. We added these extensions to make it possible to write self-contained error-reporting subroutines that require no explicit cooperation from `main`.

Here is an example showing how to handle a failure to open a file correctly. The function `open_sesame` tries to open the named file for reading and returns a stream if successful. The `fopen` library function returns a null pointer if it couldn't open the file for some reason. In that situation, `open_sesame` constructs an appropriate error message using the `strerror` function, and terminates the program. If we were going to make some other library calls before passing the error code to `strerror`, we would have to save it in a local variable instead, because those other library functions might overwrite `errno` in the meantime.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *
```

```
open_sesame (char *name)
{
  FILE *stream;

  errno = 0;
  stream = fopen (name, "r");
  if (stream == NULL)
    {
      fprintf (stderr, "%s: Couldn't open file %s; %s\n",
               program_invocation_short_name, name, strerror (errno));
      exit (EXIT_FAILURE);
    }
  else
    return stream;
}
```

Using `perror` has the advantage that the function is portable and available on all systems implementing ISO C. But often the text `perror` generates is not what is wanted, and there is no way to extend or change what `perror` does. The GNU coding standard, for instance, requires error messages to be preceded by the program name, and programs that read some input files should provide information about the input file name and the line number in case an error is encountered while reading the file. For these occasions there are two functions available which are widely used throughout the GNU project. These functions are declared in '`error.h`'.

`void` **error** (`int` *status*, `int` *errnum*, `const char *`*format*, `...`) Function

The `error` function can be used to report general problems during program execution. The *format* argument is a format string just like those given to the `printf` family of functions. The arguments required for the format can follow the *format* parameter. Just like `perror`, `error` also can report an error code in textual form. But unlike `perror` the error value is explicitly passed to the function in the *errnum* parameter. This eliminates the problem mentioned above that the error-reporting function must be called immediately after the function causing the error, since otherwise `errno` might have a different value.

The `error` prints first the program name. If the application defined a global variable `error_print_progname` and points it to a function, this function will be called to print the program name. Otherwise, the string from the global variable `program_name` is used. The program name is followed by a colon and a space, which in turn is followed by the output produced by the format string. If the *errnum* parameter is nonzero, the format string output is followed by a colon and a space, followed by the error message for the error code *errnum*. In any case, the output is terminated with a newline.

The output is directed to the `stderr` stream. If the `stderr` wasn't oriented before the call, it will be narrow-oriented afterward.

The function will return unless the *status* parameter has a nonzero value. In this case, the function will call `exit` with the *status* value for its parameter and therefore never return. If `error` returns, the global variable `error_message_count` is incremented by one to keep track of the number of errors reported.

`void` **error_at_line** (`int` *status*, `int` *errnum*, `const char *`*fname*, `unsigned int` *lineno*, `const char *`*format*, `...`) Function

The `error_at_line` function is very similar to the `error` function. The only difference are the additional parameters *fname* and *lineno*. The handling of the other parameters is identical to that of `error` except that between the program name and the string generated by the format string additional text is inserted.

Directly following the program name, a colon followed by the file name pointed to by *fname*, another colon, and a value of *lineno* are printed.

This additional output of course is meant to be used to locate an error in an input file (like a programming language source code file, for example).

If the global variable `error_one_per_line` is set to a nonzero value `error_at_line` will avoid printing consecutive messages for the same file anem line. Repetitions that are not directly following each other are not caught.

Just like `error`, this function only returns if *status* is zero. Otherwise, `exit` is called with the nonzero value. If `error` returns, the global variable `error_message_count` is incremented by one to keep track of the number of errors reported.

As mentioned above the `error` and `error_at_line` functions can be customized by defining a variable named `error_print_progname`.

`void (*` **error_print_progname** `) (void)` Variable

If the `error_print_progname` variable is defined to a nonzero value, the function pointed to is called by `error` or `error_at_line`. It is expected to print the program name or do something similarly useful.

The function is expected to be printed to the `stderr` stream, and must be able to handle whatever orientation the stream has.

The variable is global and shared by all threads.

`unsigned int` **error_message_count** Variable

The `error_message_count` variable is incremented whenever one of the functions `error` or `error_at_line` returns. The variable is global and shared by all threads.

`int` **error_one_per_line** Variable

The `error_one_per_line` variable influences only `error_at_line`. Normally the `error_at_line` function creates output for every invocation. If `error_one_per_line` is set to a nonzero value, `error_at_line`

keeps track of the last file name and line number for which an error was reported and avoids directly following messages for the same file and line. This variable is global and shared by all threads.

A program that read an input file and reports errors in it could look like this:

```
{
  char *line = NULL;
  size_t len = 0;
  unsigned int lineno = 0;

  error_message_count = 0;
  while (! feof_unlocked (fp))
    {
      ssize_t n = getline (&line, &len, fp);
      if (n <= 0)
        /* End of file or error.  */
        break;
      ++lineno;

      /* Process the line.  */
      ...

      if (Detect error in line)
        error_at_line (0, errval, filename, lineno,
                       "some error text %s", some_variable);
    }

  if (error_message_count != 0)
    error (EXIT_FAILURE, 0, "%u errors found", error_message_count);
}
```

`error` and `error_at_line` are clearly the functions of choice and enable the programmer to write applications that follow the GNU coding standard. The GNU libc also contains functions that are used in BSD for the same purpose. These functions are declared in '`err.h`'. It is generally advised to not use these functions. They are included only for compatibility.

`void` **warn** (`const char *`*format*`, ...`) Function

The `warn` function is roughly equivalent to a call like:

```
error (0, errno, format, the parameters)
```

except the global variables that `error` respects and modifies are not used.

`void` **vwarn** (`const char *`*format*`, va_list`) Function

The `vwarn` function is just like `warn` except that the parameters for the handling of the format string *format* are passed in as a value of type `va_list`.

`void` **warnx** (`const char *`*format*`, ...`) Function
The `warnx` function is roughly equivalent to a call like:

```
error (0, 0, format, the parameters)
```

except the global variables that `error` respects and modifies are not used. The difference with `warn` is that no error number string is printed.

`void` **vwarnx** (`const char *`*format*`, va_list`) Function
The `vwarnx` function is just like `warnx` except that the parameters for the handling of the format string *format* are passed in as a value of type `va_list`.

`void` **err** (`int` *status*`, const char *`*format*`, ...`) Function
The `err` function is roughly equivalent to a call like:

```
error (status, errno, format, the parameters)
```

except that the global variables `error` respects and modifies are not used and that the program is exited even if *status* is zero.

`void` **verr** (`int` *status*`, const char *`*format*`, va_list`) Function
The `verr` function is just like `err` except that the parameters for the handling of the format string *format* are passed in as a value of type `va_list`.

`void` **errx** (`int` *status*`, const char *`*format*`, ...`) Function
The `errx` function is roughly equivalent to a call like:

```
error (status, 0, format, the parameters)
```

except that the global variables `error` respects and modifies are not used and that the program is exited even if *status* is zero. The difference with `err` is that no error number string is printed.

`void` **verrx** (`int` *status*`, const char *`*format*`, va_list`) Function
The `verrx` function is just like `errx` except that the parameters for the handling of the format string *format* are passed in as a value of type `va_list`.

3 Virtual Memory Allocation and Paging

This chapter describes how processes manage and use memory in a system that uses the GNU C Library.

The GNU C Library has several functions for dynamically allocating virtual memory in various ways. They vary in generality and in efficiency. The library also provides functions for controlling paging and allocation of real memory.

Memory-mapped I/O is not discussed in this chapter.[1]

3.1 Process Memory Concepts

One of the most basic resources a process has available to it is memory. There are many different ways systems organize memory, but in a typical one, each process has one linear virtual address space, with addresses running from zero to some huge maximum. It need not be contiguous; i.e. not all of these addresses actually can be used to store data.

The virtual memory is divided into pages (4 kilobytes is typical). Backing each page of virtual memory is a page of real memory (called a *frame*) or some secondary storage, usually disk space. The disk space might be swap space or just some ordinary disk file. Actually, a page of all zeros sometimes has nothing at all backing it—there's just a flag saying it is all zeros.

The same frame of real memory or backing store can back multiple virtual pages belonging to multiple processes. This is normally the case, for example, with virtual memory occupied by GNU C Library code. The same real memory frame containing the `printf` function backs a virtual memory page in each of the existing processes that has a `printf` call in its program.

In order for a program to access any part of a virtual page, the page must at that moment be backed by (*connected to*) a real frame. But because there is usually a lot more virtual memory than real memory, the pages must move back and forth between real memory and backing store regularly, coming into real memory when a process needs to access them and then retreating to backing store when not needed anymore. This movement is called *paging*.

When a program attempts to access a page which is not at that moment backed by real memory, this is known as a *page fault*. When a page fault occurs, the kernel suspends the process, places the page into a real page frame (this is called "paging in" or "faulting in"), then resumes the process so that from the process' point of view, the page was in real memory all along. In fact, to the process, all pages always seem to be in real memory. Except for one thing: the elapsed execution time of an instruction that would normally be a few nanoseconds is suddenly much, much, longer (because the kernel normally has to do I/O to complete the page-in). For programs sensitive to that, the functions described in Section 3.4 [Locking Pages], page 74 can control it.

[1] See Loosemore et al., "Memory-Mapped I/O" (see chap. 1, n. 1).

Within each virtual address space, a process has to keep track of what is at which addresses, and that process is called memory allocation. Allocation usually brings to mind meting out scarce resources, but in the case of virtual memory, that's not a major goal, because there is generally much more of it than anyone needs. Memory allocation within a process is mainly just a matter of making sure that the same byte of memory isn't used to store two different things.

Processes allocate memory in two major ways: by exec and programmatically. Actually, forking is a third way, but it's not very interesting.[2]

Exec is the operation of creating a virtual address space for a process, loading its basic program into it, and executing the program. It is done by the *exec* family of functions (`execl`, for example). The operation takes a program file (an executable), allocates space to load all the data in the executable, loads it, and transfers control to it. The most notable data are the instructions of the program (the *text*), but also literals and constants in the program and even some variables—C variables with the static storage class (see Section 3.2.1 [Memory Allocation in C Programs], page 41).

Once that program begins to execute, it uses programmatic allocation to gain additional memory. In a C program with the GNU C Library, there are two kinds of programmatic allocation: automatic and dynamic (see Section 3.2.1 [Memory Allocation in C Programs], page 41).

Memory-mapped I/O is another form of dynamic virtual memory allocation. Mapping memory to a file means declaring that the contents of certain range of a process' addresses shall be identical to the contents of a specified regular file. The system makes the virtual memory initially contain the contents of the file, and if you modify the memory, the system writes the same modification to the file. Note that due to the magic of virtual memory and page faults, there is no reason for the system to do I/O to read the file, or allocate real memory for its contents, until the program accesses the virtual memory.[3]

Just as it programmatically allocates memory, the program can programmatically deallocate (*free*) it. You can't free the memory that was allocated by exec. When the program exits or execs, you might say that all its memory gets freed, but since in both cases the address space ceases to exist, the point is really moot (see Section 14.6 [Program Termination], page 425).

A process' virtual address space is divided into segments. A segment is a contiguous range of virtual addresses. Three important segments are

- The *text segment* contains a program's instructions and literals and static constants. It is allocated by exec and stays the same size for the life of the virtual address space.
- The *data segment* is working storage for the program. It can be preallocated and preloaded by exec and the process can extend or shrink it by calling functions as described in Section 3.3 [Resizing the Data Segment], page 74. Its lower end is fixed.

[2] Ibid., "Creating a Process".

[3] Ibid., "Memory-Mapped I/O".

- The *stack segment* contains a program stack. It grows as the stack grows, but doesn't shrink when the stack shrinks.

3.2 Allocating Storage for Program Data

This section covers how ordinary programs manage storage for their data, including the famous `malloc` function and some fancier facilities that are special to the GNU C Library and GNU Compiler.

3.2.1 Memory Allocation in C Programs

The C language supports two kinds of memory allocation through the variables in C programs:

- *Static allocation* is what happens when you declare a static or global variable. Each static or global variable defines one block of space, of a fixed size. The space is allocated once, when your program is started (part of the exec operation), and is never freed.
- *Automatic allocation* happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.

  In GNU C, the size of the automatic storage can be an expression that varies. In other C implementations, it must be a constant.

A third important kind of memory allocation, *dynamic allocation*, is not supported by C variables but is available via GNU C Library functions.

3.2.1.1 Dynamic Memory Allocation

Dynamic memory allocation is a technique in which programs determine as they are running where to store some information. You need dynamic allocation when the amount of memory you need, or how long you continue to need it, depends on factors that are not known before the program runs.

For example, you may need a block to store a line read from an input file; since there is no limit to how long a line can be, you must allocate the memory dynamically and make it dynamically larger as you read more of the line.

Or, you may need a block for each record or each definition in the input data; since you can't know in advance how many there will be, you must allocate a new block for each record or definition as you read it.

When you use dynamic allocation, the allocation of a block of memory is an action that the program requests explicitly. You call a function or macro when you want to allocate space, and specify the size with an argument. If you want to free the space, you do so by calling another function or macro. You can do these things whenever you want, as often as you want.

Dynamic allocation is not supported by C variables; there is no storage class *dynamic*, and there can never be a C variable whose value is stored in dynamically allocated space. The only way to get dynamically allocated memory is via a system call (which is generally via a GNU C Library function call), and the only way to refer to dynamically allocated space is through a pointer. Because it is less convenient, and because the actual process of dynamic allocation requires more computation time, programmers generally use dynamic allocation only when neither static nor automatic allocation will serve.

For example, if you want to allocate dynamically some space to hold a `struct foobar`, you cannot declare a variable of type `struct foobar` whose contents are the dynamically allocated space. But you can declare a variable of pointer type `struct foobar *` and assign it the address of the space. Then you can use the operators '`*`' and '`->`' on this pointer variable to refer to the contents of the space:

```
{
  struct foobar *ptr
    = (struct foobar *) malloc (sizeof (struct foobar));
  ptr->name = x;
  ptr->next = current_foobar;
  current_foobar = ptr;
}
```

3.2.2 Unconstrained Allocation

The most general dynamic allocation facility is `malloc`. It allows you to allocate blocks of memory of any size at any time, make them bigger or smaller at any time, and free the blocks individually at any time (or never).

3.2.2.1 Basic Memory Allocation

To allocate a block of memory, call `malloc`. The prototype for this function is in '`stdlib.h`'.

`void *` **malloc** (`size_t` *size*) Function

This function returns a pointer to a newly allocated block *size* bytes long, or a null pointer if the block could not be allocated.

The contents of the block are undefined; you must initialize it yourself (or use `calloc` instead; see Section 3.2.2.5 [Allocating Cleared Space], page 46). Normally you would cast the value as a pointer to the kind of object that you want to store in the block. Here we show an example of doing so, and of initializing the space with zeros using the library function `memset` (see Section 5.4 [Copying and Concatenation], page 93):

```
struct foo *ptr;
...
ptr = (struct foo *) malloc (sizeof (struct foo));
```

```
if (ptr == 0) abort ();
memset (ptr, 0, sizeof (struct foo));
```

You can store the result of `malloc` into any pointer variable without a cast, because ISO C automatically converts the type `void *` to another type of pointer when necessary. But the cast is necessary in contexts other than assignment operators or if you might want your code to run in traditional C.

Remember that when allocating space for a string, the argument to `malloc` must be one plus the length of the string. This is because a string is terminated with a null character that doesn't count in the "length" of the string but does need space. For example:

```
char *ptr;
...
ptr = (char *) malloc (length + 1);
```

See Section 5.1 [Representation of Strings], page 89, for more information about this.

3.2.2.2 Examples of `malloc`

If no more space is available, `malloc` returns a null pointer. You should check the value of *every* call to `malloc`. It is useful to write a subroutine that calls `malloc` and reports an error if the value is a null pointer, returning only if the value is nonzero. This function is conventionally called `xmalloc`. Here it is:

```
void *
xmalloc (size_t size)
{
  register void *value = malloc (size);
  if (value == 0)
    fatal ("virtual memory exhausted");
  return value;
}
```

Here is a real example of using `malloc` (by way of `xmalloc`). The function `savestring` will copy a sequence of characters into a newly allocated null-terminated string:

```
char *
savestring (const char *ptr, size_t len)
{
  register char *value = (char *) xmalloc (len + 1);
  value[len] = '\0';
  return (char *) memcpy (value, ptr, len);
}
```

The block that `malloc` gives you is guaranteed to be aligned so that it can hold any type of data. In the GNU system, the address is always a multiple of eight on most systems, and a multiple of sixteen on 64-bit systems. Only rarely is any higher boundary (such as a page boundary) necessary; for those cases, use `memalign`,

`posix_memalign` or `valloc` (see Section 3.2.2.7 [Allocating Aligned Memory Blocks], page 47).

Note that the memory located after the end of the block is likely to be in use for something else; perhaps a block already allocated by another call to `malloc`. If you attempt to treat the block as longer than you asked for it to be, you are liable to destroy the data that `malloc` uses to keep track of its blocks, or you may destroy the contents of another block. If you have already allocated a block and discover you want it to be bigger, use `realloc` (see Section 3.2.2.4 [Changing the Size of a Block], page 45).

3.2.2.3 Freeing Memory Allocated with `malloc`

When you no longer need a block that you got with `malloc`, use the function `free` to make the block available to be allocated again. The prototype for this function is in '`stdlib.h`'.

`void` **free** (`void *`*ptr*) Function

The `free` function deallocates the block of memory pointed at by *ptr*.

`void` **cfree** (`void *`*ptr*) Function

This function does the same thing as `free`. It's provided for backward compatibility with SunOS; you should use `free` instead.

Freeing a block alters the contents of the block. **Do not expect to find any data (such as a pointer to the next block in a chain of blocks) in the block after freeing it.** Copy whatever you need out of the block before freeing it! Here is an example of the proper way to free all the blocks in a chain, and the strings that they point to:

```
struct chain
  {
    struct chain *next;
    char *name;
  }

void
free_chain (struct chain *chain)
{
  while (chain != 0)
    {
      struct chain *next = chain->next;
      free (chain->name);
      free (chain);
      chain = next;
    }
}
```

Occasionally, `free` can actually return memory to the operating system and make the process smaller. Usually, all it can do is allow a later call to `malloc` to reuse the space. In the meantime, the space remains in your program as part of a free-list used internally by `malloc`.

There is no point in freeing blocks at the end of a program, because all of the program's space is given back to the system when the process terminates.

3.2.2.4 Changing the Size of a Block

Often you do not know for certain how big a block you will ultimately need at the time you must begin to use the block. For example, the block might be a buffer that you use to hold a line being read from a file; no matter how long you make the buffer initially, you may encounter a line that is longer.

You can make the block longer by calling `realloc`. This function is declared in '`stdlib.h`'.

`void *` **realloc** (`void *`*ptr*`,` `size_t` *newsize*) Function

The `realloc` function changes the size of the block whose address is *ptr* to be *newsize*.

Since the space after the end of the block may be in use, `realloc` may find it necessary to copy the block to a new address where more free space is available. The value of `realloc` is the new address of the block. If the block needs to be moved, `realloc` copies the old contents.

If you pass a null pointer for *ptr*, `realloc` behaves just like '`malloc (`*newsize*`)`'. This can be convenient, but be aware that older implementations (before ISO C) may not support this behavior, and will probably crash when `realloc` is passed a null pointer.

Like `malloc`, `realloc` may return a null pointer if no memory space is available to make the block bigger. When this happens, the original block is untouched; it has not been modified or relocated.

In most cases it makes no difference what happens to the original block when `realloc` fails, because the application program cannot continue when it is out of memory, and the only thing to do is to give a fatal error message. Often it is convenient to write and use a subroutine, conventionally called `xrealloc`, that takes care of the error message as `xmalloc` does for `malloc`:

```
void *
xrealloc (void *ptr, size_t size)
{
  register void *value = realloc (ptr, size);
  if (value == 0)
    fatal ("Virtual memory exhausted");
  return value;
}
```

You can also use `realloc` to make a block smaller. The reason you would do this is to avoid tying up a lot of memory space when only a little is needed. In several allocation implementations, making a block smaller sometimes necessitates copying it, so it can fail if no other space is available.

If the new size you specify is the same as the old size, `realloc` is guaranteed to change nothing and return the same address that you gave.

3.2.2.5 Allocating Cleared Space

The function `calloc` allocates memory and clears it to zero. It is declared in '`stdlib.h`'.

`void *` **calloc** (`size_t` *count*, `size_t` *eltsize*) Function

This function allocates a block long enough to contain a vector of *count* elements, each of size *eltsize*. Its contents are cleared to zero before `calloc` returns.

You could define `calloc` as follows:

```
void *
calloc (size_t count, size_t eltsize)
{
  size_t size = count * eltsize;
  void *value = malloc (size);
  if (value != 0)
    memset (value, 0, size);
  return value;
}
```

But in general, it is not guaranteed that `calloc` calls `malloc` internally. Therefore, if an application provides its own `malloc`/`realloc`/`free` outside the C library, it should always define `calloc`, too.

3.2.2.6 Efficiency Considerations for `malloc`

As opposed to other versions, the `malloc` in the GNU C Library does not round up block sizes to powers of two, either for large or small sizes. Neighboring chunks can be coalesced on a `free` no matter what their size is. This makes the implementation suitable for all kinds of allocation patterns, without generally incurring high memory waste through fragmentation.

Very large blocks (much larger than a page) are allocated with `mmap` (anonymous or via `/dev/zero`) by this implementation. This has the great advantage that these chunks are returned to the system immediately when they are freed. Therefore, it cannot happen that a large chunk becomes *locked* between smaller ones and, even after calling `free`, wastes memory. The size threshold for `mmap` to be used can be adjusted with `mallopt`. The use of `mmap` can also be disabled completely.

3.2.2.7 Allocating Aligned Memory Blocks

The address of a block returned by `malloc` or `realloc` in the GNU system is always a multiple of eight (or sixteen on 64-bit systems). If you need a block whose address is a multiple of a higher power of two than that, use `memalign`, `posix_memalign`, or `valloc`. `memalign` is declared in '`malloc.h`' and `posix_memalign` is declared in '`stdlib.h`'.

With the GNU library, you can use `free` to free the blocks that `memalign`, `posix_memalign`, and `valloc` return. That does not work in BSD, however—BSD does not provide any way to free such blocks.

`void *` **memalign** (`size_t` *boundary*, `size_t` *size*) Function

The `memalign` function allocates a block of *size* bytes whose address is a multiple of *boundary*. The *boundary* must be a power of two! The function `memalign` works by allocating a somewhat larger block, and then returning an address within the block that is on the specified boundary.

`int` **posix_memalign** (`void **`*memptr*, `size_t` *alignment*, `size_t` *size*) Function

The `posix_memalign` function is similar to the `memalign` function in that it returns a buffer of *size* bytes aligned to a multiple of *alignment*. But it adds one requirement to the parameter *alignment*: the value must be a power-of-two multiple of `sizeof (void *)`.

If the function succeeds in allocating memory, a pointer to the allocated memory is returned in **memptr* and the return value is zero. Otherwise, the function returns an error value indicating the problem.

This function was introduced in POSIX 1003.1d.

`void *` **valloc** (`size_t` *size*) Function

Using `valloc` is like using `memalign` and passing the page size as the value of the second argument. It is implemented like this:[4]

```
void *
valloc (size_t size)
{
  return memalign (getpagesize (), size);
}
```

3.2.2.8 `malloc` Tunable Parameters

You can adjust some parameters for dynamic memory allocation with the `mallopt` function. This function is the general SVID/XPG interface, defined in '`malloc.h`'.

[4] See Loosemore et al., "Query Memory Parameters", for more information about the memory subsystem.

`int` **mallopt** (`int` *param*, `int` *value*) Function

When calling `mallopt`, the *param* argument specifies the parameter to be set, and *value* the new value to be set. Possible choices for *param*, as defined in '`malloc.h`', are

`M_TRIM_THRESHOLD`
: This is the minimum size (in bytes) of the topmost, releasable chunk that will cause `sbrk` to be called with a negative argument in order to return memory to the system.

`M_TOP_PAD`
: This parameter determines the amount of extra memory to obtain from the system when a call to `sbrk` is required. It also specifies the number of bytes to retain when shrinking the heap by calling `sbrk` with a negative argument. This provides the necessary hysteresis in heap size such that excessive amounts of system calls can be avoided.

`M_MMAP_THRESHOLD`
: All chunks larger than this value are allocated outside the normal heap, using the `mmap` system call. This way it is guaranteed that the memory for these chunks can be returned to the system on `free`. Requests smaller than this threshold might still be allocated via `mmap`.

`M_MMAP_MAX`
: The maximum number of chunks to allocate with `mmap`. Setting this to zero disables all use of `mmap`.

3.2.2.9 Heap Consistency Checking

You can ask `malloc` to check the consistency of dynamic memory by using the `mcheck` function. This function is a GNU extension, declared in '`mcheck.h`'.

`int` **mcheck** (`void` (`*`*abortfn*) (`enum mcheck_status` *status*)) Function

Calling `mcheck` tells `malloc` to perform occasional consistency checks. These will catch things such as writing past the end of a block that was allocated with `malloc`.

The *abortfn* argument is the function to call when an inconsistency is found. If you supply a null pointer, then `mcheck` uses a default function that prints a message and calls `abort` (see Section 14.6.4 [Aborting a Program], page 427). The function you supply is called with one argument, which says what sort of inconsistency was detected; its type is described below.

It is too late to begin allocation checking once you have allocated anything with `malloc`, so `mcheck` does nothing in that case. The function returns `-1` if you call it too late, and `0` otherwise (when it is successful).

The easiest way to arrange to call `mcheck` early enough is to use the option '`-lmcheck`' when you link your program; then you don't need to modify your program source at all. Alternatively, you might use a debugger to insert a call to `mcheck` whenever the program is started. For example, these gdb commands will automatically call `mcheck` whenever the program starts:

```
(gdb) break main
Breakpoint 1, main (argc=2, argv=0xbffff964) at whatever.c:10
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>call mcheck(0)
>continue
>end
(gdb) ...
```

However, this will only work if no initialization function of any object involved calls any of the `malloc` functions, since `mcheck` must be called before the first such function.

`enum mcheck_status` **mprobe** (`void *`*pointer*) Function

The `mprobe` function lets you explicitly check for inconsistencies in a particular allocated block. You must have already called `mcheck` at the beginning of the program to do its occasional checks; calling `mprobe` requests an additional consistency check to be done at the time of the call.

The argument *pointer* must be a pointer returned by `malloc` or `realloc`. `mprobe` returns a value that says what inconsistency, if any, was found. The values are described below.

enum mcheck_status Data Type

This enumerated type describes what kind of inconsistency was detected in an allocated block, if any. Here are the possible values:

`MCHECK_DISABLED`
: `mcheck` was not called before the first allocation. No consistency checking can be done.

`MCHECK_OK`
: No inconsistency was detected.

`MCHECK_HEAD`
: The data immediately before the block were modified. This commonly happens when an array index or pointer is decremented too far.

`MCHECK_TAIL`
: The data immediately after the block were modified. This commonly happens when an array index or pointer is incremented too far.

`MCHECK_FREE`
: The block was already freed.

Another way to check for and guard against bugs in the use of `malloc`, `realloc` and `free` is to set the environment variable `MALLOC_CHECK_`. When `MALLOC_CHECK_` is set, a special (less efficient) implementation is used, which is designed to be tolerant against simple errors, such as double calls of `free` with the same argument, or overruns of a single byte (off-by-one bugs). Not all such errors can be protected against, however, and memory leaks can result. If `MALLOC_CHECK_` is set to `0`, any detected heap corruption is silently ignored; if set to `1`, a diagnostic is printed on `stderr`; if set to `2`, `abort` is called immediately. This can be useful because otherwise a crash may happen much later, and the true cause for the problem is then very hard to track down.

There is one problem with `MALLOC_CHECK_`; in SUID or SGID binaries, it could possibly be exploited since, diverging from the normal program behavior, it now writes something to the standard error descriptor. Therefore, the use of `MALLOC_CHECK_` is disabled by default for SUID and SGID binaries. It can be enabled again by the system administrator by adding a file '`/etc/suid-debug`' (the content is not important—it could be empty).

So, what is the difference between using `MALLOC_CHECK_` and linking with '`-lmcheck`'? `MALLOC_CHECK_` is orthogonal with respect to '`-lmcheck`'. '`-lmcheck`' has been added for backward compatibility. Both `MALLOC_CHECK_` and '`-lmcheck`' should uncover the same bugs—but using `MALLOC_CHECK_`, you don't need to recompile your application.

3.2.2.10 Memory Allocation Hooks

The GNU C Library lets you modify the behavior of `malloc`, `realloc`, and `free` by specifying appropriate hook functions. You can use these hooks to help you debug programs that use dynamic memory allocation, for example.

The hook variables are declared in '`malloc.h`'.

__malloc_hook Variable

The value of this variable is a pointer to the function that `malloc` uses whenever it is called. You should define this function to look like `malloc`:

```
void *function (size_t size, const void *caller)
```

The value of *caller* is the return address found on the stack when the `malloc` function was called. This value allows you to trace the memory consumption of the program.

__realloc_hook Variable

The value of this variable is a pointer to the function that `realloc` uses whenever it is called. You should define this function to look like `realloc`:

```
void *function (void *ptr, size_t size, const void *caller)
```

The value of *caller* is the return address found on the stack when the `realloc` function was called. This value allows you to trace the memory consumption of the program.

__free_hook Variable

The value of this variable is a pointer to the function that `free` uses whenever it is called. You should define this function to look like `free`:

```
void function (void *ptr, const void *caller)
```

The value of *caller* is the return address found on the stack when the `free` function was called. This value allows you to trace the memory consumption of the program.

__memalign_hook Variable

The value of this variable is a pointer to the function that `memalign` uses whenever it is called. You should define this function to look like `memalign`:

```
void *function (size_t alignment, size_t size, const void *caller)
```

The value of *caller* is the return address found on the stack when the `memalign` function was called. This value allows you to trace the memory consumption of the program.

You must make sure that the function you install as a hook for one of these functions does not call that function recursively without restoring the old value of the hook first! Otherwise, your program will get stuck in an infinite recursion. Before calling the function recursively, you should make sure to restore all the hooks to their previous value. When coming back from the recursive call, all the hooks should be resaved, since a hook might modify itself.

__malloc_initialize_hook Variable

The value of this variable is a pointer to a function that is called once when the `malloc` implementation is initialized. This is a weak variable, so it can be overridden in the application with a definition like the following:

```
void (*__malloc_initialize_hook) (void) = my_init_hook;
```

An issue to look out for is the time at which the `malloc` hook functions can be safely installed. If the hook functions call the `malloc`-related functions recursively, it is necessary that `malloc` already have properly initialized itself when the function (`__malloc_hook`, for example) is assigned to. On the other hand, if the hook functions provide a complete `malloc` implementation of their own, it is vital that the hooks are assigned to *before* the very first `malloc` call has completed, because otherwise a chunk obtained from the ordinary, unhooked `malloc` may later be handed to, for example, `__free_hook`.

In both cases, the problem can be solved by setting up the hooks from within a user-defined function pointed to by `__malloc_initialize_hook`—then the hooks will be set up safely at the right time.

Here is an example showing how to use `__malloc_hook` and `__free_hook` properly. It installs a function that prints out information every time `malloc` or `free` is called. We just assume here that `realloc` and `memalign` are not used in our program.

```
/* Prototypes for __malloc_hook, __free_hook */
#include <malloc.h>

/* Prototypes for our hooks.  */
static void *my_init_hook (void);
static void *my_malloc_hook (size_t, const void *);
static void my_free_hook (void*, const void *);

/* Override initializing hook from the C library. */
void (*__malloc_initialize_hook) (void) = my_init_hook;

static void
my_init_hook (void)
{
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
}

static void *
my_malloc_hook (size_t size, const void *caller)
{
  void *result;
  /* Restore all old hooks */
  __malloc_hook = old_malloc_hook;
  __free_hook = old_free_hook;
  /* Call recursively */
  result = malloc (size);
  /* Save underlying hooks */
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;
  /* printf might call malloc, so protect it too. */
  printf ("malloc (%u) returns %p\n", (unsigned int) size, result);
  /* Restore our own hooks */
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
  return result;
}
```

```
static void *
my_free_hook (void *ptr, const void *caller)
{
  /* Restore all old hooks */
  __malloc_hook = old_malloc_hook;
  __free_hook = old_free_hook;
  /* Call recursively */
  free (ptr);
  /* Save underlying hooks */
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;
  /* printf might call free, so protect it too. */
  printf ("freed pointer %p\n", ptr);
  /* Restore our own hooks */
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
}

main ()
{
  ...
}
```

The `mcheck` function (see Section 3.2.2.9 [Heap Consistency Checking], page 48) works by installing such hooks.

3.2.2.11 Statistics for Memory Allocation with `malloc`

You can get information about dynamic memory allocation by calling the `mallinfo` function. This function and its associated data type are declared in '`malloc.h`'; they are an extension of the standard SVID/XPG version.

struct mallinfo Data Type

This structure type is used to return information about the dynamic memory allocator. It contains the following members:

`int arena`
: This is the total size of memory allocated with `sbrk` by `malloc`, in bytes.

`int ordblks`
: This is the number of chunks not in use. (The memory allocator internally gets chunks of memory from the operating system, and then carves them up to satisfy individual `malloc` requests; see Section 3.2.2.6 [Efficiency Considerations for `malloc`], page 46.)

`int smblks`
: This field is unused.

`int hblks`
: This is the total number of chunks allocated with `mmap`.

`int hblkhd`
: This is the total size of memory allocated with `mmap`, in bytes.

`int usmblks`
: This field is unused.

`int fsmblks`
: This field is unused.

`int uordblks`
: This is the total size of memory occupied by chunks handed out by `malloc`.

`int fordblks`
: This is the total size of memory occupied by free (not in use) chunks.

`int keepcost`
: This is the size of the topmost releasable chunk that normally borders the end of the heap (i.e., the high end of the virtual address space's data segment).

`struct mallinfo` **mallinfo** (`void`) Function
This function returns information about the current dynamic memory usage in a structure of type `struct mallinfo`.

3.2.2.12 Summary of `malloc`-Related Functions

Here is a summary of the functions that work with `malloc`:

`void *malloc (size_t` *size*`)`
: Allocate a block of *size* bytes (see Section 3.2.2.1 [Basic Memory Allocation], page 42).

`void free (void *`*addr*`)`
: Free a block previously allocated by `malloc` (see Section 3.2.2.3 [Freeing Memory Allocated with `malloc`], page 44).

`void *realloc (void *`*addr*`, size_t` *size*`)`
: Make a block previously allocated by `malloc` larger or smaller, possibly by copying it to a new location (see Section 3.2.2.4 [Changing the Size of a Block], page 45).

`void *calloc (size_t` *count*`, size_t` *eltsize*`)`
: Allocate a block of *count* * *eltsize* bytes using `malloc`, and set its contents to zero (see Section 3.2.2.5 [Allocating Cleared Space], page 46).

`void *valloc (size_t size)`
: Allocate a block of *size* bytes, starting on a page boundary (see Section 3.2.2.7 [Allocating Aligned Memory Blocks], page 47).

`void *memalign (size_t size, size_t boundary)`
: Allocate a block of *size* bytes, starting on an address that is a multiple of *boundary* (see Section 3.2.2.7 [Allocating Aligned Memory Blocks], page 47).

`int mallopt (int param, int value)`
: Adjust a tunable parameter (see Section 3.2.2.8 [`malloc` Tunable Parameters], page 47).

`int mcheck (void (*abortfn) (void))`
: Tell `malloc` to perform occasional consistency checks on dynamically allocated memory, and to call *abortfn* when an inconsistency is found (see Section 3.2.2.9 [Heap Consistency Checking], page 48).

`void *(*__malloc_hook) (size_t size, const void *caller)`
: This is a pointer to a function that `malloc` uses whenever it is called.

`void *(*__realloc_hook) (void *ptr, size_t size, const void *caller)`
: This is a pointer to a function that `realloc` uses whenever it is called.

`void (*__free_hook) (void *ptr, const void *caller)`
: This is a pointer to a function that `free` uses whenever it is called.

`void (*__memalign_hook) (size_t size, size_t alignment, const void *caller)`
: This is a pointer to a function that `memalign` uses whenever it is called.

`struct mallinfo mallinfo (void)`
: Return information about the current dynamic memory usage (see Section 3.2.2.11 [Statistics for Memory Allocation with `malloc`], page 53).

3.2.3 Allocation Debugging

A complicated task when programming with languages that do not use garbage-collected dynamic memory allocation is to find memory leaks. Long running programs must assure that dynamically allocated objects are freed at the end of their lifetime. If this does not happen, the system runs out of memory, sooner or later.

The `malloc` implementation in the GNU C Library provides some simple means to detect such leaks and obtain some information on their location. To do this the application must be started in a special mode, which is enabled by an environment variable. There are no speed penalties for the program if the debugging mode is not enabled.

3.2.3.1 How to Install the Tracing Functionality

`void` **mtrace** (`void`) Function

When the `mtrace` function is called, it looks for an environment variable named `MALLOC_TRACE`. This variable is supposed to contain a valid file name. The user must have write access. If the file already exists, it is truncated. If the environment variable is not set or it does not name a valid file that can be opened for writing, nothing is done. The behavior of, for example, `malloc`, is not changed. For obvious reasons, this also happens if the application is installed with the SUID or SGID bit set.

If the named file is successfully opened, `mtrace` installs special handlers for the functions `malloc`, `realloc` and `free` (see Section 3.2.2.10 [Memory Allocation Hooks], page 50). From then on, all uses of these functions are traced and protocolled into the file. There is now of course a speed penalty for all calls to the traced functions, so tracing should not be enabled during normal use.

This function is a GNU extension and generally not available on other systems. The prototype can be found in '`mcheck.h`'.

`void` **muntrace** (`void`) Function

The `muntrace` function can be called after `mtrace` was used to enable tracing the `malloc` calls. If no (successful) call of `mtrace` was made, `muntrace` does nothing.

Otherwise, it deinstalls the handlers for `malloc`, `realloc` and `free`, then closes the protocol file. No calls are protocolled anymore and the program runs again at full speed.

This function is a GNU extension and generally not available on other systems. The prototype can be found in '`mcheck.h`'.

3.2.3.2 Example Program Excerpts

Even though the tracing functionality does not influence the run-time behavior of the program, it is not a good idea to call `mtrace` in all programs. Just imagine that you debug a program using `mtrace` and all other programs used in the debugging session also trace their `malloc` calls. The output file would be the same for all programs and thus unusable. Therefore, one should call `mtrace` only if compiled for debugging. A program could therefore start like this:

```
#include <mcheck.h>

int
main (int argc, char *argv[])
{
#ifdef DEBUGGING
  mtrace ();
#endif
```

```
  ...
}
```

This is all that is needed if you want to trace the calls during the whole runtime of the program. Alternatively, you can stop the tracing at any time with a call to `muntrace`. It is even possible to restart the tracing again with a new call to `mtrace`. But this can cause unreliable results, since there may be calls of the functions that are not called. Please note that not only the application uses the traced functions—libraries (including the C library itself) also use these functions.

This last point is also why it is not a good idea to call `muntrace` before the program terminates. The libraries are informed about the termination of the program only after the program returns from `main` or calls `exit` and so cannot free the memory they use before this time.

So the best thing you can do is to call `mtrace` as the very first function in the program and never call `muntrace`. So the program traces almost all uses of the `malloc` functions (except those calls that are executed by constructors of the program or used libraries).

3.2.3.3 Some More or Less Clever Ideas

You know the situation. The program is prepared for debugging and in all debugging sessions it runs well. But once it is started without debugging the error shows up. A typical example is a memory leak that becomes visible only when we turn off the debugging. If you foresee such situations you can still win. Simply use something equivalent to the following little program:

```
#include <mcheck.h>
#include <signal.h>

static void
enable (int sig)
{
  mtrace ();
  signal (SIGUSR1, enable);
}

static void
disable (int sig)
{
  muntrace ();
  signal (SIGUSR2, disable);
}

int
main (int argc, char *argv[])
{
```

```
  ...

  signal (SIGUSR1, enable);
  signal (SIGUSR2, disable);

  ...
}
```

The user can start the memory debugger any time she wants if the program was started with `MALLOC_TRACE` set in the environment. The output will of course not show the allocations that happened before the first signal, but if there is a memory leak, this will nevertheless show up.

3.2.3.4 Interpreting the Traces

If you take a look at the output it will look similar to this:

```
= Start
 [0x8048209] - 0x8064cc8
 [0x8048209] - 0x8064ce0
 [0x8048209] - 0x8064cf8
 [0x80481eb] + 0x8064c48 0x14
 [0x80481eb] + 0x8064c60 0x14
 [0x80481eb] + 0x8064c78 0x14
 [0x80481eb] + 0x8064c90 0x14
= End
```

What this all means is not really important since the trace file is not meant to be read by a human. Therefore no attention is given to readability. Instead there is a program that comes with the GNU C library that interprets the traces and outputs a summary in a user-friendly way. The program is called `mtrace` (it is in fact a Perl script) and it takes one or two arguments. In any case, the name of the file with the trace output must be specified. If an optional argument precedes the name of the trace file, this must be the name of the program that generated the trace.

```
drepper$ mtrace tst-mtrace log
No memory leaks.
```

In this case, the program `tst-mtrace` was run and it produced a trace file '`log`'. The message printed by `mtrace` shows there are no problems with the code—all allocated memory was freed afterwards.

If we call `mtrace` on the example trace given above, we would get a different output:

```
drepper$ mtrace errlog
- 0x08064cc8 Free 2 was never alloc'd 0x8048209
- 0x08064ce0 Free 3 was never alloc'd 0x8048209
- 0x08064cf8 Free 4 was never alloc'd 0x8048209

Memory not freed:
```

```
-----------------
   Address     Size     Caller
0x08064c48     0x14  at 0x80481eb
0x08064c60     0x14  at 0x80481eb
0x08064c78     0x14  at 0x80481eb
0x08064c90     0x14  at 0x80481eb
```

We have called `mtrace` with only one argument, so the script has no chance to find out what is meant with the addresses given in the trace. We can do better:

```
drepper$ mtrace tst errlog
- 0x08064cc8 Free 2 was never alloc'd /home/drepper/tst.c:39
- 0x08064ce0 Free 3 was never alloc'd /home/drepper/tst.c:39
- 0x08064cf8 Free 4 was never alloc'd /home/drepper/tst.c:39

Memory not freed:
-----------------
   Address     Size     Caller
0x08064c48     0x14  at /home/drepper/tst.c:33
0x08064c60     0x14  at /home/drepper/tst.c:33
0x08064c78     0x14  at /home/drepper/tst.c:33
0x08064c90     0x14  at /home/drepper/tst.c:33
```

Suddenly the output makes much more sense and the user can see immediately where the function calls causing the trouble can be found.

Interpreting this output is not complicated. There are at most two different situations being detected. First, `free` was called for pointers that were never returned by one of the allocation functions. This is usually a very bad problem and what this looks like is shown in the first three lines of the output. Situations like this are quite rare and if they appear show up drastically—the program normally crashes.

The other situation, which is much harder to detect, is memory leaks. As you can see in the output, the `mtrace` function collects all this information and so can say that the program calls an allocation function from line 33 in the source file '`/home/drepper/tst-mtrace.c`' four times without freeing this memory before the program terminates. Whether this is a real problem remains to be investigated.

3.2.4 Obstacks

An *obstack* is a pool of memory containing a stack of objects. You can create any number of separate obstacks, and then allocate objects in specified obstacks. Within each obstack, the last object allocated must always be the first one freed, but distinct obstacks are independent of each other.

Aside from this one constraint on the order of freeing, obstacks are totally general—an obstack can contain any number of objects of any size. They are implemented with macros, so allocation is usually very fast as long as the objects are

usually small. And the only space overhead per object is the padding needed to start each object on a suitable boundary.

3.2.4.1 Creating Obstacks

The utilities for manipulating obstacks are declared in the header file ‘`obstack.h`’.

struct obstack Data Type

An obstack is represented by a data structure of type `struct obstack`. This structure has a small fixed size; it records the status of the obstack and how to find the space in which objects are allocated. It does not contain any of the objects themselves. You should not try to access the contents of the structure directly; use only the functions described in this chapter.

You can declare variables of type `struct obstack` and use them as obstacks, or you can allocate obstacks dynamically like any other kind of object. Dynamic allocation of obstacks allows your program to have a variable number of different stacks. (You can even allocate an obstack structure in another obstack, but this is rarely useful.)

All the functions that work with obstacks require you to specify which obstack to use. You do this with a pointer of type `struct obstack *`. In the following, we often say "an obstack" when, strictly speaking, the object at hand is such a pointer.

The objects in the obstack are packed into large blocks called *chunks*. The `struct obstack` structure points to a chain of the chunks currently in use.

The obstack library obtains a new chunk whenever you allocate an object that won't fit in the previous chunk. Since the obstack library manages chunks automatically, you don't need to pay much attention to them, but you do need to supply a function that the obstack library should use to get a chunk. Usually you supply a function that uses `malloc` directly or indirectly. You must also supply a function to free a chunk. These matters are described in the following section.

3.2.4.2 Preparing for Using Obstacks

Each source file in which you plan to use the obstack functions must include the header file ‘`obstack.h`’, like this:

```
#include <obstack.h>
```

Also, if the source file uses the macro `obstack_init`, it must declare or define two functions or macros that will be called by the obstack library. One, `obstack_chunk_alloc`, is used to allocate the chunks of memory into which objects are packed. The other, `obstack_chunk_free`, is used to return chunks when the objects in them are freed. These macros should appear before any use of obstacks in the source file.

Usually these are defined to use `malloc` via the intermediary `xmalloc` (see Section 3.2.2 [Unconstrained Allocation], page 42). This is done with the following pair of macro definitions:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

Though the memory you get using obstacks really comes from `malloc`, using obstacks is faster because `malloc` is called less often, for larger blocks of memory (see Section 3.2.4.10 [Obstack Chunks], page 69, for full details).

At run time, before the program can use a `struct obstack` object as an obstack, it must initialize the obstack by calling `obstack_init`.

`int` **obstack_init** (`struct obstack *`*obstack-ptr*) Function

Initialize obstack *obstack-ptr* for allocation of objects. This function calls the obstack's `obstack_chunk_alloc` function. If allocation of memory fails, the function pointed to by `obstack_alloc_failed_handler` is called. The `obstack_init` function always returns 1. **Compatibility Note:** Former versions of obstack returned 0 if allocation failed.)

Here are two examples of how to allocate the space for an obstack and initialize it; first, an obstack that is a static variable:

```
static struct obstack myobstack;
...
obstack_init (&myobstack);
```

second, an obstack that is itself dynamically allocated:

```
struct obstack *myobstack_ptr
  = (struct obstack *) xmalloc (sizeof (struct obstack));

obstack_init (myobstack_ptr);
```

obstack_alloc_failed_handler Variable

The value of this variable is a pointer to a function that `obstack` uses when `obstack_chunk_alloc` fails to allocate memory. The default action is to print a message and abort. You should supply a function that either calls `exit` (see Section 14.6 [Program Termination], page 425) or `longjmp`[5] and doesn't return.

```
void my_obstack_alloc_failed (void)
...
obstack_alloc_failed_handler = &my_obstack_alloc_failed;
```

3.2.4.3 Allocation in an Obstack

The most direct way to allocate an object in an obstack is with `obstack_alloc`, which is invoked almost like `malloc`.

[5] Ibid., "Nonlocal Exits".

`void *` **obstack_alloc** (`struct obstack *`*obstack-ptr*`, int` *size*) Function

This allocates an uninitialized block of *size* bytes in an obstack and returns its address. Here *obstack-ptr* specifies which obstack to allocate the block in; it is the address of the `struct obstack` object that represents the obstack. Each obstack function or macro requires you to specify an *obstack-ptr* as the first argument.

This function calls the obstack's `obstack_chunk_alloc` function if it needs to allocate a new chunk of memory; it calls `obstack_alloc_failed_handler` if allocation of memory by `obstack_chunk_alloc` failed.

For example, here is a function that allocates a copy of a string *str* in a specific obstack, which is in the variable `string_obstack`:

```
struct obstack string_obstack;

char *
copystring (char *string)
{
  size_t len = strlen (string) + 1;
  char *s = (char *) obstack_alloc (&string_obstack, len);
  memcpy (s, string, len);
  return s;
}
```

To allocate a block with specified contents, use the function `obstack_copy`, declared like this:

`void *` **obstack_copy** (`struct obstack *`*obstack-ptr*`, void *`*address*`, int` *size*) Function

This allocates a block and initializes it by copying *size* bytes of data starting at *address*. It calls `obstack_alloc_failed_handler` if allocation of memory by `obstack_chunk_alloc` failed.

`void *` **obstack_copy0** (`struct obstack *`*obstack-ptr*`, void *`*address*`, int` *size*) Function

Like `obstack_copy`, but appends an extra byte containing a null character. This extra byte is not counted in the argument *size*.

The `obstack_copy0` function is convenient for copying a sequence of characters into an obstack as a null-terminated string. Here is an example of its use:

```
char *
obstack_savestring (char *addr, int size)
{
  return obstack_copy0 (&myobstack, addr, size);
}
```

Contrast this with the previous example of `savestring` using `malloc` (see Section 3.2.2.1 [Basic Memory Allocation], page 42).

3.2.4.4 Freeing Objects in an Obstack

To free an object allocated in an obstack, use the function `obstack_free`. Since the obstack is a stack of objects, freeing one object automatically frees all other objects allocated more recently in the same obstack.

`void` **obstack_free** (`struct obstack` **obstack-ptr*, `void` **object*) Function

If *object* is a null pointer, everything allocated in the obstack is freed. Otherwise, *object* must be the address of an object allocated in the obstack. Then *object* is freed, along with everything allocated in *obstack* since *object*.

Note that if *object* is a null pointer, the result is an uninitialized obstack. To free all memory in an obstack but leave it valid for further allocation, call `obstack_free` with the address of the first object allocated on the obstack:

```
obstack_free (obstack_ptr, first_object_allocated_ptr);
```

Recall that the objects in an obstack are grouped into chunks. When all the objects in a chunk become free, the obstack library automatically frees the chunk (see Section 3.2.4.2 [Preparing for Using Obstacks], page 60). Then other obstacks, or non-obstack allocation, can reuse the space of the chunk.

3.2.4.5 Obstack Functions and Macros

The interfaces for using obstacks may be defined either as functions or as macros, depending on the compiler. The obstack facility works with all C compilers, including both ISO C and traditional C, but there are precautions you must take if you plan to use compilers other than GNU C.

If you are using an old-fashioned non-ISO C compiler, all the obstack "functions" are actually defined only as macros. You can call these macros like functions, but you cannot use them in any other way (for example, you cannot take their address).

Calling the macros requires a special precaution—the first operand (the obstack pointer) may not contain any side effects, because it may be computed more than once. For example, if you write this:

```
obstack_alloc (get_obstack (), 4);
```

you will find that `get_obstack` may be called several times. If you use `*obstack_list_ptr++` as the obstack pointer argument, you will get very strange results, since the incrementation may occur several times.

In ISO C, each function has both a macro definition and a function definition. The function definition is used if you take the address of the function without calling it. An ordinary call uses the macro definition by default, but you can request the

function definition instead by writing the function name in parentheses, as shown here:

```
char *x;
void *(*funcp) ();
/* Use the macro. */
x = (char *) obstack_alloc (obptr, size);
/* Call the function. */
x = (char *) (obstack_alloc) (obptr, size);
/* Take the address of the function. */
funcp = obstack_alloc;
```

This is the same situation that exists in ISO C for the standard library functions (see Section 1.3.2 [Macro Definitions of Functions], page 5).

Warning: When you do use the macros, you must observe the precaution of avoiding side effects in the first operand, even in ISO C.

If you use the GNU C Compiler, this precaution is not necessary, because various language extensions in GNU C permit defining the macros so as to compute each argument only once.

3.2.4.6 Growing Objects

Because memory in obstack chunks is used sequentially, it is possible to build up an object step-by-step, adding one or more bytes at a time to the end of the object. With this technique, you do not need to know how much data you will put in the object until you come to the end of it. We call this the technique of *growing objects*. The special functions for adding data to the growing object are described in this section.

You don't need to do anything special when you start to grow an object. Using one of the functions to add data to the object automatically starts it. However, it is necessary to say explicitly when the object is finished. This is done with the function `obstack_finish`.

The actual address of the object thus built up is not known until the object is finished. Until then, it always remains possible that you will add so much data that the object must be copied into a new chunk.

While the obstack is in use for a growing object, you cannot use it for ordinary allocation of another object. If you try to do so, the space already added to the growing object will become part of the other object.

`void` **obstack_blank** (`struct obstack *`*obstack-ptr*`, int` *size*) Function

The most basic function for adding to a growing object is `obstack_blank`, which adds space without initializing it.

`void` **obstack_grow** (`struct obstack *`*obstack-ptr*`, void *`*data*`, int` *size*) Function

To add a block of initialized space, use `obstack_grow`, which is the growing-object analogue of `obstack_copy`. It adds *size* bytes of data to the growing object, copying the contents from *data*.

`void` **obstack_grow0** (`struct obstack *`*obstack-ptr*`, void *`*data*`, int` *size*) Function

This is the growing-object analogue of `obstack_copy0`. It adds *size* bytes copied from *data*, followed by an additional null character.

`void` **obstack_1grow** (`struct obstack *`*obstack-ptr*`, char` *c*) Function

To add one character at a time, use the function `obstack_1grow`. It adds a single byte containing *c* to the growing object.

`void` **obstack_ptr_grow** (`struct obstack *`*obstack-ptr*`, void *`*data*) Function

Adding the value of a pointer one can use the function `obstack_ptr_grow`. It adds `sizeof (void *)` bytes containing the value of *data*.

`void` **obstack_int_grow** (`struct obstack *`*obstack-ptr*`, int` *data*) Function

A single value of type `int` can be added by using the `obstack_int_grow` function. It adds `sizeof (int)` bytes to the growing object and initializes them with the value of *data*.

`void *` **obstack_finish** (`struct obstack *`*obstack-ptr*) Function

When you are finished growing the object, use the function `obstack_finish` to close it off and return its final address.

Once you have finished the object, the obstack is available for ordinary allocation or for growing another object.

This function can return a null pointer under the same conditions as `obstack_alloc` (see Section 3.2.4.3 [Allocation in an Obstack], page 61).

When you build an object by growing it, you will probably need to know afterward how long it became. You need not keep track of this as you grow the object, because you can find out the length from the obstack just before finishing the object with the function `obstack_object_size`, declared as follows:

`int` **obstack_object_size** (`struct obstack *`*obstack-ptr*) Function

This function returns the current size of the growing object, in bytes. Remember to call this function *before* finishing the object. After it is finished, `obstack_object_size` will return zero.

If you have started growing an object and wish to cancel it, you should finish it and then free it, like this:

```
obstack_free (obstack_ptr, obstack_finish (obstack_ptr));
```

This has no effect if no object was growing.

You can use `obstack_blank` with a negative size argument to make the current object smaller. Just don't try to shrink it beyond zero length—there's no telling what will happen if you do that.

3.2.4.7 Extra-Fast Growing Objects

The usual functions for growing objects incur overhead for checking whether there is room for the new growth in the current chunk. If you are frequently constructing objects in small steps of growth, this overhead can be significant.

You can reduce the overhead by using special "fast growth" functions that grow the object without checking. In order to have a robust program, you must do the checking yourself. If you do this checking in the simplest way each time you are about to add data to the object, you have not saved anything, because that is what the ordinary growth functions do. But if you can arrange to check less often, or check more efficiently, then you make the program faster.

The function `obstack_room` returns the amount of room available in the current chunk. It is declared as follows:

`int` **obstack_room** (`struct obstack *`*obstack-ptr*) Function

This returns the number of bytes that can be added safely to the current growing object (or to an object about to be started) in obstack *obstack* using the fast-growth functions.

While you know there is room, you can use these fast-growth functions for adding data to a growing object:

`void` **obstack_1grow_fast** (`struct obstack *`*obstack-ptr*`, char` *c*) Function

The function `obstack_1grow_fast` adds one byte containing the character *c* to the growing object in obstack *obstack-ptr*.

`void` **obstack_ptr_grow_fast** (`struct obstack *`*obstack-ptr*`, void *`*data*) Function

The function `obstack_ptr_grow_fast` adds `sizeof (void *)` bytes containing the value of *data* to the growing object in obstack *obstack-ptr*.

`void` **obstack_int_grow_fast** (`struct obstack *`*obstack-ptr*`, int` *data*) Function

The function `obstack_int_grow_fast` adds `sizeof (int)` bytes containing the value of *data* to the growing object in obstack *obstack-ptr*.

`void` **obstack_blank_fast** (`struct obstack *`*obstack-ptr*, `int` *size*) Function

The function `obstack_blank_fast` adds *size* bytes to the growing object in obstack *obstack-ptr* without initializing them.

When you check for space using `obstack_room` and there is not enough room for what you want to add, the fast-growth functions are not safe. In this case, simply use the corresponding ordinary growth function instead. Very soon this will copy the object to a new chunk; then there will be lots of room available again.

So, each time you use an ordinary growth function, check afterward for sufficient space using `obstack_room`. Once the object is copied to a new chunk, there will be plenty of space again, so the program will start using the fast-growth functions again.

Here is an example:

```
void
add_string (struct obstack *obstack, const char *ptr, int len)
{
  while (len > 0)
    {
      int room = obstack_room (obstack);
      if (room == 0)
        {
          /* Not enough room. Add one character slowly,
             which may copy to a new chunk and make room.  */
          obstack_1grow (obstack, *ptr++);
          len--;
        }
      else
        {
          if (room > len)
            room = len;
          /* Add fast as much as we have room for.  */
          len -= room;
          while (room-- > 0)
            obstack_1grow_fast (obstack, *ptr++);
        }
    }
}
```

3.2.4.8 Status of an Obstack

Here are functions that provide information on the current status of allocation in an obstack. You can use them to learn about an object while still growing it.

`void *` **obstack_base** (`struct obstack *`*obstack-ptr*) Function
This function returns the tentative address of the beginning of the currently growing object in *obstack-ptr*. If you finish the object immediately, it will have that address. If you make it larger first, it may outgrow the current chunk—then its address will change!

If no object is growing, this value says where the next object you allocate will start (once again assuming it fits in the current chunk).

`void *` **obstack_next_free** (`struct obstack *`*obstack-ptr*) Function
This function returns the address of the first free byte in the current chunk of obstack *obstack-ptr*. This is the end of the currently growing object. If no object is growing, `obstack_next_free` returns the same value as `obstack_base`.

`int` **obstack_object_size** (`struct obstack *`*obstack-ptr*) Function
This function returns the size in bytes of the currently growing object. This is equivalent to:

```
obstack_next_free (obstack-ptr) - obstack_base (obstack-ptr)
```

3.2.4.9 Alignment of Data in Obstacks

Each obstack has an *alignment boundary*; each object allocated in the obstack automatically starts on an address that is a multiple of the specified boundary. By default, this boundary is 4 bytes.

To access an obstack's alignment boundary, use the macro `obstack_alignment_mask`, whose function prototype looks like this:

`int` **obstack_alignment_mask** (`struct obstack *`*obstack-ptr*) Macro
The value is a bit mask; a bit that is 1 indicates that the corresponding bit in the address of an object should be 0. The mask value should be one less than a power of 2; the effect is that all object addresses are multiples of that power of 2. The default value of the mask is 3, so that addresses are multiples of 4. A mask value of 0 means an object can start on any multiple of 1 (that is, no alignment is required).

The expansion of the macro `obstack_alignment_mask` is an lvalue, so you can alter the mask by assignment. For example, this statement:

```
obstack_alignment_mask (obstack_ptr) = 0;
```

has the effect of turning off alignment processing in the specified obstack.

Note that a change in alignment mask does not take effect until *after* the next time an object is allocated or finished in the obstack. If you are not growing an object, you can make the new alignment mask take effect immediately by calling `obstack_finish`. This will finish a zero-length object and then do proper alignment for the next object.

3.2.4.10 Obstack Chunks

Obstacks work by allocating space for themselves in large chunks, and then parceling out space in the chunks to satisfy your requests. Chunks are normally 4096 bytes long unless you specify a different chunk size. The chunk size includes 8 bytes of overhead that are not actually used for storing objects. Regardless of the specified size, longer chunks will be allocated when necessary for long objects.

The obstack library allocates chunks by calling the function `obstack_chunk_alloc`, which you must define. When a chunk is no longer needed because you have freed all the objects in it, the obstack library frees the chunk by calling `obstack_chunk_free`, which you must also define.

These two must be defined (as macros) or declared (as functions) in each source file that uses `obstack_init` (see Section 3.2.4.1 [Creating Obstacks], page 60). Most often they are defined as macros like this:

```
#define obstack_chunk_alloc malloc
#define obstack_chunk_free free
```

Note that these are simple macros (no arguments). Macro definitions with arguments will not work! It is necessary that `obstack_chunk_alloc` or `obstack_chunk_free`, alone, expand into a function name if it is not itself a function name.

If you allocate chunks with `malloc`, the chunk size should be a power of 2. The default chunk size, 4096, was chosen because it is long enough to satisfy many typical requests on the obstack yet short enough not to waste too much memory in the portion of the last chunk not yet used.

`int` **obstack_chunk_size** (`struct obstack *`*obstack-ptr*) Macro

This returns the chunk size of the given obstack.

Since this macro expands to an lvalue, you can specify a new chunk size by assigning it a new value. Doing so does not affect the chunks already allocated, but will change the size of chunks allocated for that particular obstack in the future. It is unlikely to be useful to make the chunk size smaller, but making it larger might improve efficiency if you are allocating many objects whose size is comparable to the chunk size. Here is how to do so cleanly:

```
if (obstack_chunk_size (obstack_ptr) < new-chunk-size)
  obstack_chunk_size (obstack_ptr) = new-chunk-size;
```

3.2.4.11 Summary of Obstack Functions

Here is a summary of all the functions associated with obstacks. Each takes the address of an obstack (`struct obstack *`) as its first argument.

`void obstack_init (struct obstack *`*obstack-ptr*`)`

Initialize use of an obstack (see Section 3.2.4.1 [Creating Obstacks], page 60).

`void *obstack_alloc (struct obstack *`*obstack-ptr*`, int` *size*`)`
: Allocate an object of *size* uninitialized bytes (see Section 3.2.4.3 [Allocation in an Obstack], page 61).

`void *obstack_copy (struct obstack *`*obstack-ptr*`, void *`*address*`, int` *size*`)`
: Allocate an object of *size* bytes, with contents copied from *address* (see Section 3.2.4.3 [Allocation in an Obstack], page 61).

`void *obstack_copy0 (struct obstack *`*obstack-ptr*`, void *`*address*`, int` *size*`)`
: Allocate an object of *size*+1 bytes, with *size* of them copied from *address*, followed by a null character at the end (see Section 3.2.4.3 [Allocation in an Obstack], page 61).

`void obstack_free (struct obstack *`*obstack-ptr*`, void *`*object*`)`
: Free *object* and everything allocated in the specified obstack more recently than *object* (see Section 3.2.4.4 [Freeing Objects in an Obstack], page 63).

`void obstack_blank (struct obstack *`*obstack-ptr*`, int` *size*`)`
: Add *size* uninitialized bytes to a growing object (see Section 3.2.4.6 [Growing Objects], page 64).

`void obstack_grow (struct obstack *`*obstack-ptr*`, void *`*address*`, int` *size*`)`
: Add *size* bytes, copied from *address*, to a growing object (see Section 3.2.4.6 [Growing Objects], page 64).

`void obstack_grow0 (struct obstack *`*obstack-ptr*`, void *`*address*`, int` *size*`)`
: Add *size* bytes, copied from *address*, to a growing object, and then add another byte containing a null character (see Section 3.2.4.6 [Growing Objects], page 64).

`void obstack_1grow (struct obstack *`*obstack-ptr*`, char` *data-char*`)`
: Add one byte containing *data-char* to a growing object (see Section 3.2.4.6 [Growing Objects], page 64).

`void *obstack_finish (struct obstack *`*obstack-ptr*`)`
: Finalize the object that is growing and return its permanent address (see Section 3.2.4.6 [Growing Objects], page 64).

`int obstack_object_size (struct obstack *`*obstack-ptr*`)`
: Get the current size of the currently growing object (see Section 3.2.4.6 [Growing Objects], page 64).

`void obstack_blank_fast (struct obstack *`*obstack-ptr*`, int` *size*`)`
: Add *size* uninitialized bytes to a growing object without checking that there is enough room (see Section 3.2.4.7 [Extra-Fast Growing Objects], page 66).

`void obstack_1grow_fast (struct obstack *`*obstack-ptr*`, char` *data-char*`)`
: Add one byte containing *data-char* to a growing object without checking that there is enough room (see Section 3.2.4.7 [Extra-Fast Growing Objects], page 66).

`int obstack_room (struct obstack *`*obstack-ptr*`)`
: Get the amount of room now available for growing the current object (see Section 3.2.4.7 [Extra-Fast Growing Objects], page 66).

`int obstack_alignment_mask (struct obstack *`*obstack-ptr*`)`
: The mask used for aligning the beginning of an object. This is an lvalue (see Section 3.2.4.9 [Alignment of Data in Obstacks], page 68).

`int obstack_chunk_size (struct obstack *`*obstack-ptr*`)`
: The size for allocating chunks. This is an lvalue (see Section 3.2.4.10 [Obstack Chunks], page 69).

`void *obstack_base (struct obstack *`*obstack-ptr*`)`
: Tentative starting address of the currently growing object (see Section 3.2.4.8 [Status of an Obstack], page 67).

`void *obstack_next_free (struct obstack *`*obstack-ptr*`)`
: Address just after the end of the currently growing object (see Section 3.2.4.8 [Status of an Obstack], page 67).

3.2.5 Automatic Storage with Variable Size

The function `alloca` supports a kind of half-dynamic allocation in which blocks are allocated dynamically but freed automatically.

Allocating a block with `alloca` is an explicit action; you can allocate as many blocks as you wish, and compute the size at run time. But all the blocks are freed when you exit the function that `alloca` was called from, just as if they were automatic variables declared in that function. There is no way to free the space explicitly.

The prototype for `alloca` is in '`stdlib.h`'. This function is a BSD extension.

`void *` **alloca** `(size_t` *size*`);` Function
: The return value of `alloca` is the address of a block of *size* bytes of memory, allocated in the stack frame of the calling function.

Do not use `alloca` inside the arguments of a function call—you will get unpredictable results, because the stack space for the `alloca` would appear on the stack in the middle of the space for the function arguments. An example of what to avoid is `foo (x, alloca (4), y)`.

3.2.5.1 `alloca` Example

As an example of the use of `alloca`, here is a function that opens a file name made from concatenating two argument strings, and returns a file descriptor or minus one, signifying failure:

```
int
open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  stpcpy (stpcpy (name, str1), str2);
  return open (name, flags, mode);
}
```

Here is how you would get the same results with `malloc` and `free`:

```
int
open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) malloc (strlen (str1) + strlen (str2) + 1);
  int desc;
  if (name == 0)
    fatal ("virtual memory exceeded");
  stpcpy (stpcpy (name, str1), str2);
  desc = open (name, flags, mode);
  free (name);
  return desc;
}
```

As you can see, it is simpler with `alloca`. But `alloca` has other, more important advantages, and some disadvantages.

3.2.5.2 Advantages of `alloca`

Here are the reasons why `alloca` may be preferable to `malloc`:

- Using `alloca` wastes very little space and is very fast. (It is open coded by the GNU C Compiler.)
- Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size. `alloca` does not cause memory fragmentation.
- Nonlocal exits done with `longjmp`[6] automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

  To illustrate this, suppose you have a function `open_or_report_error` that returns a descriptor, like `open`, if it succeeds, but does not return to its caller if it fails. If the file cannot be opened, it prints an error message and

[6] Ibid., “Nonlocal Exits”.

jumps out to the command level of your program using `longjmp`. Let's change `open2` (see Section 3.2.5.1 [`alloca` Example], page 72) to use this subroutine:

```
int
open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  stpcpy (stpcpy (name, str1), str2);
  return open_or_report_error (name, flags, mode);
}
```

Because of the way `alloca` works, the memory it allocates is freed even when an error occurs, with no special effort required.

By contrast, the previous definition of `open2` (which uses `malloc` and `free`) would develop a memory leak if it were changed in this way. Even if you are willing to make more changes to fix it, there is no easy way to do so.

3.2.5.3 Disadvantages of `alloca`

These are the disadvantages of `alloca` in comparison with `malloc`:

- If you try to allocate more memory than the machine can provide, you don't get a clean error message. Instead you get a fatal signal like the one you would get from an infinite recursion; probably a segmentation violation.[7]
- Some non-GNU systems fail to support `alloca`, so it is less portable. However, a slower emulation of `alloca` written in C is available for use on systems with this deficiency.

3.2.5.4 GNU C Variable-Size Arrays

In GNU C, you can replace most uses of `alloca` with an array of variable size. Here is how `open2` would look then:

```
int open2 (char *str1, char *str2, int flags, int mode)
{
  char name[strlen (str1) + strlen (str2) + 1];
  stpcpy (stpcpy (name, str1), str2);
  return open (name, flags, mode);
}
```

But `alloca` is not always equivalent to a variable-size array, for several reasons:

- A variable-size array's space is freed at the end of the scope of the name of the array. The space allocated with `alloca` remains until the end of the function.
- It is possible to use `alloca` within a loop, allocating an additional block on each iteration. This is impossible with variable-size arrays.

[7] Ibid., "Program-Error Signals".

Note: If you mix use of `alloca` and variable-size arrays within one function, exiting a scope in which a variable-size array was declared frees all blocks allocated with `alloca` during the execution of that scope.

3.3 Resizing the Data Segment

The symbols in this section are declared in '`unistd.h`'.

You will not normally use the functions in this section, because the functions described in Section 3.2 [Allocating Storage for Program Data], page 41 are easier to use. Those are interfaces to a GNU C Library memory allocator that uses the functions below itself. The functions below are simple interfaces to system calls.

`int` **brk** (`void *`*addr*) Function

`brk` sets the high end of the calling process' data segment to *addr*.

The address of the end of a segment is defined to be the address of the last byte in the segment plus 1.

The function has no effect if *addr* is lower than the low end of the data segment. (This is considered success.)

The function fails if it would cause the data segment to overlap another segment or exceed the process' data storage limit.[8]

The function is named for a common historical case where data storage and the stack are in the same segment. Data storage allocation grows upward from the bottom of the segment while the stack grows downward toward it from the top of the segment and the curtain between them is called the *break*.

The return value is zero on success. On failure, the return value is `-1` and `errno` is set accordingly. The following `errno` values are specific to this function:

`ENOMEM` The request would cause the data segment to overlap another segment or exceed the process' data storage limit.

`int` **sbrk** (`ptrdiff_t` *delta*) Function

This function is the same as `brk` except that you specify the new end of the data segment as an offset *delta* from the current end and on success the return value is the address of the resulting end of the data segment instead of zero.

This means you can use '`sbrk(0)`' to find out what the current end of the data segment is.

3.4 Locking Pages

You can tell the system to associate a particular virtual memory page with a real page frame and keep it that way—i.e., cause the page to be paged in if it isn't

[8] Ibid., "Limiting Resource Usage".

already and mark it so it will never be paged out and consequently will never cause a page fault. This is called *locking* a page.

The functions in this chapter lock and unlock the calling process' pages.

3.4.1 Why Lock Pages?

Because page faults cause paged out pages to be paged in transparently, a process rarely needs to be concerned about locking pages. However, there are two reasons people sometimes are:

- Speed. A page fault is transparent only insofar as the process is not sensitive to how long it takes to do a simple memory access. Time-critical processes, especially real-time processes, may not be able to wait or may not be able to tolerate variance in execution speed.

  A process that needs to lock pages for this reason probably also needs priority among other processes for use of the CPU.[9]

  In some cases, the programmer knows better than the system's demand paging allocator which pages should remain in real memory to optimize system performance. In this case, locking pages can help.
- Privacy. If you keep secrets in virtual memory and that virtual memory gets paged out, that increases the chance that the secrets will get out. If a password gets written out to disk swap space, for example, it might still be there long after virtual and real memory have been wiped clean.

Be aware that when you lock a page, that is one fewer page frame that can be used to back other virtual memory (by the same or other processes), which can mean more page faults, which means the system runs more slowly. In fact, if you lock enough memory, some programs may not be able to run at all for lack of real memory.

3.4.2 Locked-Memory Details

A memory lock is associated with a virtual page, not a real frame. The paging rule is: If a frame backs at least one locked page, don't page it out.

Memory locks do not stack—you can't lock a particular page twice so that it has to be unlocked twice before it is truly unlocked. It is either locked or it isn't.

A memory lock persists until the process that owns the memory explicitly unlocks it. (But process termination and exec cause the virtual memory to cease to exist, which you might say means it isn't locked any more.)

Memory locks are not inherited by child processes (but note that on a modern Unix system, immediately after a fork, the parent's and the child's virtual address space are backed by the same real page frames, so the child enjoys the parent's locks).[10]

---

[9] Ibid., "Process CPU Priority and Scheduling".

[10] Ibid., "Creating a Process".

Because of its ability to impact other processes, only the superuser can lock a page. Any process can unlock its own page.

The system sets limits on the amount of memory a process can have locked and the amount of real memory it can have dedicated to it.[11]

In Linux, locked pages aren't as locked as you might think. Two virtual pages that are not shared memory can nonetheless be backed by the same real frame. The kernel does this in the name of efficiency when it knows both virtual pages contain identical data, and does it even if one or both of the virtual pages are locked.

But when a process modifies one of those pages, the kernel must get it a separate frame and fill it with the page's data. This is known as a *copy-on-write page fault*. It takes a small amount of time and in a pathological case, getting that frame may require I/O.

To make sure this doesn't happen to your program, don't just lock the pages. Write to them as well, unless you know you won't ever write to them ever. And to make sure you have preallocated frames for your stack, enter a scope that declares a C automatic variable larger than the maximum stack size you will need, set it to something, then return from its scope.

3.4.3 Functions to Lock and Unlock Pages

The symbols in this section are declared in '`sys/mman.h`'. These functions are defined by POSIX.1b, but their availability depends on your kernel. If your kernel doesn't allow these functions, they exist but always fail. They *are* available with a Linux kernel.

Portability Note: POSIX.1b requires that when the `mlock` and `munlock` functions are available, the file '`unistd.h`' define the macro `_POSIX_MEMLOCK_RANGE` and the file `limits.h` define the macro `PAGESIZE` to be the size of a memory page in bytes. It requires that when the `mlockall` and `munlockall` functions are available, the '`unistd.h`' file define the macro `_POSIX_MEMLOCK`. The GNU C Library conforms to this requirement.

`int` **mlock** (`const void *`*addr*`, size_t` *len*) Function

`mlock` locks a range of the calling process' virtual pages.

The range of memory starts at address *addr* and is *len* bytes long. Actually, since you must lock whole pages, it is the range of pages that include any part of the specified range.

When the function returns successfully, each of those pages is backed by (connected to) a real frame (is resident) and is marked to stay that way. This means the function may cause page-ins and have to wait for them.

When the function fails, it does not affect the lock status of any pages.

The return value is zero if the function succeeds. Otherwise, it is `-1` and `errno` is set accordingly. `errno` values specific to this function are

[11] Ibid., "Limiting Resource Usage".

`ENOMEM`

- At least some of the specified address range does not exist in the calling process' virtual-address space.
- The locking would cause the process to exceed its locked page limit.

`EPERM` The calling process is not superuser.

`EINVAL` *len* is not positive.

`ENOSYS` The kernel does not provide `mlock` capability.

You can lock *all* a process' memory with `mlockall`. You unlock memory with `munlock` or `munlockall`.

To avoid all page faults in a C program, you have to use `mlockall`, because some of the memory a program uses is hidden from the C code, e.g., the stack and automatic variables, and you wouldn't know what address to tell `mlock`.

`int` **munlock** (`const void *`*addr*`, size_t` *len*) Function

`mlock` unlocks a range of the calling process' virtual pages.

`munlock` is the inverse of `mlock` and functions completely analogously to `mlock`, except that there is no `EPERM` failure.

`int` **mlockall** (`int` *flags*) Function

`mlockall` locks all the pages in a process' virtual-memory address space, and/or any that are added to it in the future. This includes the pages of the code, data and stack segment, as well as shared libraries, user-space kernel data, shared memory, and memory-mapped files.

flags is a string of single-bit flags represented by the following macros. They tell `mlockall` which of its functions you want. All other bits must be zero.

`MCL_CURRENT`
Lock all pages that currently exist in the calling process' virtual-address space.

`MCL_FUTURE`
Set a mode such that any pages added to the process' virtual-address space in the future will be locked from birth. This mode does not affect future address spaces owned by the same process, so exec, which replaces a process' address space, wipes out `MCL_FUTURE`.[12]

When the function returns successfully, and you specified `MCL_CURRENT`, all of the process' pages are backed by (connected to) real frames (they are resident) and are marked to stay that way. This means the function may cause page-ins and have to wait for them.

[12] Ibid., "Executing a File".

When the process is in `MCL_FUTURE` mode because it successfully executed this function and specified `MCL_CURRENT`, any system call by the process that requires space be added to its virtual address space fails with `errno` = `ENOMEM` if locking the additional space would cause the process to exceed its locked page limit. In the case that the address space addition that can't be accommodated is stack expansion, the stack expansion fails and the kernel sends a `SIGSEGV` signal to the process.

When the function fails, it does not affect the lock status of any pages or the future locking mode.

The return value is zero if the function succeeds. Otherwise, it is `-1` and `errno` is set accordingly. `errno` values specific to this function are

`ENOMEM`

- At least some of the specified address range does not exist in the calling process' virtual-address space.
- The locking would cause the process to exceed its locked page limit.

`EPERM` The calling process is not superuser.

`EINVAL` Undefined bits in *flags* are not zero.

`ENOSYS` The kernel does not provide `mlockall` capability.

You can lock just specific pages with `mlock`. You unlock pages with `munlockall` and `munlock`.

`int` **munlockall** (`void`) Function

`munlockall` unlocks every page in the calling process' virtual-address space and turn off `MCL_FUTURE` future locking mode.

The return value is zero if the function succeeds. Otherwise, it is `-1` and `errno` is set accordingly. The only way this function can fail is for generic reasons that all functions and system calls can fail, so there are no specific `errno` values.

4 Character Handling

Programs that work with characters and strings often need to classify a character as alphabetic, digit, white space, etc., and perform case conversion operations on characters. The functions in the header file ‘ctype.h’ are provided for this purpose.

Since the choice of locale and character set can alter the classifications of particular character codes, all of these functions are affected by the current locale. (More precisely, they are affected by the locale currently selected for character classification—the LC_CTYPE category; see Section 7.3 [Categories of Activities That Locales Affect], page 182.)

The ISO C standard specifies two different sets of functions. The one set works on char type characters, the other one on wchar_t wide characters (see Section 6.1 [Introduction to Extended Characters], page 133).

4.1 Classification of Characters

This section explains the library functions for classifying characters. For example, isalpha is the function to test for an alphabetic character. It takes one argument, the character to test, and returns a nonzero integer if the character is alphabetic, and zero otherwise. You would use it like this:

```
if (isalpha (c))
  printf ("The character '%c' is alphabetic.\n", c);
```

Each of the functions in this section tests for membership in a particular class of characters; each has a name starting with ‘is’. Each of them takes one argument, which is a character to test, and returns an int which is treated as a Boolean value. The character argument is passed as an int, and it may be the constant value EOF instead of a real character.

The attributes of any given character can vary between locales. See Chapter 7 [Locales and Internationalization], page 181, for more information on locales.

These functions are declared in the header file ‘ctype.h’.

int **islower** (int *c*) Function

Returns true if *c* is a lowercase letter. The letter need not be from the Latin alphabet—any representable alphabet is valid.

int **isupper** (int *c*) Function

Returns true if *c* is an uppercase letter. The letter need not be from the Latin alphabet—any representable alphabet is valid.

int **isalpha** (int *c*) Function

Returns true if *c* is an alphabetic character (a letter). If islower or isupper is true of a character, then isalpha is also true.

In some locales, there may be additional characters for which `isalpha` is true—letters that are neither uppercase nor lowercase. But in the standard `"C"` locale, there are no such additional characters.

`int` **isdigit** (`int` *c*) Function

Returns true if *c* is a decimal digit (‘`0`’ through ‘`9`’).

`int` **isalnum** (`int` *c*) Function

Returns true if *c* is an alphanumeric character (a letter or number); in other words, if either `isalpha` or `isdigit` is true of a character, then `isalnum` is also true.

`int` **isxdigit** (`int` *c*) Function

Returns true if *c* is a hexadecimal digit. Hexadecimal digits include the normal decimal digits ‘`0`’ through ‘`9`’ and the letters ‘`A`’ through ‘`F`’ and ‘`a`’ through ‘`f`’.

`int` **ispunct** (`int` *c*) Function

Returns true if *c* is a punctuation character. This means any printing character that is not alphanumeric or a space character.

`int` **isspace** (`int` *c*) Function

Returns true if *c* is a *white-space* character. In the standard `"C"` locale, `isspace` returns true for only the standard white-space characters:

`' '`	space
`'\f'`	formfeed
`'\n'`	newline
`'\r'`	carriage return
`'\t'`	horizontal tab
`'\v'`	vertical tab

`int` **isblank** (`int` *c*) Function

Returns true if *c* is a blank character; that is, a space or a tab. This function is a GNU extension.

`int` **isgraph** (`int` *c*) Function

Returns true if *c* is a graphic character; that is, a character that has a glyph associated with it. The white-space characters are not considered graphic.

`int` **isprint** (`int` *c*) Function

Returns true if *c* is a printing character. Printing characters include all the graphic characters, plus the space (' ') character.

`int` **iscntrl** (`int` *c*) Function

Returns true if *c* is a control character; that is, a character that is not a printing character.

`int` **isascii** (`int` *c*) Function

Returns true if *c* is a 7-bit `unsigned char` value that fits into the US/UK ASCII character set. This function is a BSD extension and is also an SVID extension.

4.2 Case Conversion

This section explains the library functions for performing conversions such as case mappings on characters. For example, `toupper` converts any character to upper case if possible. If the character can't be converted, `toupper` returns it unchanged.

These functions take one argument of type `int`, which is the character to convert, and return the converted character as an `int`. If the conversion is not applicable to the argument given, the argument is returned unchanged.

Compatibility Note: In pre-ISO C dialects, instead of returning the argument unchanged, these functions may fail when the argument is not suitable for the conversion. Thus for portability, you may need to write `islower(c) ? toupper(c) : c` rather than just `toupper(c)`.

These functions are declared in the header file '`ctype.h`'.

`int` **tolower** (`int` *c*) Function

If *c* is an uppercase letter, `tolower` returns the corresponding lowercase letter. If *c* is not an uppercase letter, *c* is returned unchanged.

`int` **toupper** (`int` *c*) Function

If *c* is a lowercase letter, `toupper` returns the corresponding uppercase letter. Otherwise *c* is returned unchanged.

`int` **toascii** (`int` *c*) Function
This function converts *c* to a 7-bit `unsigned char` value that fits into the US/UK ASCII character set, by clearing the high-order bits. This function is a BSD extension and is also an SVID extension.

`int` **_tolower** (`int` *c*) Function
This is identical to `tolower`, and is provided for compatibility with the SVID (see Section 1.2.4 [SVID (The System V Interface Description)], page 3).

`int` **_toupper** (`int` *c*) Function
This is identical to `toupper`, and is provided for compatibility with the SVID.

4.3 Character Class Determination for Wide Characters

Amendment 1 to ISO C90 defines functions to classify wide characters. Although the original ISO C90 standard already defined the type `wchar_t`, no functions operating on them were defined.

The general design of the classification functions for wide characters is more general. It allows extensions to the set of available classifications, beyond those that are always available. The POSIX standard specifies how extensions can be made, and this is already implemented in the GNU C Library implementation of the `localedef` program.

The character class functions are normally implemented with bitsets, with a bitset per character. For a given character, the appropriate bitset is read from a table and a test is performed as to whether a certain bit is set. Which bit is tested for is determined by the class.

For the wide-character classification functions this is made visible. There is a type classification type defined, a function to retrieve this value for a given class, and a function to test whether a given character is in this class, using the classification value. On top of this the normal character classification functions as used for `char` objects can be defined.

wctype_t Data type
The `wctype_t` can hold a value which represents a character class. The only defined way to generate such a value is by using the `wctype` function.
This type is defined in ‘`wctype.h`’.

`wctype_t` **wctype** (`const char *`*property*) Function
The `wctype` returns a value representing a class of wide characters that is identified by the string *property*. Besides some standard properties, each locale can define its own ones. In case no property with the given name is known for the current locale selected for the `LC_CTYPE` category, the function returns zero.
The properties known in every locale are
`"alnum"` `"alpha"` `"cntrl"` `"digit"`

```
"graph"          "lower"        "print"        "punct"
"space"          "upper"        "xdigit"
```

This function is declared in ‘wctype.h’.

To test the membership of a character to one of the nonstandard classes, the ISO C standard defines a completely new function.

`int` **iswctype** (`wint_t` *wc*, `wctype_t` *desc*) Function

This function returns a nonzero value if *wc* is in the character class specified by *desc*. *desc* must previously be returned by a successful call to `wctype`.

This function is declared in ‘wctype.h’.

To make it easier to use the commonly used classification functions, they are defined in the C library. There is no need to use `wctype` if the property string is one of the known character classes. In some situations it is desirable to construct the property strings, and then it is important that `wctype` can also handle the standard classes.

`int` **iswalnum** (`wint_t` *wc*) Function

This function returns a nonzero value if *wc* is an alphanumeric character (a letter or number). In other words, if either `iswalpha` or `iswdigit` is true of a character, then `iswalnum` is also true.

This function can be implemented using:

```
iswctype (wc, wctype ("alnum"))
```

It is declared in ‘wctype.h’.

`int` **iswalpha** (`wint_t` *wc*) Function

Returns true if *wc* is an alphabetic character (a letter). If `iswlower` or `iswupper` is true of a character, then `iswalpha` is also true.

In some locales, there may be additional characters for which `iswalpha` is true—letters that are neither uppercase nor lowercase. But in the standard `"C"` locale, there are no such additional characters.

This function can be implemented using:

```
iswctype (wc, wctype ("alpha"))
```

It is declared in ‘wctype.h’.

`int` **iswcntrl** (`wint_t` *wc*) Function

Returns true if *wc* is a control character; that is, a character that is not a printing character.

This function can be implemented using:

```
iswctype (wc, wctype ("cntrl"))
```

It is declared in ‘wctype.h’.

`int` **iswdigit** (`wint_t` *wc*) Function

Returns true if *wc* is a digit (e.g., '`0`' through '`9`'). Please note that this function does not only return a nonzero value for *decimal* digits, but for all kinds of digits. A consequence is that code like the following will *not* work unconditionally for wide characters:

```
n = 0;
while (iswdigit (*wc))
  {
    n *= 10;
    n += *wc++ - L'0';
  }
```

This function can be implemented using:

```
iswctype (wc, wctype ("digit"))
```

It is declared in '`wctype.h`'.

`int` **iswgraph** (`wint_t` *wc*) Function

Returns true if *wc* is a graphic character; that is, a character that has a glyph associated with it. The white-space characters are not considered graphic.

This function can be implemented using:

```
iswctype (wc, wctype ("graph"))
```

It is declared in '`wctype.h`'.

`int` **iswlower** (`wint_t` *wc*) Function

Returns true if *wc* is a lowercase letter. The letter need not be from the Latin alphabet—any representable alphabet is valid.

This function can be implemented using:

```
iswctype (wc, wctype ("lower"))
```

It is declared in '`wctype.h`'.

`int` **iswprint** (`wint_t` *wc*) Function

Returns true if *wc* is a printing character. Printing characters include all the graphic characters, plus the space (' ') character.

This function can be implemented using:

```
iswctype (wc, wctype ("print"))
```

It is declared in '`wctype.h`'.

`int` **iswpunct** (`wint_t` *wc*) Function

Returns true if *wc* is a punctuation character. This means any printing character that is not alphanumeric or a space character.

This function can be implemented using:

```
iswctype (wc, wctype ("punct"))
```

It is declared in ‘wctype.h’.

`int` **iswspace** (`wint_t` *wc*) Function

Returns true if *wc* is a *white-space* character. In the standard `"C"` locale, `iswspace` returns true for only the standard white-space characters:

`L' '`	space
`L'\f'`	formfeed
`L'\n'`	newline
`L'\r'`	carriage return
`L'\t'`	horizontal tab
`L'\v'`	vertical tab

This function can be implemented using:

```
iswctype (wc, wctype ("space"))
```

It is declared in ‘wctype.h’.

`int` **iswupper** (`wint_t` *wc*) Function

Returns true if *wc* is an uppercase letter. The letter need not be from the Latin alphabet—any representable alphabet is valid.

This function can be implemented using:

```
iswctype (wc, wctype ("upper"))
```

It is declared in ‘wctype.h’.

`int` **iswxdigit** (`wint_t` *wc*) Function

Returns true if *wc* is a hexadecimal digit. Hexadecimal digits include the normal decimal digits ‘0’ through ‘9’ and the letters ‘A’ through ‘F’ and ‘a’ through ‘f’.

This function can be implemented using:

```
iswctype (wc, wctype ("xdigit"))
```

It is declared in ‘wctype.h’.

The GNU C Library also provides a function that is not defined in the ISO C standard but that is available as a version for single-byte characters as well.

`int` **iswblank** (`wint_t` *wc*) Function

Returns true if *wc* is a blank character; that is, a space or a tab. This function is a GNU extension. It is declared in ‘wchar.h’.

4.4 Notes on Using the Wide-Character Classes

The first note is probably not astonishing but still occasionally a cause of problems. The `isw`*XXX* functions can be implemented using macros and in fact, the GNU C Library does this. They are still available as real functions, but when the '`wctype.h`' header is included the macros will be used. This is the same as the `char` type versions of these functions.

The second note covers something new. It can be best illustrated by a (real-world) example. The first piece of code is an excerpt from the original code. It is truncated a bit but the intention should be clear:

```
int
is_in_class (int c, const char *class)
{
  if (strcmp (class, "alnum") == 0)
    return isalnum (c);
  if (strcmp (class, "alpha") == 0)
    return isalpha (c);
  if (strcmp (class, "cntrl") == 0)
    return iscntrl (c);
  ...
  return 0;
}
```

Now, with the `wctype` and `iswctype` you can avoid the `if` cascades, but rewriting the code as follows is wrong:

```
int
is_in_class (int c, const char *class)
{
  wctype_t desc = wctype (class);
  return desc ? iswctype ((wint_t) c, desc) : 0;
}
```

The problem is that there is no guarantee that the wide-character representation of a single-byte character can be found using casting. In fact, usually this fails miserably. The correct solution to this problem is to write the code as follows:

```
int
is_in_class (int c, const char *class)
{
  wctype_t desc = wctype (class);
  return desc ? iswctype (btowc (c), desc) : 0;
}
```

See Section 6.3.3 [Converting Single Characters], page 140, for more information on `btowc`. Note that this change probably does not improve the performance of the program a lot since the `wctype` function still has to make the string comparisons. It gets really interesting if the `is_in_class` function is called more than once for the same class name. In this case, the variable *desc* could be com-

puted once and reused for all the calls. Therefore, the above form of the function is probably not the final one.

4.5 Mapping of Wide Characters

The classification functions are also generalized by the ISO C standard. Instead of just allowing the two standard mappings, a locale can contain others. Again, the `localedef` program already supports generating such locale data files.

wctrans_t Data Type

This data type is defined as a scalar type that can hold a value representing the locale-dependent character mapping. There is no way to construct such a value apart from using the return value of the `wctrans` function.

This type is defined in '`wctype.h`'.

`wctrans_t` **wctrans** (`const char *`*property*) Function

The `wctrans` function has to be used to find out whether a named mapping is defined in the current locale selected for the `LC_CTYPE` category. If the return value is nonzero, you can use it afterwards in calls to `towctrans`. If the return value is zero, no such mapping is known in the current locale.

Besides locale-specific mappings, there are two mappings that are guaranteed to be available in every locale: `"tolower"` and `"toupper"`.

These functions are declared in '`wctype.h`'.

`wint_t` **towctrans** (`wint_t` *wc*, `wctrans_t` *desc*) Function

`towctrans` maps the input character *wc* according to the rules of the mapping for which *desc* is a descriptor, and returns the value it finds. *desc* must be obtained by a successful call to `wctrans`.

This function is declared in '`wctype.h`'.

For the generally available mappings, the ISO C standard defines convenient shortcuts so that it is not necessary to call `wctrans` for them.

`wint_t` **towlower** (`wint_t` *wc*) Function

If *wc* is an uppercase letter, `towlower` returns the corresponding lowercase letter. If *wc* is not an uppercase letter, *wc* is returned unchanged.

`towlower` can be implemented using:

```
towctrans (wc, wctrans ("tolower"))
```

This function is declared in '`wctype.h`'.

`wint_t` **towupper** (`wint_t` *wc*) Function

If *wc* is a lowercase letter, `towupper` returns the corresponding uppercase letter. Otherwise *wc* is returned unchanged.

`towupper` can be implemented using:

```
towctrans (wc, wctrans ("toupper"))
```

This function is declared in ‘wctype.h’.

The same warnings given in the last section for the use of the wide-character classification functions apply here. It is not possible to simply cast a char type value to a wint_t and use it as an argument to towctrans calls.

5 String and Array Utilities

Operations on strings (or arrays of characters) are an important part of many programs. The GNU C Library provides an extensive set of string utility functions, including functions for copying, concatenating, comparing and searching strings. Many of these functions can also operate on arbitrary regions of storage; for example, the `memcpy` function can be used to copy the contents of any kind of array.

It's fairly common for beginning C programmers to "reinvent the wheel" by duplicating this functionality in their own code, but it pays to become familiar with the library functions and to make use of them, since this offers benefits in maintenance, efficiency, and portability.

For instance, you could easily compare one string to another in two lines of C code, but if you use the built-in `strcmp` function, you're less likely to make a mistake. And, since these library functions are typically highly optimized, your program may run faster too.

5.1 Representation of Strings

This section is a quick summary of string concepts for beginning C programmers. It describes how character strings are represented in C and some common pitfalls. If you are already familiar with this material, you can skip this section.

A *string* is an array of `char` objects. But string-valued variables are usually declared to be pointers of type `char *`. Such variables do not include space for the text of a string; that has to be stored somewhere else—in an array variable, a string constant, or dynamically allocated memory (see Section 3.2 [Allocating Storage for Program Data], page 41). It's up to you to store the address of the chosen memory space into the pointer variable. Alternatively, you can store a *null pointer* in the pointer variable. The null pointer does not point anywhere, so attempting to reference the string it points to gets an error.

string normally refers to multibyte-character strings as opposed to wide-character strings. Wide-character strings are arrays of type `wchar_t` and, as for multibyte-character strings, usually pointers of type `wchar_t *` are used.

By convention, a *null character*, `'\0'`, marks the end of a multibyte-character string and the *null wide character*, `L'\0'`, marks the end of a wide-character string. For example, in testing to see whether the `char *` variable *p* points to a null character marking the end of a string, you can write `!*p` or `*p == '\0'`.

A null character is quite different conceptually from a null pointer, although both are represented by the integer `0`.

String literals appear in C program source as strings of characters between double-quote characters (`"`) where the initial double-quote character is immediately preceded by a capital `L` (ell) character (as in `L"foo"`). In ISO C, string literals can also be formed by *string concatenation*: `"a" "b"` is the same as `"ab"`. For wide-character strings one can either use `L"a" L"b"` or `L"a" "b"`. Modification of string literals is not allowed by the GNU C Compiler, because literals are placed in read-only storage.

Character arrays that are declared `const` cannot be modified either. It's generally good style to declare nonmodifiable string pointers to be of type `const char *`, since this often allows the C compiler to detect accidental modifications as well as providing some amount of documentation about what your program intends to do with the string.

The amount of memory allocated for the character array may extend past the null character that normally marks the end of the string. In this document, the term *allocated size* is always used to refer to the total amount of memory allocated for the string, while the term *length* refers to the number of characters up to (but not including) the terminating null character.

A notorious source of program bugs is trying to put more characters in a string than fit in its allocated size. When writing code that extends strings or moves characters into a preallocated array, you should be very careful to keep track of the length of the text and make explicit checks for overflowing the array. Many of the library functions *do not* do this for you! Remember also that you need to allocate an extra byte to hold the null character that marks the end of the string.

Originally strings were sequences of bytes where each byte represents a single character. This is still true today if the strings are encoded using a single-byte character encoding. Things are different if the strings are encoded using a multibyte encoding (for more information on encodings see Section 6.1 [Introduction to Extended Characters], page 133). There is no difference in the programming interface for these two kind of strings; the programmer has to be aware of this and interpret the byte sequences accordingly.

But since there is no separate interface taking care of these differences the byte-based string functions are sometimes hard to use. Since the count parameters of these functions specify bytes, a call to `strncpy` could cut a multibyte character in the middle and put an incomplete (and therefore unusable) byte-sequence in the target buffer.

To avoid these problems, later versions of the ISO C standard introduce a second set of functions that operate on *wide characters* (see Section 6.1 [Introduction to Extended Characters], page 133). These functions don't have the problems the single-byte versions have, since every wide character is a legal, interpretable value. This does not mean that cutting wide-character strings at arbitrary points is without problems. It normally is for alphabet-based languages (except for nonnormalized text) but languages based on syllables still have the problem that more than one wide character is necessary to complete a logical unit. This is a higher-level problem that the C library functions are not designed to solve. But it is at least good that no invalid byte-sequences can be created. Also, the higher-level functions can also operate more easily on wide characters than on multibyte characters, so that in general, it is advisable to use wide characters internally whenever text is more than simply copied.

The remainder of this chapter will discuss the functions for handling wide-character strings in parallel with the discussion of the multibyte-character strings, since there is almost always an exact equivalent available.

5.2 String and Array Conventions

This chapter describes both functions that work on arbitrary arrays or blocks of memory, and functions that are specific to null-terminated arrays of characters and wide characters.

Functions that operate on arbitrary blocks of memory have names beginning with '`mem`' and '`wmem`' (such as `memcpy` and `wmemcpy`) and invariably take an argument that specifies the size (in bytes and wide characters respectively) of the block of memory to operate on. The array arguments and return values for these functions have type `void *` or `wchar_t`. As a matter of style, the elements of the arrays used with the '`mem`' functions are referred to as *bytes*. You can pass any kind of pointer to these functions, and the `sizeof` operator is useful in computing the value for the size argument. Parameters to the '`wmem`' functions must be of type `wchar_t *`. These functions are not really usable with anything but arrays of this type.

In contrast, functions that operate specifically on strings and wide character strings have names beginning with '`str`' and '`wcs`' respectively (such as `strcpy` and `wcscpy`) and look for a null character to terminate the string instead of requiring an explicit size argument to be passed. (Some of these functions accept a specified maximum length, but they also check for premature termination with a null character.) The array arguments and return values for these functions have type `char *` and `wchar_t *` respectively, and the array elements are referred to as *characters* and *wide characters*.

In many cases, there are both '`mem`' and '`str`'/'`wcs`' versions of a function. The one that is more appropriate to use depends on the exact situation. When your program is manipulating arbitrary arrays or blocks of storage, then you should always use the '`mem`' functions. On the other hand, when you are manipulating null-terminated strings it is usually more convenient to use the '`str`'/'`wcs`' functions, unless you already know the length of the string in advance. The '`wmem`' functions should be used for wide-character arrays with known size.

Some of the memory and string functions take single characters as arguments. Since a value of type `char` is automatically promoted into a value of type `int` when used as a parameter, the functions are declared with `int` as the type of the parameter in question. In case of the wide-character function, the situation is similar—the parameter type for a single wide character is `wint_t` and not `wchar_t`. This would, for many implementations, not be necessary since the `wchar_t` is large enough to not be automatically promoted, but since the ISO C standard does not require such a choice of types, the `wint_t` type is used.

5.3 String Length

You can get the length of a string using the `strlen` function. This function is declared in the header file '`string.h`'.

`size_t` **strlen** (`const char *`*s*) Function

The `strlen` function returns the length of the null-terminated string *s* in bytes. In other words, it returns the offset of the terminating null character within the array.

For example:

```
strlen ("hello, world")
    ⇒ 12
```

When applied to a character array, the `strlen` function returns the length of the string stored there, not its allocated size. You can get the allocated size of the character array that holds a string using the `sizeof` operator:

```
char string[32] = "hello, world";
sizeof (string)
    ⇒ 32
strlen (string)
    ⇒ 12
```

But beware, this will not work unless *string* is the character array itself, not a pointer to it. For example:

```
char string[32] = "hello, world";
char *ptr = string;
sizeof (string)
    ⇒ 32
sizeof (ptr)
    ⇒ 4   /* (on a machine with 4 byte pointers) */
```

This is an easy mistake to make when you are working with functions that take string arguments; those arguments are always pointers, not arrays.

It must also be noted that for multibyte-encoded strings the return value does not have to correspond to the number of characters in the string. To get this value, the string can be converted to wide characters and `wcslen` can be used, or something like the following code can be used:

```
/* The input is in string.
   The length is expected in n.  */
{
  mbstate_t t;
  char *scopy = string;
  /* In initial state.  */
  memset (&t, '\0', sizeof (t));
  /* Determine number of characters.  */
  n = mbsrtowcs (NULL, &scopy, strlen (scopy), &t);
}
```

This is cumbersome, so if the number of characters (as opposed to bytes) is needed, often it is better to work with wide characters.

The wide character equivalent is declared in '`wchar.h`'.

`size_t` **wcslen** (`const wchar_t *`*ws*) Function

The `wcslen` function is the wide character equivalent to `strlen`. The return value is the number of wide characters in the wide-character string pointed to by *ws* (this is also the offset of the terminating null wide character of *ws*).

Since there are no multi-wide-character sequences making up one character, the return value is not only the offset in the array, but also the number of wide characters.

This function was introduced in Amendment 1 to ISO C90.

`size_t` **strnlen** (`const char *`*s*`, size_t` *maxlen*) Function

The `strnlen` function returns the length of the string *s* in bytes if this length is smaller than *maxlen* bytes. Otherwise it returns *maxlen*. Therefore, this function is equivalent to `(strlen (`*s*`) < n ? strlen (`*s*`) :` *maxlen*`)`, but it is more efficient and works even if the string *s* is not null-terminated.

```
char string[32] = "hello, world";
strnlen (string, 32)
    ⇒ 12
strnlen (string, 5)
    ⇒ 5
```

This function is a GNU extension and is declared in '`string.h`'.

`size_t` **wcsnlen** (`const wchar_t *`*ws*`, size_t` *maxlen*) Function

`wcsnlen` is the wide-character equivalent to `strnlen`. The *maxlen* parameter specifies the maximum number of wide characters.

This function is a GNU extension and is declared in '`wchar.h`'.

5.4 Copying and Concatenation

You can use the functions described in this section to copy the contents of strings and arrays, or to append the contents of one string to another. The '`str`' and '`mem`' functions are declared in the header file '`string.h`' while the '`wstr`' and '`wmem`' functions are declared in the file '`wchar.h`'.

A helpful way to remember the ordering of the arguments to the functions in this section is that it corresponds to an assignment expression, with the destination array specified to the left of the source array. All of these functions return the address of the destination array.

Most of these functions do not work properly if the source and destination arrays overlap. For example, if the beginning of the destination array overlaps the end of the source array, the original contents of that part of the source array may get overwritten before it is copied. Even worse, in the case of the string functions, the null character marking the end of the string may be lost, and the copy function might get stuck in a loop, trashing all the memory allocated to your program.

All functions that have problems copying between overlapping arrays are explicitly identified in this manual. In addition to functions in this section, there are

a few others like `sprintf` (see Section 17.12.7 [Formatted Output Functions], page 470) and `scanf` (see Section 17.14.8 [Formatted Input Functions], page 495).

`void *` **memcpy** (`void *restrict` *to*, `const void *restrict` *from*, `size_t` *size*) Function

The `memcpy` function copies *size* bytes from the object beginning at *from* into the object beginning at *to*. The behavior of this function is undefined if the two arrays *to* and *from* overlap; use `memmove` instead if overlapping is possible.

The value returned by `memcpy` is the value of *to*.

Here is an example of how you might use `memcpy` to copy the contents of an array:

```
struct foo *oldarray, *newarray;
int arraysize;
...
memcpy (new, old, arraysize * sizeof (struct foo));
```

`wchar_t *` **wmemcpy** (`wchar_t *restrict` *wto*, `const wchar_t *restruct` *wfrom*, `size_t` *size*) Function

The `wmemcpy` function copies *size* wide characters from the object beginning at *wfrom* into the object beginning at *wto*. The behavior of this function is undefined if the two arrays *wto* and *wfrom* overlap; use `wmemmove` instead if overlapping is possible.

The following is one possible implementation of `wmemcpy`, but there are also other optimizations possible:

```
wchar_t *
wmemcpy (wchar_t *restrict wto, const wchar_t *restrict wfrom,
         size_t size)
{
  return (wchar_t *) memcpy (wto, wfrom, size * sizeof (wchar_t));
}
```

The value returned by `wmemcpy` is the value of *wto*.

This function was introduced in Amendment 1 to ISO C90.

`void *` **mempcpy** (`void *restrict` *to*, `const void *restrict` *from*, `size_t` *size*) Function

The `mempcpy` function is nearly identical to the `memcpy` function. It copies *size* bytes from the object beginning at `from` into the object pointed to by *to*. But instead of returning the value of *to*, it returns a pointer to the byte following the last written byte in the object beginning at *to*—the value is `((void *) ((char *)` *to* + *size*)).

This function is useful in situations where a number of objects will be copied to consecutive memory positions:

```
void *
combine (void *o1, size_t s1, void *o2, size_t s2)
```

```
{
  void *result = malloc (s1 + s2);
  if (result != NULL)
    mempcpy (mempcpy (result, o1, s1), o2, s2);
  return result;
}
```

This function is a GNU extension.

`wchar_t *` **wmempcpy** (`wchar_t *restrict` *wto*, `const wchar_t *restrict` *wfrom*, `size_t` *size*) Function

The `wmempcpy` function is nearly identical to the `wmemcpy` function. It copies *size* wide characters from the object beginning at `wfrom` into the object pointed to by *wto*. But instead of returning the value of *wto*, it returns a pointer to the wide character following the last written wide character in the object beginning at *wto*—the value is *wto* + *size*.

This function is useful in situations where a number of objects will be copied to consecutive memory positions.

The following is one possible implementation of `wmemcpy`, but there are also other optimizations possible:

```
wchar_t *
wmempcpy (wchar_t *restrict wto, const wchar_t *restrict wfrom,
          size_t size)
{
  return (wchar_t *) mempcpy (wto, wfrom, size * sizeof (wchar_t));
}
```

This function is a GNU extension.

`void *` **memmove** (`void *`*to*, `const void *`*from*, `size_t` *size*) Function

`memmove` copies the *size* bytes at *from* into the *size* bytes at *to*, even if those two blocks of space overlap. In the case of overlap, `memmove` is careful to copy the original values of the bytes in the block at *from*, including those bytes that also belong to the block at *to*.

The value returned by `memmove` is the value of *to*.

`wchar_t *` **wmemmove** (`wchar *`*wto*, `const wchar_t *`*wfrom*, `size_t` *size*) Function

`wmemmove` copies the *size* wide characters at *wfrom* into the *size* wide characters at *wto*, even if those two blocks of space overlap. In the case of overlap, `memmove` is careful to copy the original values of the wide characters in the block at *wfrom*, including those wide characters that also belong to the block at *wto*.

The following is one possible implementation of `wmemcpy`, but there are also other optimizations possible:

```
wchar_t *
wmempcpy (wchar_t *restrict wto, const wchar_t *restrict wfrom,
          size_t size)
{
  return (wchar_t *) mempcpy (wto, wfrom, size * sizeof (wchar_t));
}
```

The value returned by `wmemmove` is the value of *wto*.

This function is a GNU extension.

`void *` **memccpy** (`void *restrict` *to*, `const void *restrict` *from*, `int` *c*, `size_t` *size*) Function

This function copies no more than *size* bytes from *from* to *to*, stopping if a byte matching *c* is found. The return value is a pointer into *to* one byte past where *c* was copied, or a null pointer if no byte matching *c* appeared in the first *size* bytes of *from*.

`void *` **memset** (`void *`*block*, `int` *c*, `size_t` *size*) Function

This function copies the value of *c* (converted to an `unsigned char`) into each of the first *size* bytes of the object beginning at *block*. It returns the value of *block*.

`wchar_t *` **wmemset** (`wchar_t *`*block*, `wchar_t` *wc*, `size_t` *size*) Function

This function copies the value of *wc* into each of the first *size* wide characters of the object beginning at *block*. It returns the value of *block*.

`char *` **strcpy** (`char *restrict` *to*, `const char *restrict` *from*) Function

This copies characters from the string *from* (up to and including the terminating null character) into the string *to*. Like `memcpy`, this function has undefined results if the strings overlap. The return value is the value of *to*.

`wchar_t *` **wcscpy** (`wchar_t *restrict` *wto*, `const wchar_t *restrict` *wfrom*) Function

This copies wide characters from the string *wfrom* (up to and including the terminating null wide character) into the string *wto*. Like `wmemcpy`, this function has undefined results if the strings overlap. The return value is the value of *wto*.

`char *` **strncpy** (`char *restrict` *to*, `const char *restrict` *from*, `size_t` *size*) Function

This function is similar to `strcpy` but always copies exactly *size* characters into *to*.

If the length of *from* is more than *size*, then `strncpy` copies just the first *size* characters. Note that in this case there is no null terminator written into *to*.

If the length of *from* is less than *size*, then `strncpy` copies all of *from*, followed by enough null characters to add up to *size* characters in all. This behavior is rarely useful, but it is specified by the ISO C standard.

The behavior of `strncpy` is undefined if the strings overlap.

Using `strncpy` as opposed to `strcpy` is a way to avoid bugs relating to writing past the end of the allocated space for *to*. However, it can also make your program much slower in one common case: copying a string that is probably small into a potentially large buffer. In this case, *size* may be large, and when it is, `strncpy` will waste a considerable amount of time copying null characters.

`wchar_t *` **wcsncpy** (`wchar_t *restrict` *wto*`, const wchar_t *restrict` *wfrom*`, size_t` *size*) Function

This function is similar to `wcscpy` but always copies exactly *size* wide characters into *wto*.

If the length of *wfrom* is more than *size*, then `wcsncpy` copies just the first *size* wide characters. Note that in this case there is no null terminator written into *wto*.

If the length of *wfrom* is less than *size*, then `wcsncpy` copies all of *wfrom*, followed by enough null wide characters to add up to *size* wide characters in all. This behavior is rarely useful, but it is specified by the ISO C standard.

The behavior of `wcsncpy` is undefined if the strings overlap.

Using `wcsncpy` as opposed to `wcscpy` is a way to avoid bugs relating to writing past the end of the allocated space for *wto*. However, it can also make your program much slower in one common case: copying a string that is probably small into a potentially large buffer. In this case, *size* may be large, and when it is, `wcsncpy` will waste a considerable amount of time copying null wide characters.

`char *` **strdup** (`const char *`*s*) Function

This function copies the null-terminated string *s* into a newly allocated string. The string is allocated using `malloc` (see Section 3.2.2 [Unconstrained Allocation], page 42). If `malloc` cannot allocate space for the new string, `strdup` returns a null pointer. Otherwise, it returns a pointer to the new string.

`wchar_t *` **wcsdup** (`const wchar_t *`*ws*) Function

This function copies the null-terminated wide-character string *ws* into a newly allocated string. The string is allocated using `malloc` (see Section 3.2.2 [Unconstrained Allocation], page 42). If `malloc` cannot allocate space for the new string, `wcsdup` returns a null pointer. Otherwise, it returns a pointer to the new wide-character string.

This function is a GNU extension.

`char *` **strndup** (`const char *`*s*`, size_t` *size*) Function

This function is similar to `strdup` but always copies at most *size* characters into the newly allocated string.

If the length of *s* is more than *size*, then `strndup` copies just the first *size* characters and adds a closing null terminator. Otherwise, all characters are copied and the string is terminated.

This function is different to `strncpy` in that it always terminates the destination string.

`strndup` is a GNU extension.

`char *` **stpcpy** (`char *restrict` *to*, `const char *restrict` *from*) Function

This function is like `strcpy`, except that it returns a pointer to the end of the string *to* (that is, the address of the terminating null character `to + strlen (from)`) rather than the beginning.

For example, this program uses `stpcpy` to concatenate '`foo`' and '`bar`' to produce '`foobar`', which it then prints:

```
#include <string.h>
#include <stdio.h>

int
main (void)
{
  char buffer[10];
  char *to = buffer;
  to = stpcpy (to, "foo");
  to = stpcpy (to, "bar");
  puts (buffer);
  return 0;
}
```

This function is not part of the ISO or POSIX standards, and is not customary on Unix systems, but we did not invent it either. Perhaps it comes from MS-DOS.

Its behavior is undefined if the strings overlap. The function is declared in '`string.h`'.

`wchar_t *` **wcpcpy** (`wchar_t *restrict` *wto*, `const wchar_t *restrict` *wfrom*) Function

This function is like `wcscpy`, except that it returns a pointer to the end of the string *wto* (that is, the address of the terminating null character `wto + strlen (wfrom)`) rather than the beginning.

This function is not part of ISO or POSIX but was found useful while developing the GNU C Library itself.

The behavior of `wcpcpy` is undefined if the strings overlap.

`wcpcpy` is a GNU extension and is declared in '`wchar.h`'.

`char *` **stpncpy** (`char *restrict` *to*, `const char *restrict` *from*, `size_t` *size*) Function

This function is similar to `stpcpy` but copies always exactly *size* characters into *to*.

If the length of *from* is more then *size*, then `stpncpy` copies just the first *size* characters and returns a pointer to the character directly following the one that was copied last. In this case, there is no null terminator written into *to*.

If the length of *from* is less than *size*, then `stpncpy` copies all of *from*, followed by enough null characters to add up to *size* characters in all. This behavior is rarely useful, but it is implemented to be useful in contexts where this behavior of the `strncpy` is used. `stpncpy` returns a pointer to the *first* written null character.

This function is not part of ISO or POSIX but was found useful while developing the GNU C Library itself.

Its behavior is undefined if the strings overlap. The function is declared in '`string.h`'.

`wchar_t *` **wcpncpy** (`wchar_t *restrict` *wto*, `const wchar_t *restrict` *wfrom*, `size_t` *size*) Function

This function is similar to `wcpcpy` but copies always exactly *wsize* characters into *wto*.

If the length of *wfrom* is more then *size*, then `wcpncpy` copies just the first *size* wide characters and returns a pointer to the wide character directly following the last nonnull wide character which was copied last. Note that in this case there is no null terminator written into *wto*.

If the length of *wfrom* is less than *size*, then `wcpncpy` copies all of *wfrom*, followed by enough null characters to add up to *size* characters in all. This behavior is rarely useful, but it is implemented to be useful in contexts where this behavior of the `wcsncpy` is used. `wcpncpy` returns a pointer to the *first* written null character.

This function is not part of ISO or POSIX but was found useful while developing the GNU C Library itself.

Its behavior is undefined if the strings overlap.

`wcpncpy` is a GNU extension and is declared in '`wchar.h`'.

`char *` **strdupa** (`const char *`*s*) Macro

This macro is similar to `strdup` but allocates the new string using `alloca` instead of `malloc` (see Section 3.2.5 [Automatic Storage with Variable Size], page 71). This means that the returned string has the same limitations as any block of memory allocated using `alloca`.

For obvious reasons, `strdupa` is implemented only as a macro; you cannot get the address of this function. Despite this limitation, it is a useful function. The following code shows a situation where using `malloc` would be much more expensive:

```
#include <paths.h>
#include <string.h>
#include <stdio.h>

const char path[] = _PATH_STDPATH;

int
main (void)
{
  char *wr_path = strdupa (path);
  char *cp = strtok (wr_path, ":");

  while (cp != NULL)
    {
      puts (cp);
      cp = strtok (NULL, ":");
    }
  return 0;
}
```

`strtok` must be passed a writeable string. Passing a const string, such as *path* in the above example, would be invalid. Also, using `strdupa` directly would also cause conflict due to the use of `alloca` in `strdupa` (see Section 3.2.5 [Automatic Storage with Variable Size], page 71).

This function is only available if GNU CC is used.

`char *` **strndupa** (`const char *`*s*, `size_t` *size*) Macro

This function is similar to `strndup`, but like `strdupa` it allocates the new string using `alloca` (see Section 3.2.5 [Automatic Storage with Variable Size], page 71). The same advantages and limitations of `strdupa` are valid for `strndupa` too.

This function is implemented only as a macro, just like `strdupa`. Just as `strdupa`, this macro also must not be used inside the parameter list in a function call.

`strndupa` is only available if GNU CC is used.

`char *` **strcat** (`char *restrict` *to*, `const char *restrict` *from*) Function

The `strcat` function is similar to `strcpy`, except that the characters from *from* are concatenated or appended to the end of *to*, instead of overwriting it. The first character from *from* overwrites the null character marking the end of *to*.

An equivalent definition for `strcat` would be

```
char *
```

```
strcat (char *restrict to, const char *restrict from)
{
  strcpy (to + strlen (to), from);
  return to;
}
```

This function has undefined results if the strings overlap.

`wchar_t *` **wcscat** (`wchar_t *restrict` *wto*, `const wchar_t *restrict` *wfrom*) Function

The `wcscat` function is similar to `wcscpy`, except that the characters from *wfrom* are concatenated or appended to the end of *wto*, instead of overwriting it. The first character from *wfrom* overwrites the null character marking the end of *wto*.

An equivalent definition for `wcscat` would be

```
wchar_t *
wcscat (wchar_t *wto, const wchar_t *wfrom)
{
  wcscpy (wto + wcslen (wto), wfrom);
  return wto;
}
```

This function has undefined results if the strings overlap.

Programmers using the `strcat` or `wcscat` function (or the following `strncat` or `wcsncar` functions for that matter) can easily be recognized as lazy and reckless. In almost all situations, the lengths of the participating strings are known (since otherwise you cannot ensure that the allocated size of the buffer is sufficient). Or at least you could know them if you keep track of the results of the various function calls. But then it is very inefficient to use `strcat`/`wcscat`. Much time is wasted finding the end of the destination string so that the actual copying can start. This is a common example:

```
/* This function concatenates arbitrarily many strings.  The last
   parameter must be NULL.  */
char *
concat (const char *str, ...)
{
  va_list ap, ap2;
  size_t total = 1;
  const char *s;
  char *result;

  va_start (ap, str);
  /* Actually va_copy, but this is the name more gcc versions
     understand.  */
  __va_copy (ap2, ap);
```

```
  /* Determine how much space we need.  */
  for (s = str; s != NULL; s = va_arg (ap, const char *))
    total += strlen (s);

  va_end (ap);

  result = (char *) malloc (total);
  if (result != NULL)
    {
      result[0] = '\0';

      /* Copy the strings.  */
      for (s = str; s != NULL; s = va_arg (ap2, const char *))
        strcat (result, s);
    }

  va_end (ap2);

  return result;
}
```

This looks quite simple, especially the second loop, where the strings are actually copied. But these innocent lines hide a major performance penalty. Just imagine that ten strings of 100 bytes each have to be concatenated. For the second string, we search the already stored 100 bytes for the end of the string so that we can append the next string. For all strings in total, the comparisons necessary to find the end of the intermediate results add up to 5500! If we combine the copying with the search for the allocation, we can write this function more efficiently:

```
char *
concat (const char *str, ...)
{
  va_list ap;
  size_t allocated = 100;
  char *result = (char *) malloc (allocated);
  char *wp;

  if (allocated != NULL)
    {
      char *newp;

      va_start (ap, atr);

      wp = result;
      for (s = str; s != NULL; s = va_arg (ap, const char *))
```

```
          {
            size_t len = strlen (s);

            /* Resize the allocated memory if necessary.  */
            if (wp + len + 1 > result + allocated)
              {
                allocated = (allocated + len) * 2;
                newp = (char *) realloc (result, allocated);
                if (newp == NULL)
                  {
                    free (result);
                    return NULL;
                  }
                wp = newp + (wp - result);
                result = newp;
              }

            wp = mempcpy (wp, s, len);
          }

        /* Terminate the result string.  */
        *wp++ = '\0';

        /* Resize memory to the optimal size.  */
        newp = realloc (result, wp - result);
        if (newp != NULL)
          result = newp;

        va_end (ap);
      }

    return result;
  }
```

With a bit more knowledge about the input strings, one could fine-tune the memory allocation. The difference we are pointing to here is that we don't use `strcat` anymore. We always keep track of the length of the current intermediate result so we can save the search for the end of the string and use `mempcpy`. We also don't use `stpcpy`, which might seem more natural since we handle strings. But this is not necessary, since we already know the length of the string and therefore can use the faster memory copying function. The example would work for wide characters the same way.

Whenever a programmer feels the need to use `strcat`, she should think twice and look through the program to see whether the code cannot be rewritten to take

advantage of already calculated results. It is almost always unnecessary to use `strcat`.

`char *` **strncat** (`char *restrict` *to*, `const char *restrict` *from*, `size_t` *size*) Function

This function is like `strcat` except that not more than *size* characters from *from* are appended to the end of *to*. A single null character is also always appended to *to*, so the total allocated size of *to* must be at least *size* + `1` bytes longer than its initial length.

The `strncat` function could be implemented like this:

```
char *
strncat (char *to, const char *from, size_t size)
{
  to[strlen (to) + size] = '\0';
  strncpy (to + strlen (to), from, size);
  return to;
}
```

The behavior of `strncat` is undefined if the strings overlap.

`wchar_t *` **wcsncat** (`wchar_t *restrict` *wto*, `const wchar_t *restrict` *wfrom*, `size_t` *size*) Function

This function is like `wcscat` except that not more than *size* characters from *from* are appended to the end of *to*. A single null character is also always appended to *to*, so the total allocated size of *to* must be at least *size* + `1` bytes longer than its initial length.

The `wcsncat` function could be implemented like this:

```
wchar_t *
wcsncat (wchar_t *restrict wto, const wchar_t *restrict wfrom,
         size_t size)
{
  wto[wcslen (to) + size] = L'\0';
  wcsncpy (wto + wcslen (wto), wfrom, size);
  return wto;
}
```

The behavior of `wcsncat` is undefined if the strings overlap.

Here is an example showing the use of `strncpy` and `strncat` (the wide-character version is equivalent). Notice how, in the call to `strncat`, the *size* parameter is computed to avoid overflowing the character array `buffer`.

```
#include <string.h>
#include <stdio.h>
```

```
#define SIZE 10

static char buffer[SIZE];

main ()
{
  strncpy (buffer, "hello", SIZE);
  puts (buffer);
  strncat (buffer, ", world", SIZE - strlen (buffer) - 1);
  puts (buffer);
}
```

The output produced by this program looks like:

```
hello
hello, wo
```

`void` **bcopy** (`const void *`*from*`, void *`*to*`, size_t` *size*) Function

This is a partially obsolete alternative for `memmove`, derived from BSD. It is not quite equivalent to `memmove`, because the arguments are not in the same order and there is no return value.

`void` **bzero** (`void *`*block*`, size_t` *size*) Function

This is a partially obsolete alternative for `memset`, derived from BSD. It is not as general as `memset`, because the only value it can store is zero.

5.5 String/Array Comparison

You can use the functions in this section to perform comparisons on the contents of strings and arrays. As well as checking for equality, these functions can also be used as the ordering functions for sorting operations (see Chapter 12 [Searching and Sorting], page 343 for an example of this).

Unlike most comparison operations in C, the string comparison functions return a nonzero value if the strings are *not* equivalent rather than if they are. The sign of the value indicates the relative ordering of the first characters in the strings that are not equivalent: a negative value indicates that the first string is "less" than the second, while a positive value indicates that the first string is "greater".

The most common use of these functions is to check only for equality. This is canonically done with an expression like '`! strcmp (s1, s2)`'.

All of these functions are declared in the header file '`string.h`'.

`int` **memcmp** (`const void *`*a1*`, const void *`*a2*`, size_t` *size*) Function

The function `memcmp` compares the *size* bytes of memory beginning at *a1* against the *size* bytes of memory beginning at *a2*. The value returned has the

same sign as the difference between the first differing pair of bytes (interpreted as `unsigned char` objects, then promoted to `int`).

If the contents of the two blocks are equal, `memcmp` returns `0`.

`int` **wmemcmp** (`const wchar_t *`*a1*`, const wchar_t *`*a2*`, size_t` *size*) Function

The function `wmemcmp` compares the *size* wide characters beginning at *a1* against the *size* wide characters beginning at *a2*. The value returned is smaller than or larger than zero depending on whether the first differing wide character is *a1* is smaller or larger than the corresponding character in *a2*.

If the contents of the two blocks are equal, `wmemcmp` returns `0`.

On arbitrary arrays, the `memcmp` function is mostly useful for testing equality. It usually isn't meaningful to do byte-wise ordering comparisons on arrays of things other than bytes. For example, a byte-wise comparison on the bytes that make up floating-point numbers isn't likely to tell you anything about the relationship between the values of the floating-point numbers.

`wmemcmp` is really only useful to compare arrays of type `wchar_t`, since the function looks at `sizeof (wchar_t)` bytes at a time, and this number of bytes is system dependent.

You should also be careful about using `memcmp` to compare objects that can contain "holes", such as the padding inserted into structure objects to enforce alignment requirements, extra space at the end of unions and extra characters at the ends of strings whose length is less than their allocated size. The contents of these "holes" are indeterminate and may cause strange behavior when performing byte-wise comparisons. For more predictable results, perform an explicit component-wise comparison.

For example, given a structure-type definition like:

```
struct foo
  {
    unsigned char tag;
    union
      {
        double f;
        long i;
        char *p;
      } value;
  };
```

you are better off writing a specialized comparison function to compare `struct foo` objects instead of comparing them with `memcmp`.

`int` **strcmp** (`const char *`*s1*`, const char *`*s2*) Function

The `strcmp` function compares the string *s1* against *s2*, returning a value that has the same sign as the difference between the first differing pair of characters (interpreted as `unsigned char` objects, then promoted to `int`).

If the two strings are equal, `strcmp` returns `0`.

A consequence of the ordering used by `strcmp` is that if *s1* is an initial substring of *s2*, then *s1* is considered to be "less than" *s2*.

`strcmp` does not take sorting conventions of the language the strings are written in into account. To get that one has to use `strcoll`.

`int` **wcscmp** (`const wchar_t *`*ws1*`, const wchar_t *`*ws2*) Function

The `wcscmp` function compares the wide-character string *ws1* against *ws2*. The value returned is smaller than or larger than zero, depending on whether the first differing wide character *ws1* is smaller or larger than the corresponding character in *ws2*.

If the two strings are equal, `wcscmp` returns `0`.

A consequence of the ordering used by `wcscmp` is that if *ws1* is an initial substring of *ws2*, then *ws1* is considered to be "less than" *ws2*.

`wcscmp` does not take sorting conventions of the language the strings are written in into account. To get that one has to use `wcscoll`.

`int` **strcasecmp** (`const char *`*s1*`, const char *`*s2*) Function

This function is like `strcmp`, except that differences in case are ignored. How uppercase and lowercase characters are related is determined by the currently selected locale. In the standard `"C"` locale, the characters Ä and ä do not match, but in a locale that regards these characters as parts of the alphabet, they do match.

`strcasecmp` is derived from BSD.

`int` **wcscasecmp** (`const wchar_t *`*ws1*`, const wchar_T *`*ws2*) Function

This function is like `wcscmp`, except that differences in case are ignored. How uppercase and lowercase characters are related is determined by the currently selected locale. In the standard `"C"` locale, the characters Ä and ä do not match, but in a locale that regards these characters as parts of the alphabet, they do match.

`wcscasecmp` is a GNU extension.

`int` **strncmp** (`const char *`*s1*`, const char *`*s2*`, size_t` *size*) Function

This function is the similar to `strcmp`, except that no more than *size* wide characters are compared. In other words, if the two strings are the same in their first *size* wide characters, the return value is zero.

`int` **wcsncmp** (`const wchar_t *`*ws1*`, const wchar_t *`*ws2*`, size_t` *size*) Function

This function is the similar to `wcscmp`, except that no more than *size* wide characters are compared. In other words, if the two strings are the same in their first *size* wide characters, the return value is zero.

`int` **strncasecmp** (`const char *`*s1*`, const char *`*s2*`, size_t` *n*) Function

This function is like `strncmp`, except that differences in case are ignored. Like `strcasecmp`, how uppercase and lowercase characters are related is locale dependent.

`strncasecmp` is a GNU extension.

`int` **wcsncasecmp** (`const wchar_t *`*ws1*`, const wchar_t *`*s2*`, size_t` *n*) Function

This function is like `wcsncmp`, except that differences in case are ignored. Like `wcscasecmp`, how uppercase and lowercase characters are related is locale dependent.

`wcsncasecmp` is a GNU extension.

Here are some examples showing the use of `strcmp` and `strncmp` (equivalent examples can be constructed for the wide character functions). These examples assume the use of the ASCII character set. If some other character set—say, EBCDIC—is used instead, then the glyphs are associated with different numeric codes, and the return values and ordering may differ.

```
strcmp ("hello", "hello")
    ⇒ 0    /* These two strings are the same. */
strcmp ("hello", "Hello")
    ⇒ 32   /* Comparisons are case-sensitive. */
strcmp ("hello", "world")
    ⇒ -15  /* The character 'h' comes before 'w'. */
strcmp ("hello", "hello, world")
    ⇒ -44  /* Comparing a null character against a comma. */
strncmp ("hello", "hello, world", 5)
    ⇒ 0    /* The initial 5 characters are the same. */
strncmp ("hello, world", "hello, stupid world!!!", 5)
    ⇒ 0    /* The initial 5 characters are the same. */
```

`int` **strverscmp** (`const char *`*s1*`, const char *`*s2*) Function

The `strverscmp` function compares the string *s1* against *s2*, considering them as holding indices/version numbers. Return value follows the same conventions as found in the `strverscmp` function. In fact, if *s1* and *s2* contain no digits, `strverscmp` behaves like `strcmp`.

Basically, we compare strings normally (character-by-character) until we find a digit in each string. Then we enter a special comparison mode, where each

sequence of digits is taken as a whole. If we reach the end of these two parts without noticing a difference, we return to the standard comparison mode. There are two types of numeric parts: *integral* and *fractional* (those begin with a '0'). The types of the numeric parts affect the way we sort them:

- *integral/integral*: We compare values as you would expect.
- *fractional/integral*: The fractional part is less than the integral one.
- *fractional/fractional*: Things become a bit more complex. If the common prefix contains only leading zeros, the longest part is less than the other one; else the comparison behaves normally.

```
strverscmp ("no digit", "no digit")
    ⇒ 0    /* same behavior as strcmp. */
strverscmp ("item#99", "item#100")
    ⇒ <0   /* same prefix, but 99 < 100. */
strverscmp ("alpha1", "alpha001")
    ⇒ >0   /* fractional part inferior to integral one. */
strverscmp ("part1_f012", "part1_f01")
    ⇒ >0   /* two fractional parts. */
strverscmp ("foo.009", "foo.0")
    ⇒ <0   /* idem, but with leading zeros only. */
```

This function is especially useful when dealing with file-name sorting, because file names frequently hold indices and version numbers.

`strverscmp` is a GNU extension.

`int` **bcmp** (`const void *`*a1*`, const void *`*a2*`, size_t` *size*) Function

This is an obsolete alias for `memcmp`, derived from BSD.

5.6 Collation Functions

In some locales, the conventions for lexicographic ordering differ from the strict numeric ordering of character codes. For example, in Spanish most glyphs with diacritical marks such as accents are not considered distinct letters for the purposes of collation. On the other hand, the two-character sequence '`ll`' is treated as a single letter that is collated immediately after '`l`'.

You can use the functions `strcoll` and `strxfrm` (declared in the headers file '`string.h`') and `wcscoll` and `wcsxfrm` (declared in the headers file '`wchar`') to compare strings using a collation ordering appropriate for the current locale. The locale used by these functions in particular can be specified by setting the locale for the `LC_COLLATE` category (see Chapter 7 [Locales and Internationalization], page 181).

In the standard C locale, the collation sequence for `strcoll` is the same as that for `strcmp`. Similarly, `wcscoll` and `wcscmp` are the same in this situation.

Effectively, the way these functions work is by applying a mapping to transform the characters in a string to a byte sequence that represents the string's position in

the collating sequence of the current locale. Comparing two such byte-sequences in a simple fashion is equivalent to comparing the strings with the locale's collating sequence.

The functions `strcoll` and `wcscoll` perform this translation implicitly, in order to do one comparison. By contrast, `strxfrm` and `wcsxfrm` perform the mapping explicitly. If you are making multiple comparisons using the same string or set of strings, it is likely to be more efficient to use `strxfrm` or `wcsxfrm` to transform all the strings just once, and subsequently compare the transformed strings with `strcmp` or `wcscmp`.

`int` **strcoll** (`const char *`*s1*`, const char *`*s2*) Function

The `strcoll` function is similar to `strcmp` but uses the collating sequence of the current locale for collation (the `LC_COLLATE` locale).

`int` **wcscoll** (`const wchar_t *`*ws1*`, const wchar_t *`*ws2*) Function

The `wcscoll` function is similar to `wcscmp` but uses the collating sequence of the current locale for collation (the `LC_COLLATE` locale).

Here is an example of sorting an array of strings, using `strcoll` to compare them. The actual sort algorithm is not written here; it comes from `qsort` (see Section 12.3 [Array Sort Function], page 344). The job of the code shown here is to say how to compare the strings while sorting them (later on in this section, we will show a way to do this more efficiently using `strxfrm`).

```
/* This is the comparison function used with qsort. */

int
compare_elements (char **p1, char **p2)
{
  return strcoll (*p1, *p2);
}

/* This is the entry point—the function to sort
    strings using the locale's collating sequence. */

void
sort_strings (char **array, int nstrings)
{
  /* Sort temp_array by comparing the strings. */
  qsort (array, nstrings,
         sizeof (char *), compare_elements);
}
```

`size_t` **strxfrm** (`char *restrict` *to*`, const char *restrict` *from*`, size_t` *size*) Function

The function `strxfrm` transforms the string *from* using the collation transformation determined by the locale currently selected for collation, and stores the

transformed string in the array *to*. Up to *size* characters (including a terminating null character) are stored.

The behavior is undefined if the strings *to* and *from* overlap (see Section 5.4 [Copying and Concatenation], page 93).

The return value is the length of the entire transformed string. This value is not affected by the value of *size*, but if it is greater than or equal to *size*, it means that the transformed string did not entirely fit in the array *to*. In this case, only as much of the string as actually fits was stored. To get the whole transformed string, call `strxfrm` again with a bigger output array.

The transformed string may be longer than the original string, and it may also be shorter.

If *size* is zero, no characters are stored in *to*. In this case, `strxfrm` simply returns the number of characters that would be the length of the transformed string. This is useful for determining what size the allocated array should be. It does not matter what *to* is if *size* is zero; *to* may even be a null pointer.

`size_t` **wcsxfrm** (`wchar_t *restrict` *wto*`, const wchar_t *`*wfrom*`, size_t` *size*) Function

The function `wcsxfrm` transforms wide-character string *wfrom* using the collation transformation determined by the locale currently selected for collation, and stores the transformed string in the array *wto*. Up to *size* wide characters (including a terminating null character) are stored.

The behavior is undefined if the strings *wto* and *wfrom* overlap (see Section 5.4 [Copying and Concatenation], page 93).

The return value is the length of the entire transformed wide-character string. This value is not affected by the value of *size*, but if it is greater than or equal to *size*, it means that the transformed wide-character string did not entirely fit in the array *wto*. In this case, only as much of the wide-character string as actually fits was stored. To get the whole transformed wide-character string, call `wcsxfrm` again with a bigger output array.

The transformed wide-character string may be longer than the original wide-character string, and it may also be shorter.

If *size* is zero, no characters are stored in *to*. In this case, `wcsxfrm` simply returns the number of wide characters that would be the length of the transformed wide-character string. This is useful for determining what size the allocated array should be (remember to multiply with `sizeof (wchar_t)`). It does not matter what *wto* is if *size* is zero; *wto* may even be a null pointer.

Here is an example of how you can use `strxfrm` when you plan to do many comparisons. It does the same thing as the previous example, but much faster, because it has to transform each string only once, no matter how many times it is compared with other strings. Even the time needed to allocate and free storage is much less than the time we save, when there are many strings.

```
struct sorter { char *input; char *transformed; };
```

```
/* This is the comparison function used with qsort
   to sort an array of struct sorter. */

int
compare_elements (struct sorter *p1, struct sorter *p2)
{
  return strcmp (p1->transformed, p2->transformed);
}

/* This is the entry point—the function to sort
   strings using the locale's collating sequence. */

void
sort_strings_fast (char **array, int nstrings)
{
  struct sorter temp_array[nstrings];
  int i;

  /* Set up temp_array. Each element contains
     one input string and its transformed string. */
  for (i = 0; i < nstrings; i++)
    {
      size_t length = strlen (array[i]) * 2;
      char *transformed;
      size_t transformed_length;

      temp_array[i].input = array[i];

      /* First make a buffer that you guess is big enough. */
      transformed = (char *) xmalloc (length);

      /* Transform array[i]. */
      transformed_length = strxfrm (transformed, array[i], length);

      /* If the buffer was not large enough, resize it
         and try again. */
      if (transformed_length >= length)
        {
          /* Allocate the needed space. +1 for terminating
             NUL character. */
          transformed = (char *) xrealloc (transformed,
                                           transformed_length + 1);

          /* The return value is not interesting because we know
```

```
                how long the transformed string is.  */
             (void) strxfrm (transformed, array[i],
                             transformed_length + 1);
           }

         temp_array[i].transformed = transformed;
       }

     /* Sort temp_array by comparing transformed strings.  */
     qsort (temp_array, sizeof (struct sorter),
            nstrings, compare_elements);

     /* Put the elements back in the permanent array
        in their sorted order.  */
     for (i = 0; i < nstrings; i++)
       array[i] = temp_array[i].input;

     /* Free the strings we allocated.  */
     for (i = 0; i < nstrings; i++)
       free (temp_array[i].transformed);
   }
```

The interesting part of this code for the wide-character version would look like this:

```
void
sort_strings_fast (wchar_t **array, int nstrings)
{
  ...
       /* Transform array[i].  */
       transformed_length = wcsxfrm (transformed, array[i], length);

       /* If the buffer was not large enough, resize it
          and try again.  */
       if (transformed_length >= length)
         {
           /* Allocate the needed space. +1 for terminating
              NUL character.  */
           transformed = (wchar_t *) xrealloc (transformed,
                                               (transformed_length + 1)
                                               * sizeof (wchar_t));

           /* The return value is not interesting because we know
              how long the transformed string is.  */
           (void) wcsxfrm (transformed, array[i],
                           transformed_length + 1);
```

```
        }
    ...
```

Note the additional multiplication with `sizeof (wchar_t)` in the `realloc` call.

Compatibility Note: The string collation functions are a new feature of ISO C90. Older C dialects have no equivalent feature. The wide-character versions were introduced in Amendment 1 to ISO C90.

5.7 Search Functions

This section describes library functions that perform various kinds of search operations on strings and arrays. These functions are declared in the header file '`string.h`'.

`void *` **memchr** (`const void *`*block*`, int` *c*`, size_t` *size*) Function

This function finds the first occurrence of the byte *c* (converted to an `unsigned char`) in the initial *size* bytes of the object beginning at *block*. The return value is a pointer to the located byte, or a null pointer if no match was found.

`wchar_t *` **wmemchr** (`const wchar_t *`*block*`, wchar_t` *wc*`, size_t` *size*) Function

This function finds the first occurrence of the wide character *wc* in the initial *size* wide characters of the object beginning at *block*. The return value is a pointer to the located wide character, or a null pointer if no match was found.

`void *` **rawmemchr** (`const void *`*block*`, int` *c*) Function

Often the `memchr` function is used with the knowledge that the byte *c* is available in the memory block specified by the parameters. But this means that the *size* parameter is not really needed and that the tests performed with it at run time (to check whether the end of the block is reached) are not needed.

The `rawmemchr` function exists for just this situation, which is surprisingly frequent. The interface is similar to `memchr` except that the *size* parameter is missing. The function will look beyond the end of the block pointed to by *block* in case the programmer made an error in assuming that the byte *c* is present in the block. In this case, the result is unspecified. Otherwise, the return value is a pointer to the located byte.

This function is of special interest when looking for the end of a string. Since all strings are terminated by a null byte a call like:

```
rawmemchr (str, '\0')
```

will never go beyond the end of the string.

This function is a GNU extension.

`void *` **memrchr** (`const void *`*block*`, int` *c*`, size_t` *size*) Function

The function `memrchr` is like `memchr`, except that it searches backward from the end of the block defined by *block* and *size* (instead of forward from the front).

`char *` **strchr** (`const char *`*string*`, int` *c*) Function

The `strchr` function finds the first occurrence of the character *c* (converted to a `char`) in the null-terminated string beginning at *string*. The return value is a pointer to the located character, or a null pointer if no match was found.

For example:

```
strchr ("hello, world", 'l')
    ⇒ "llo, world"
strchr ("hello, world", '?')
    ⇒ NULL
```

The terminating null character is considered to be part of the string, so you can use this function get a pointer to the end of a string by specifying a null character as the value of the *c* argument. It would be better (but less portable) to use `strchrnul` in this case, though.

`wchar_t *` **wcschr** (`const wchar_t *`*wstring*`, int` *wc*) Function

The `wcschr` function finds the first occurrence of the wide character *wc* in the null-terminated wide-character string beginning at *wstring*. The return value is a pointer to the located wide character, or a null pointer if no match was found.

The terminating null character is considered to be part of the wide character string, so you can use this function get a pointer to the end of a wide-character string by specifying a null wide character as the value of the *wc* argument. It would be better (but less portable) to use `wcschrnul` in this case, though.

`char *` **strchrnul** (`const char *`*string*`, int` *c*) Function

`strchrnul` is the same as `strchr` except that if it does not find the character, it returns a pointer to *string*'s terminating null character rather than a null pointer.

This function is a GNU extension.

`wchar_t *` **wcschrnul** (`const wchar_t *`*wstring*`, wchar_t` *wc*) Function

`wcschrnul` is the same as `wcschr` except that if it does not find the wide character, it returns a pointer to the wide-character string's terminating null wide character rather than a null pointer.

This function is a GNU extension.

One useful but unusual use of the `strchr` function is when you want to have a pointer pointing to the NUL byte terminating a string. This is often written in this way:

```
s += strlen (s);
```

This is almost optimal, but the addition operation duplicated a bit of the work already done in the `strlen` function. A better solution is this:

```
s = strchr (s, '\0');
```

There is no restriction on the second parameter of `strchr`, so it could very well also be the NUL character. The `strchr` function is more expensive than the `strlen` function, since we have two abort criteria. But in the GNU C Library, the implementation of `strchr` is optimized in a special way, so that `strchr` actually is faster.

`char *` **strrchr** (`const char *`*string*`, int` *c*) Function

The function `strrchr` is like `strchr`, except that it searches backward from the end of the string *string* (instead of forward from the front).

For example:

```
strrchr ("hello, world", 'l')
    ⇒ "ld"
```

`wchar_t *` **wcsrchr** (`const wchar_t *`*wstring*`, wchar_t` *c*) Function

The function `wcsrchr` is like `wcschr`, except that it searches backwards from the end of the string *wstring* (instead of forward from the front).

`char *` **strstr** (`const char *`*haystack*`, const char *`*needle*) Function

This is like `strchr`, except that it searches *haystack* for a substring *needle* rather than just a single character. It returns a pointer into the string *haystack* that is the first character of the substring, or a null pointer if no match was found. If *needle* is an empty string, the function returns *haystack*.

For example:

```
strstr ("hello, world", "l")
    ⇒ "llo, world"
strstr ("hello, world", "wo")
    ⇒ "world"
```

`wchar_t *` **wcsstr** (`const wchar_t *`*haystack*`, const wchar_t *`*needle*) Function

This is like `wcschr`, except that it searches *haystack* for a substring *needle* rather than just a single wide character. It returns a pointer into the string *haystack* that is the first wide character of the substring, or a null pointer if no match was found. If *needle* is an empty string, the function returns *haystack*.

`wchar_t *` **wcswcs** (`const wchar_t *`*haystack*`, const wchar_t *`*needle*) Function

`wcsstr` is a deprecated alias for `wcsstr`. This is the name originally used in the *X/Open Portability Guide*[1] before the Amendment 1 to ISO C90 was published.

`char *` **strcasestr** (`const char *`*haystack*`, const char *`*needle*) Function

This is like `strstr`, except that it ignores case in searching for the substring. Like `strcasecmp`, how uppercase and lowercase characters are related is locale dependent.

For example:

```
strstr ("hello, world", "L")
    ⇒ "llo, world"
strstr ("hello, World", "wo")
    ⇒ "World"
```

`void *` **memmem** (`const void *`*haystack*`, size_t` *haystack-len*`, const void *`*needle*`, size_t` *needle-len*) Function

This is like `strstr`, but *needle* and *haystack* are byte arrays rather than null-terminated strings. *needle-len* is the length of *needle* and *haystack-len* is the length of *haystack*.

This function is a GNU extension.

`size_t` **strspn** (`const char *`*string*`, const char *`*skipset*) Function

The `strspn` ("string span") function returns the length of the initial substring of *string* that consists entirely of characters that are members of the set specified by the string *skipset*. The order of the characters in *skipset* is not important.

For example:

```
strspn ("hello, world", "abcdefghijklmnopqrstuvwxyz")
    ⇒ 5
```

Character used here in the sense of byte. In a string using a multibyte-character encoding (abstract), characters consisting of more than one byte are not treated as an entity. Each byte is treated separately. The function is not locale dependent.

`size_t` **wcsspn** (`const wchar_t *`*wstring*`, const wchar_t *`*skipset*) Function

The `wcsspn` ("wide-character string span") function returns the length of the initial substring of *wstring* that consists entirely of wide characters that are members of the set specified by the string *skipset*. The order of the wide characters in *skipset* is not important.

---

[1] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

`size_t` **strcspn** (`const char *`*string*`, const char *`*stopset*) Function

The `strcspn` ("string complement span") function returns the length of the initial substring of *string* that consists entirely of characters that are *not* members of the set specified by the string *stopset*. (In other words, it returns the offset of the first character in *string* that is a member of the set *stopset*.)

For example:

```
strcspn ("hello, world", " \t\n,.;!?")
    ⇒ 5
```

Character is used here in the sense of byte. In a string using a multibyte-character encoding (abstract), characters consisting of more than one byte are not treated as an entity. Each byte is treated separately. The function is not locale dependent.

`size_t` **wcscspn** (`const wchar_t *`*wstring*`, const wchar_t *`*stopset*) Function

The `wcscspn` ("wide-character string complement span") function returns the length of the initial substring of *wstring* that consists entirely of wide characters that are *not* members of the set specified by the string *stopset*. (In other words, it returns the offset of the first character in *string* that is a member of the set *stopset*.)

`char *` **strpbrk** (`const char *`*string*`, const char *`*stopset*) Function

The `strpbrk` ("string pointer break") function is related to `strcspn`, except that it returns a pointer to the first character in *string* that is a member of the set *stopset* instead of the length of the initial substring. It returns a null pointer if no such character from *stopset* is found.

For example:

```
strpbrk ("hello, world", " \t\n,.;!?")
    ⇒ ", world"
```

Character is used here in the sense of byte. In a string using a multibyte-character encoding (abstract), characters consisting of more than one byte are not treated as an entity. Each byte is treated separately. The function is not locale dependent.

`wchar_t *` **wcspbrk** (`const wchar_t *`*wstring*`, const wchar_t *`*stopset*) Function

The `wcspbrk` ("wide-character string pointer break") function is related to `wcscspn`, except that it returns a pointer to the first wide character in *wstring* that is a member of the set *stopset* instead of the length of the initial substring. It returns a null pointer if no such character from *stopset* is found.

5.7.1 Compatibility String Search Functions

`char *` **index** (`const char *`*string*`, int` *c*) Function

`index` is another name for `strchr`; they are exactly the same. New code should always use `strchr`, since this name is defined in ISO C while `index` is a BSD invention that was never available on System V derived systems.

`char *` **rindex** (`const char *`*string*`, int` *c*) Function

`rindex` is another name for `strrchr`; they are exactly the same. New code should always use `strrchr`, since this name is defined in ISO C while `rindex` is a BSD invention that was never available on System V derived systems.

5.8 Finding Tokens in a String

It's fairly common for programs to have a need to do some simple kinds of lexical analysis and parsing, such as splitting a command string up into tokens. You can do this with the `strtok` function, declared in the header file '`string.h`'.

`char *` **strtok** (`char *restrict` *newstring*`, const char *restrict` *delimiters*) Function

A string can be split into tokens by making a series of calls to the function `strtok`.

The string to be split up is passed as the *newstring* argument on the first call only. The `strtok` function uses this to set up some internal state information. Subsequent calls to get additional tokens from the same string are indicated by passing a null pointer as the *newstring* argument. Calling `strtok` with another nonnull *newstring* argument reinitializes the state information. It is guaranteed that no other library function ever calls `strtok` behind your back (which would mess up this internal state information).

The *delimiters* argument is a string that specifies a set of delimiters that may surround the token being extracted. All the initial characters that are members of this set are discarded. The first character that is *not* a member of this set of delimiters marks the beginning of the next token. The end of the token is found by looking for the next character that is a member of the delimiter set. This character in the original string *newstring* is overwritten by a null character, and the pointer to the beginning of the token in *newstring* is returned.

On the next call to `strtok`, the searching begins at the next character beyond the one that marked the end of the previous token. Note that the set of delimiters *delimiters* do not have to be the same on every call in a series of calls to `strtok`.

If the end of the string *newstring* is reached, or if the remainder of the string consists only of delimiter characters, `strtok` returns a null pointer.

Character is used here in the sense of byte. In a string using a multibyte-character encoding (abstract), characters consisting of more than 1 byte are not

treated as an entity. Each byte is treated separately. The function is not locale dependent.

`wchar_t *` **wcstok** (`wchar_t *`*newstring*`, const char *`*delimiters*) Function

A string can be split into tokens by making a series of calls to the function `wcstok`.

The string to be split up is passed as the *newstring* argument on the first call only. The `wcstok` function uses this to set up some internal state information. Subsequent calls to get additional tokens from the same wide-character string are indicated by passing a null pointer as the *newstring* argument. Calling `wcstok` with another nonnull *newstring* argument reinitializes the state information. It is guaranteed that no other library function ever calls `wcstok` behind your back (which would mess up this internal state information).

The *delimiters* argument is a wide-character string that specifies a set of delimiters that may surround the token being extracted. All the initial wide characters that are members of this set are discarded. The first wide character that is *not* a member of this set of delimiters marks the beginning of the next token. The end of the token is found by looking for the next wide character that is a member of the delimiter set. This wide character in the original wide-character string *newstring* is overwritten by a null wide character, and the pointer to the beginning of the token in *newstring* is returned.

On the next call to `wcstok`, the searching begins at the next wide character beyond the one that marked the end of the previous token. The set of delimiters *delimiters* do not have to be the same on every call in a series of calls to `wcstok`.

If the end of the wide-character string *newstring* is reached, or if the remainder of the string consists only of delimiter wide characters, `wcstok` returns a null pointer.

Character is used here in the sense of byte. In a string using a multibyte-character encoding (abstract), characters consisting of more than one byte are not treated as an entity. Each byte is treated separately. The function is not locale dependent.

Warning: Since `strtok` and `wcstok` alter the string they are parsing, you should always copy the string to a temporary buffer before parsing it with `strtok`/`wcstok` (see Section 5.4 [Copying and Concatenation], page 93). If you allow `strtok` or `wcstok` to modify a string that came from another part of your program, you are asking for trouble; that string might be used for other purposes after `strtok` or `wcstok` has modified it, and it would not have the expected value.

The string that you are operating on might even be a constant. Then when `strtok` or `wcstok` tries to modify it, your program will get a fatal signal for writing in read-only memory.[2] Even if the operation of `strtok` or `wcstok` would

[2] See Loosemore et al., "Program-Error Signals" (see chap. 1, n. 1).

not require a modification of the string (e.g., if there is exactly one token), the string can (and in the GNU libc case will) be modified.

This is a special case of a general principle: if a part of a program does not have as its purpose the modification of a certain data structure, then to modify the data structure temporarily is to risk errors.

The functions `strtok` and `wcstok` are not reentrant.[3]

Here is a simple example showing the use of `strtok`:

```
#include <string.h>
#include <stddef.h>

...

const char string[] = "words separated by spaces -- and, punctuation!";
const char delimiters[] = " .,;:!-";
char *token, *cp;

...

cp = strdupa (string);                /* Make writable copy.  */
token = strtok (cp, delimiters);      /* token => "words" */
token = strtok (NULL, delimiters);    /* token => "separated" */
token = strtok (NULL, delimiters);    /* token => "by" */
token = strtok (NULL, delimiters);    /* token => "spaces" */
token = strtok (NULL, delimiters);    /* token => "and" */
token = strtok (NULL, delimiters);    /* token => "punctuation" */
token = strtok (NULL, delimiters);    /* token => NULL */
```

The GNU C Library contains two more functions for tokenizing a string which overcome the limitation of nonreentrancy. They are only available for multibyte-character strings.

`char *` **strtok_r** (`char *`*newstring*`, const char *`*delimiters*`, char **`*save_ptr*) Function

Just like `strtok`, this function splits the string into several tokens that can be accessed by successive calls to `strtok_r`. The difference is that the information about the next token is stored in the space pointed to by the third argument, *save_ptr*, which is a pointer to a string pointer. Calling `strtok_r` with a null pointer for *newstring* and leaving *save_ptr* between the calls unchanged does the job without hindering reentrancy.

This function is defined in POSIX.1 and can be found on many systems which support multithreading.

[3] See Loosemore et al., "Signal Handling and Nonreentrant Functions", for a discussion of where and why reentrancy is important.

`char *` **strsep** (`char **`*string_ptr*`, const char *`*delimiter*) Function

This function has a similar functionality to `strtok_r`, with the *newstring* argument replaced by the *save_ptr* argument. The initialization of the moving pointer has to be done by the user. Successive calls to `strsep` move the pointer along the tokens separated by *delimiter*, returning the address of the next token and updating *string_ptr* to point to the beginning of the next token.

One difference between `strsep` and `strtok_r` is that if the input string contains more than one character from *delimiter* in a row, `strsep` returns an empty string for each pair of characters from *delimiter*. This means that a program normally should test for `strsep` returning an empty string before processing it.

This function was introduced in 4.3BSD and therefore is widely available.

Here is how the above example looks like when `strsep` is used:

```
#include <string.h>
#include <stddef.h>

...

const char string[] = "words separated by spaces -- and, punctuation!";
const char delimiters[] = " .,;:!-";
char *running;
char *token;

...

running = strdupa (string);
token = strsep (&running, delimiters);    /* token => "words" */
token = strsep (&running, delimiters);    /* token => "separated" */
token = strsep (&running, delimiters);    /* token => "by" */
token = strsep (&running, delimiters);    /* token => "spaces" */
token = strsep (&running, delimiters);    /* token => "" */
token = strsep (&running, delimiters);    /* token => "" */
token = strsep (&running, delimiters);    /* token => "" */
token = strsep (&running, delimiters);    /* token => "and" */
token = strsep (&running, delimiters);    /* token => "" */
token = strsep (&running, delimiters);    /* token => "punctuation" */
token = strsep (&running, delimiters);    /* token => "" */
token = strsep (&running, delimiters);    /* token => NULL */
```

`char *` **basename** (`const char *`*filename*) Function

The GNU version of the `basename` function returns the last component of the path in *filename*. This function is the preferred usage, since it does not modify the argument, *filename*, and respects trailing slashes. The prototype for `basename` can be found in '`string.h`'. This function is overriden by the XPG version, if '`libgen.h`' is included.

Here is an example using GNU `basename`:

```
#include <string.h>

int
main (int argc, char *argv[])
{
  char *prog = basename (argv[0]);

  if (argc < 2)
    {
      fprintf (stderr, "Usage %s <arg>\n", prog);
      exit (1);
    }

  ...
}
```

Portability Note: This function may produce different results on different systems.

`char *` **basename** (`char *`*path*) Function

This is the standard XPG-defined `basename`. It is similar in spirit to the GNU version, but may modify the *path* by removing trailing '`/`' characters. If the *path* is made up entirely of '`/`' characters, then "/" will be returned. Also, if *path* is `NULL` or an empty string, then "." is returned. The prototype for the XPG version can be found in '`libgen.h`'.

Here is an example using XPG `basename`:

```
#include <libgen.h>

int
main (int argc, char *argv[])
{
  char *prog;
  char *path = strdupa (argv[0]);

  prog = basename (path);

  if (argc < 2)
    {
      fprintf (stderr, "Usage %s <arg>\n", prog);
      exit (1);
    }

  ...
```

```
}
```

`char *` **dirname** (`char *`*path*) Function

The `dirname` function is the compliment to the XPG version of `basename`. It returns the parent directory of the file specified by *path*. If *path* is `NULL`, an empty string, or contains no '`/`' characters, then "." is returned. The prototype for this function can be found in '`libgen.h`'.

5.9 `strfry`

The function below addresses the perennial programming quandary, "How do I take good data in string form and painlessly turn it into garbage?" This is actually a fairly simple task for C programmers who do not use the GNU C Library string functions, but for programs based on the GNU C Library, the `strfry` function is the preferred method for destroying string data.

The prototype for this function is in '`string.h`'.

`char *` **strfry** (`char *`*string*) Function

`strfry` creates a pseudorandom anagram of a string, replacing the input with the anagram. For each position in the string, `strfry` swaps it with a position in the string selected at random (from a uniform distribution). The two positions may be the same.

The return value of `strfry` is always *string*.

Portability Note: This function is unique to the GNU C Library.

5.10 Trivial Encryption

The `memfrob` function converts an array of data to something unrecognizable and back again. It is not encryption in its usual sense since it is easy for someone to convert the encrypted data back to clear text. The transformation is analogous to Usenet's "Rot13" encryption method for obscuring offensive jokes from sensitive eyes and such. Unlike Rot13, `memfrob` works on arbitrary binary data, not just text.[4]

This function is declared in '`string.h`'.

`void *` **memfrob** (`void *`*mem*`, size_t` *length*) Function

`memfrob` transforms (frobnicates) each byte of the data structure at *mem*, which is *length* bytes long, by bit-wise exclusive-oring it with binary 00101010. It does the transformation in place and its return value is always *mem*.

`memfrob` a second time on the same data structure returns it to its original state.

[4] See Loosemore et al., "DES Encryption and Password Handling", for a discussion of true encryption.

This is a good function for hiding information from someone who doesn't want to see it. To really prevent people from retrieving the information, use stronger encryption.[5]

Portability Note: This function is unique to the GNU C Library.

5.11 Encode Binary Data

To store or transfer binary data in environments that only support text, you have to encode the binary data by mapping the input bytes to characters in the range allowed for storing or transferring. SVID systems (and nowadays XPG-compliant systems) provide minimal support for this task.

`char *` **l64a** (`long int` *n*) Function

This function encodes a 32-bit input value using characters from the basic character set. It returns a pointer to a 6-character buffer that contains an encoded version of *n*. To encode a series of bytes, the user must copy the returned string to a destination buffer. It returns the empty string if *n* is zero, which is somewhat bizarre but mandated by the standard.

Warning: Since a static buffer is used, this function should not be used in multithreaded programs. There is no thread-safe alternative to this function in the C library.

Compatibility Note: The XPG standard states that the return value of `l64a` is undefined if *n* is negative. In the GNU implementation, `l64a` treats its argument as unsigned, so it will return a sensible encoding for any nonzero *n*; however, portable programs should not rely on this.

To encode a large buffer `l64a` must be called in a loop, once for each 32-bit word of the buffer. For example, one could do something like this:

```
char *
encode (const void *buf, size_t len)
{
  /* We know in advance how long the buffer has to be. */
  unsigned char *in = (unsigned char *) buf;
  char *out = malloc (6 + ((len + 3) / 4) * 6 + 1);
  char *cp = out;

  /* Encode the length. */
  /* Using 'htonl' is necessary so that the data can be
     decoded even on machines with different byte order. */

  cp = mempcpy (cp, l64a (htonl (len)), 6);

  while (len > 3)
```

[5] Ibid., "DESEncryption and Password Handling".

```
    {
      unsigned long int n = *in++;
      n = (n << 8) | *in++;
      n = (n << 8) | *in++;
      n = (n << 8) | *in++;
      len -= 4;
      if (n)
        cp = mempcpy (cp, l64a (htonl (n)), 6);
      else
            /* 'l64a' returns the empty string for n==0, so we
               must generate its encoding ("......") by hand. */
        cp = stpcpy (cp, "......");
    }
  if (len > 0)
    {
      unsigned long int n = *in++;
      if (--len > 0)
        {
          n = (n << 8) | *in++;
          if (--len > 0)
            n = (n << 8) | *in;
        }
      memcpy (cp, l64a (htonl (n)), 6);
      cp += 6;
    }
  *cp = '\0';
  return out;
}
```

To decode data produced with `l64a`, the following function should be used.

`long int` **a64l** (`const char *`*string*) Function

The parameter *string* should contain a string that was produced by a call to `l64a`. The function processes at least 6 characters of this string, and decodes the characters it finds according to the table below. It stops decoding when it finds a character not in the table, rather like `atoi`; if you have a buffer that has been broken into lines, you must be careful to skip over the end-of-line characters.

The decoded number is returned as a `long int` value.

The `l64a` and `a64l` functions use a base-64 encoding, in which each character of an encoded string represents 6 bits of an input word. These symbols are used for the base-64 digits:

	0	1	2	3	4	5	6	7
0	.	/	0	1	2	3	4	5

```
8       6    7    8    9    A    B    C    D
16      E    F    G    H    I    J    K    L
24      M    N    O    P    Q    R    S    T
32      U    V    W    X    Y    Z    a    b
40      c    d    e    f    g    h    i    j
48      k    l    m    n    o    p    q    r
56      s    t    u    v    w    x    y    z
```

This encoding scheme is not standard. There are some other encoding methods that are much more widely used (UU encoding, MIME encoding). Generally, it is better to use one of these encodings.

5.12 Argz and Envz Vectors

argz vectors are vectors of strings in a contiguous block of memory, each element separated from its neighbors by null-characters (`'\0'`).

envz vectors are an extension of argz vectors where each element is a name-value pair, separated by a `'='` character (as in a Unix environment).

5.12.1 Argz Functions

Each argz vector is represented by a pointer to the first element, of type `char *`, and a size, of type `size_t`, both of which can be initialized to `0` to represent an empty argz vector. All argz functions accept either a pointer and a size argument, or pointers to them, if they will be modified.

The argz functions use `malloc`/`realloc` to allocate and grow argz vectors, so any argz vector created using these functions may be freed by using `free`; conversely, any argz function that may grow a string expects that string to have been allocated using `malloc` (those argz functions that only examine their arguments or modify them in place will work on any sort of memory). See Section 3.2.2 [Unconstrained Allocation], page 42, for more information.

All argz functions that do memory allocation have a return type of `error_t`, and return `0` for success or `ENOMEM` if an allocation error occurs.

These functions are declared in the standard include file '`argz.h`'.

`error_t` **argz_create** (`char *const` *argv*`[],` `char **`*argz*`,` `size_t *`*argz_len*) Function

The `argz_create` function converts the Unix-style argument vector *argv* (a vector of pointers to normal C strings, terminated by `(char *)0`; see Section 14.1 [Program Arguments], page 379) into an argz vector with the same elements, which is returned in *argz* and *argz_len*.

`error_t` **argz_create_sep** (`const char *`*string*`, int` *sep*`, char **`*argz*`, size_t *`*argz_len*) Function

The `argz_create_sep` function converts the null-terminated string *string* into an argz vector (returned in *argz* and *argz_len*) by splitting it into elements at every occurrence of the character *sep*.

`size_t` **argz_count** (`const char *`*argz*`, size_t` *arg_len*) Function

Returns the number of elements in the argz vector *argz* and *argz_len*.

`void` **argz_extract** (`char *`*argz*`, size_t` *argz_len*`, char **`*argv*) Function

The `argz_extract` function converts the argz vector *argz* and *argz_len* into a Unix-style argument vector stored in *argv*, by putting pointers to every element in *argz* into successive positions in *argv*, followed by a terminator of `0`. *Argv* must be preallocated with enough space to hold all the elements in *argz* plus the terminating `(char *)0 ((argz_count (`*argz*`, `*argz_len*`) + 1) * sizeof (char *)` bytes should be enough). The string pointers stored into *argv* point into *argz*—they are not copies—and so *argz* must be copied if it will be changed while *argv* is still active. This function is useful for passing the elements in *argz* to an exec function.[6]

`void` **argz_stringify** (`char *`*argz*`, size_t` *len*`, int` *sep*) Function

The `argz_stringify` converts *argz* into a normal string with the elements separated by the character *sep*, by replacing each `'\0'` inside *argz* (except the last one, which terminates the string) with *sep*. This is handy for printing *argz* in a readable manner.

`error_t` **argz_add** (`char **`*argz*`, size_t *`*argz_len*`, const char *`*str*) Function

The `argz_add` function adds the string *str* to the end of the argz vector *argz, and updates *argz and *argz_len accordingly.

`error_t` **argz_add_sep** (`char **`*argz*`, size_t *`*argz_len*`, const char *`*str*`, int` *delim*) Function

The `argz_add_sep` function is similar to `argz_add`, but *str* is split into separate elements in the result at occurrences of the character *delim*. This is useful, for instance, for adding the components of a Unix search path to an argz vector, by using a value of `':'` for *delim*.

`error_t` **argz_append** (`char **`*argz*`, size_t *`*argz_len*`, const char *`*buf*`, size_t` *buf_len*) Function

The `argz_append` function appends *buf_len* bytes starting at *buf* to the argz vector *argz, reallocating *argz to accommodate it, and adding *buf_len* to *argz_len.

[6] Ibid., “Executing a File”.

`error_t` **argz_delete** (`char **`*argz*`, size_t *`*argz_len*`, char *`*entry*) Function

If *entry* points to the beginning of one of the elements in the argz vector **argz*, the `argz_delete` function will remove this entry and reallocate **argz*, modifying **argz* and **argz_len* accordingly. As destructive argz functions usually reallocate their argz argument, pointers into argz vectors such as *entry* will then become invalid.

`error_t` **argz_insert** (`char **`*argz*`, size_t *`*argz_len*`, char *`*before*`, const char *`*entry*) Function

The `argz_insert` function inserts the string *entry* into the argz vector **argz* at a point just before the existing element pointed to by *before*, reallocating **argz* and updating **argz* and **argz_len*. If *before* is `0`, *entry* is added to the end instead (as if by `argz_add`). Since the first element is in fact the same as **argz*, passing in **argz* as the value of *before* will result in *entry* being inserted at the beginning.

`char *` **argz_next** (`char *`*argz*`, size_t` *argz_len*`, const char *`*entry*) Function

The `argz_next` function provides a convenient way of iterating over the elements in the argz vector *argz*. It returns a pointer to the next element in *argz* after the element *entry*, or `0` if there are no elements following *entry*. If *entry* is `0`, the first element of *argz* is returned.

This behavior suggests two styles of iteration:

```
char *entry = 0;
while ((entry = argz_next (argz, argz_len, entry)))
  action;
```

(the double parentheses are necessary to make some C compilers shut up about what they consider a questionable `while`-test) and:

```
char *entry;
for (entry = argz;
     entry;
     entry = argz_next (argz, argz_len, entry))
  action;
```

The latter depends on *argz* having a value of `0` if it is empty (rather than a pointer to an empty block of memory); this invariant is maintained for argz vectors created by the functions here.

`error_t` **argz_replace** (`char **`*argz*`, size_t *`*argz_len*`, const char *`*str*`, const char *`*with*`, unsigned *`*replace_count*) Function

Replace any occurrences of the string *str* in *argz* with *with*, reallocating *argz* as necessary. If *replace_count* is nonzero, **replace_count* will be incremented by the number of replacements performed.

5.12.2 Envz Functions

Envz vectors are just argz vectors with additional constraints on the form of each element; as such, argz functions can also be used on them, where it makes sense.

Each element in an envz vector is a name-value pair, separated by a `'='` character; if multiple `'='` characters are present in an element, those after the first are considered part of the value, and treated like all other non-`'\0'` characters.

If *no* `'='` characters are present in an element, that element is considered the name of a "null" entry, as distinct from an entry with an empty value: `envz_get` will return `0` if given the name of null entry, whereas an entry with an empty value would result in a value of `""`; `envz_entry` will still find such entries, however. Null entries can be removed with `envz_strip` function.

As with argz functions, envz functions that may allocate memory (and thus fail) have a return type of `error_t`, and return either `0` or `ENOMEM`.

These functions are declared in the standard include file '`envz.h`'.

`char *` **envz_entry** (`const char *`*envz*`, size_t` *envz_len*`,` Function
`const char *`*name*)

The `envz_entry` function finds the entry in *envz* with the name *name*, and returns a pointer to the whole entry—the argz element that begins with *name* followed by an `'='` character. If there is no entry with that name, `0` is returned.

`char *` **envz_get** (`const char *`*envz*`, size_t` *envz_len*`,` Function
`const char *`*name*)

The `envz_get` function finds the entry in *envz* with the name *name* (like `envz_entry`) and returns a pointer to the value portion of that entry (following the `'='`). If there is no entry with that name (or only a null entry), `0` is returned.

`error_t` **envz_add** (`char **`*envz*`, size_t *`*envz_len*`,` Function
`const char *`*name*`, const char *`*value*)

The `envz_add` function adds an entry to **envz* (updating **envz* and **envz_len*) with the name *name*, and value *value*. If an entry with the same name already exists in *envz*, it is removed first. If *value* is `0`, then the new entry will be the special null type of entry mentioned above.

`error_t` **envz_merge** (`char **`*envz*`, size_t *`*envz_len*`,` Function
`const char *`*envz2*`, size_t` *envz2_len*`, int` *override*)

The `envz_merge` function adds each entry in *envz2* to *envz*, as if with `envz_add`, updating **envz* and **envz_len*. If *override* is true, then values in *envz2* will supersede those with the same name in *envz*, otherwise they will not.

Null entries are treated just like other entries in this respect, so a null entry in *envz* can prevent an entry of the same name in *envz2* from being added to *envz*, if *override* is false.

`void` **envz_strip** (`char **`*envz*`, size_t *`*envz_len*) Function

The `envz_strip` function removes any null entries from *envz*, updating **envz* and **envz_len*.

6 Character-Set Handling

Character sets used in the early days of computing had only 6, 7, or 8 bits for each character—there was never a case where more than 8 bits (1 byte) were used to represent a single character. The limitations of this approach became more apparent as more people grappled with non-Roman character sets, where all the characters that make up a language's character set cannot be represented by 2^8 choices. This chapter shows the functionality that was added to the C library to support multiple character sets.

6.1 Introduction to Extended Characters

A variety of solutions is available to overcome the differences between character sets with a 1:1 relation between bytes and characters and character sets with ratios of 2:1 or 4:1. The remainder of this section gives a few examples to help you understand the design decisions made while developing the functionality of the C library.

A distinction we have to make right away is between internal and external representation. *Internal representation* means the representation used by a program while keeping the text in memory. External representations are used when text is stored or transmitted through some communication channel. Examples of external representations include files waiting in a directory to be read and parsed.

Traditionally, there has been no difference between the two representations. It was equally comfortable and useful to use the same single-byte representation internally and externally. This comfort level decreases with more and larger character sets.

One of the problems to overcome with the internal representation is handling text that is externally encoded using different character sets. Assume a program that reads two texts and compares them using some metric. The comparison can be usefully done only if the texts are internally kept in a common format.

For such a common format (character set), 8 bits are certainly no longer enough. So the smallest entity will have to grow—*wide characters* will now be used. Instead of 1 byte per character, 2 or 4 will be used instead (three are not good to address in memory and using more than 4 bytes seems to be unnecessary).

As shown in Chapter 5 [String and Array Utilities], page 89, a completely new family has been created of functions that can handle wide-character texts in memory. The most commonly used character sets for such internal wide-character representations are Unicode and ISO 10646 (also known as UCS for Universal Character Set). Unicode was originally planned as a 16-bit character set, whereas ISO 10646 was designed to be a 31-bit large code space. The two standards are practically identical. They have the same character repertoire and code table, but Unicode specifies added semantics. At the moment, only characters in the first `0x10000` code positions (the so-called Basic Multilingual Plane or BMP) have been assigned, but the assignment of more specialized characters outside this 16-bit space is already in progress. A number of encodings have been defined for Unicode and

ISO 10646 characters: UCS-2 is a 16-bit word that can only represent characters from the BMP, UCS-4 is a 32-bit word than can represent any Unicode and ISO 10646 character, UTF-8 is an ASCII-compatible encoding where ASCII characters are represented by ASCII bytes and non-ASCII characters by sequences of 2-6 non-ASCII bytes, and UTF-16 is an extension of UCS-2 in which pairs of certain UCS-2 words can be used to encode non-BMP characters up to `0x10ffff`.

To represent wide characters, the `char` type is not suitable. For this reason, the ISO C standard introduces a new type that is designed to keep one character of a wide-character string. To maintain the similarity, there is also a type corresponding to `int` for those functions that take a single wide character.

wchar_t Data type

This data type is used as the base type for wide-character strings. In other words, arrays of objects of this type are the equivalent of `char[]` for multibyte-character strings. The type is defined in '`stddef.h`'.

The ISO C90 standard, where `wchar_t` was introduced, does not say anything specific about the representation. It only requires that this type be capable of storing all elements of the basic character set. Therefore, it would be legitimate to define `wchar_t` as `char`, which might make sense for embedded systems.

But for GNU systems, `wchar_t` is always 32 bits wide and therefore capable of representing all UCS-4 values, which means it is capable of covering all of ISO 10646. Some Unix systems define `wchar_t` as a 16-bit type and thereby follow Unicode very strictly. This definition is perfectly fine with the standard, but it also means that to represent all characters from Unicode and ISO 10646, you have to use UTF-16 surrogate characters, which is in fact a multi-wide-character encoding. But resorting to multi-wide-character encoding contradicts the purpose of the `wchar_t` type.

wint_t Data type

`wint_t` is a data type used for parameters and variables that contain a single wide character. As the name suggests, this type is the equivalent of `int` when using the normal `char` strings. The types `wchar_t` and `wint_t` often have the same representation if their size is 32 bits wide, but if `wchar_t` is defined as `char`, the type `wint_t` must be defined as `int` due to the parameter promotion.

This type is defined in '`wchar.h`' and was introduced in Amendment 1 to ISO C90.

As there are for the `char` data type, macros are available for specifying the minimum and maximum value representable in an object of type `wchar_t`.

`wint_t` **WCHAR_MIN** Macro

The macro `WCHAR_MIN` evaluates to the minimum value representable by an object of type `wint_t`.

This macro was introduced in Amendment 1 to ISO C90.

`wint_t` **WCHAR_MAX** Macro

The macro `WCHAR_MAX` evaluates to the maximum value representable by an object of type `wint_t`.

This macro was introduced in Amendment 1 to ISO C90.

Another special wide-character value is the equivalent to `EOF`.

`wint_t` **WEOF** Macro

The macro `WEOF` evaluates to a constant expression of type `wint_t`, whose value is different from any member of the extended character set.

`WEOF` need not be the same value as `EOF`, and unlike `EOF`, it also need *not* be negative. In other words, sloppy code like:

```
{
  int c;
  ...
  while ((c = getc (fp)) < 0)
    ...
}
```

has to be rewritten to use `WEOF` explicitly when wide characters are used:

```
{
  wint_t c;
  ...
  while ((c = wgetc (fp)) != WEOF)
    ...
}
```

This macro was introduced in Amendment 1 to ISO C90 and is defined in ‘`wchar.h`’.

These internal representations present problems when it comes to storing and transmittal. Because each single wide character consists of more than 1 byte, they are effected by byte-ordering. Thus, machines with different endianesses would see different values when accessing the same data. This byte-ordering concern also applies for communication protocols that are all byte-based and therefore require the sender to decide about splitting the wide character in bytes. A last point is that wide characters often require more storage space than a customized byte-oriented character set.

For all the above reasons, an external encoding that is different from the internal encoding is often used if the latter is UCS-2 or UCS-4. The external encoding is byte-based and can be chosen appropriately for the environment and for the texts to be handled. A variety of different character sets can be used for this external encoding (information that will not be exhaustively presented here—instead, a description of the major groups will suffice). All of the ASCII-based character sets fulfill one requirement: they are file-system safe. This means that the character `'/'` is used in the encoding *only* to represent itself. Things are a bit different for character sets like EBCDIC (Extended Binary Coded Decimal Interchange Code, a

character-set family used by IBM), but if the operation system does not understand EBCDIC directly, the parameters-to-system calls have to be converted first anyhow.

- The simplest character sets are single-byte character sets. There can only be up to 256 characters (for 8-bit character sets), which is not sufficient to cover all languages but might be sufficient to handle a specific text. Handling of an 8-bit character set is simple. This is not true for other kinds presented later, and therefore, the application you use might require 8-bit character sets.
- The ISO 2022 standard defines a mechanism for extended character sets where one character *can* be represented by more than one byte. This is achieved by associating a state with the text. Characters that can be used to change the state can be embedded in the text. Each byte in the text might have a different interpretation in each state. The state might even influence whether a given byte stands for a character on its own or whether it has to be combined with some more bytes.

  In most uses of ISO 2022, the defined character sets do not allow state changes that cover more than the next character. This has the big advantage that whenever you can identify the beginning of the byte sequence of a character, you can interpret a text correctly. Examples of character sets using this policy are the various EUC character sets (used by Sun's operations systems, EUC-JP, EUC-KR, EUC-TW, and EUC-CN) or Shift_JIS (SJIS, a Japanese encoding).

  But there are also character sets using a state that is valid for more than one character and has to be changed by another byte sequence. Examples for this are ISO-2022-JP, ISO-2022-KR and ISO-2022-CN.
- Early attempts to fix 8-bit character sets for other languages using the Roman alphabet lead to character sets like ISO 6937. Here, bytes representing characters like the acute accent do not produce output themselves—you have to combine them with other characters to get the desired result. For example, the byte sequence `0xc2 0x61` (nonspacing acute accent, followed by lowercase 'a') to get the "small a with acute" character. To get the acute accent character on its own, you have to write `0xc2 0x20` (the nonspacing acute followed by a space).

  Character sets like ISO 6937 are used in some embedded systems, such as teletex.
- Instead of converting the Unicode or ISO 10646 text used internally, it is often sufficient to simply use an encoding different than UCS-2/UCS-4. The Unicode and ISO 10646 standards even specify such an encoding: UTF-8. This encoding is able to represent all of ISO 10646 31 bits in a byte string of length 1 to 6.

  There were a few other attempts to encode ISO 10646, such as UTF-7, but UTF-8 is today the only encoding that should be used. In fact, with any luck, UTF-8 will soon be the only external encoding that has to be supported. It proves to be universally usable and its only disadvantage is that it favors Roman languages by making the byte string representation of other scripts (Cyrillic, Greek, Asian scripts) longer than necessary if using a specific character set

for these scripts. Methods like the Unicode compression scheme can alleviate these problems.

The question remaining is how to select the character set or encoding to use. You cannot decide yourself—it is decided by the developers of the system or the majority of the users. Since the goal is interoperability, you have to use whatever the other people you work with use. If there are no constraints, the selection is based on the requirements the expected circle of users will have. In other words, if a project is expected to be used in only Russia, it is fine to use KOI8-R or a similar character set. But if at the same time people from Greece are participating, you should use a character set that allows all people to collaborate.

The most widely useful solution seems to be to go with the most general character set, ISO 10646. Use UTF-8 as the external encoding and problems with users not being able to use their own language adequately are a thing of the past.

One final comment about the choice of the wide-character representation is necessary at this point. We have said above that the natural choice is using Unicode or ISO 10646. This is not required but is encouraged by the ISO C standard. The standard defines at least a macro `__STDC_ISO_10646__` that is only defined on systems where the `wchar_t` type encodes ISO 10646 characters. If this symbol is not defined, you should avoid making assumptions about the wide-character representation. If the programmer uses only the functions provided by the C library to handle wide-character strings, there should be no compatibility problems with other systems.

6.2 Overview About Character-Handling Functions

A Unix C library contains three different sets of functions in two families to handle character-set conversion. One of the function families (the most commonly used) is specified in the ISO C90 standard and, therefore, is portable even beyond the Unix world. Unfortunately, this family is the least useful one. These functions should be avoided whenever possible, especially when developing libraries (as opposed to applications).

The second family of functions was introduced in the early Unix standards (XPG2) and is still part of the latest and greatest Unix standard, Unix 98. It is also the most powerful and useful set of functions. But we will start with the functions defined in Amendment 1 to ISO C90.

6.3 Restartable Multibyte Conversion Functions

The ISO C standard defines functions to convert strings from a multibyte representation to wide-character strings. There are a number of peculiarities:

- The character set assumed for the multibyte encoding is not specified as an argument to the functions. Instead, the character set specified by the `LC_CTYPE` category of the current locale is used (see Section 7.3 [Categories of Activities That Locales Affect], page 182).

- The functions handling more than one character at a time require NUL terminated strings as the argument (i.e., converting blocks of text does not work unless one can add a NUL byte at an appropriate place). The GNU C Library contains some extensions to the standard that allow specifying a size, but basically they also expect terminated strings.

Despite these limitations, the ISO C functions can be used in many contexts. In graphical user interfaces, for instance, it is not uncommon to have functions that require text to be displayed in a wide-character string if the text is not simple ASCII. The text itself might come from a file with translations, and the user should decide about the current locale, which determines the translation and therefore also the external encoding used. In such a situation (and many others), the functions described here are perfect. If more freedom while performing the conversion is necessary, take a look at the `iconv` functions (see Section 6.5 [Generic Charset Conversion], page 157).

6.3.1 Selecting the Conversion and Its Properties

We already said above that the currently selected locale for the `LC_CTYPE` category decides about the conversion that is performed by the functions we are about to describe. Each locale uses its own character set (given as an argument to `localedef`) and this is the one assumed as the external multibyte encoding. The wide-character character set always is UCS-4, at least on GNU systems.

A characteristic of each multibyte-character set is the maximum number of bytes that can be necessary to represent one character. This information is quite important when writing code that uses the conversion functions (as shown in the examples below). The ISO C standard defines two macros that provide this information.

`int` **MB_LEN_MAX** Macro

`MB_LEN_MAX` specifies the maximum number of bytes in the multibyte sequence for a single character in any of the supported locales. It is a compile-time constant and is defined in ‘`limits.h`’.

`int` **MB_CUR_MAX** Macro

`MB_CUR_MAX` expands into a positive integer expression that is the maximum number of bytes in a multibyte character in the current locale. The value is never greater than `MB_LEN_MAX`. Unlike `MB_LEN_MAX`, this macro need not be a compile-time constant, and in the GNU C Library it is not.

`MB_CUR_MAX` is defined in ‘`stdlib.h`’.

Two different macros are necessary, since strictly ISO C90 compilers do not allow variable-length array definitions, but still it is desirable to avoid dynamic allocation. This incomplete piece of code shows the problem:

```
{
  char buf[MB_LEN_MAX];
  ssize_t len = 0;
```

```
  while (! feof (fp))
    {
      fread (&buf[len], 1, MB_CUR_MAX - len, fp);
      /* ... process buf */
      len -= used;
    }
}
```

The code in the inner loop is expected to always have enough bytes in the array *buf* to convert one multibyte character. The array *buf* has to be sized statically since many compilers do not allow a variable size. The `fread` call makes sure that `MB_CUR_MAX` bytes are always available in *buf*. It isn't a problem if `MB_CUR_MAX` is not a compile-time constant.

6.3.2 Representing the State of the Conversion

In the introduction of this chapter, it was said that certain character sets use a *stateful* encoding. That is, the encoded values depend in some way on the previous bytes in the text.

Since the conversion functions allow converting a text in more than one step, we must have a way to pass this information from one call of the functions to another.

mbstate_t Data type

A variable of type `mbstate_t` can contain all the information about the *shift state* needed from one call to a conversion function to another.

`mbstate_t` is defined in '`wchar.h`'. It was introduced in Amendment 1 to ISO C90.

To use objects of type `mbstate_t`, the programmer has to define such objects (normally as local variables on the stack) and pass a pointer to the object to the conversion functions. This way, the conversion function can update the object if the current multibyte-character set is stateful.

There is no specific function or initializer to put the state object in any specific state. The rules are that the object should always represent the initial state before the first use, and this is achieved by clearing the whole variable with code such as:

```
{
  mbstate_t state;
  memset (&state, '\0', sizeof (state));
  /* from now on state can be used.  */
  ...
}
```

When using the conversion functions to generate output, it is often necessary to test whether the current state corresponds to the initial state. This is necessary, for example, to decide whether to emit escape sequences to set the state to the initial state at certain sequence points. Communication protocols often require this.

`int` **mbsinit** (`const mbstate_t *`*ps*) Function

The `mbsinit` function determines whether the state object pointed to by *ps* is in the initial state. If *ps* is a null pointer or the object is in the initial state, the return value is nonzero. Otherwise it is zero.

`mbsinit` was introduced in Amendment 1 to ISO C90 and is declared in 'wchar.h'.

Code using `mbsinit` often looks similar to this:

```
{
  mbstate_t state;
  memset (&state, '\0', sizeof (state));
  /* Use state.  */
  ...
  if (! mbsinit (&state))
    {
      /* Emit code to return to initial state.  */
      const wchar_t empty[] = L"";
      const wchar_t *srcp = empty;
      wcsrtombs (outbuf, &srcp, outbuflen, &state);
    }
  ...
}
```

The code to emit the escape sequence to get back to the initial state is interesting. The `wcsrtombs` function can be used to determine the necessary output code (see Section 6.3.4 [Converting Multibyte- and Wide-Character Strings], page 147). On GNU systems it is not necessary to perform this extra action for the conversion from multibyte text to wide-character text since the wide-character encoding is not stateful. But there is nothing mentioned in any standard that prohibits making `wchar_t` using a stateful encoding.

6.3.3 Converting Single Characters

The most fundamental of the conversion functions are those dealing with single characters. This does not always mean single bytes. But since there is very often a subset of the multibyte-character set that consists of single-byte sequences, there are functions to help with converting bytes. Frequently, ASCII is a subpart of the multibyte-character set. In such a scenario, each ASCII character stands for itself, and all other characters have at least a first byte that is beyond the range 0 to 127.

`wint_t` **btowc** (`int` *c*) Function

The `btowc` function ("byte to wide character") converts a valid single-byte character *c* in the initial shift state into the wide-character equivalent using the conversion rules from the currently selected locale of the `LC_CTYPE` category.

If `(unsigned char)` *c* is not a valid single-byte multibyte character or if *c* is `EOF`, the function returns `WEOF`.

Please note the restriction of *c* being tested for validity only in the initial shift state. No `mbstate_t` object is used from which the state information is taken, and the function also does not use any static state.

The `btowc` function was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

Despite the limitation that the single-byte value is always interpreted in the initial state, this function is actually useful most of the time. Most characters are either entirely single-byte character sets or extensions to ASCII. But then, it is possible to write code like this (not that this specific example is very useful):

```
wchar_t *
itow (unsigned long int val)
{
  static wchar_t buf[30];
  wchar_t *wcp = &buf[29];
  *wcp = L'\0';
  while (val != 0)
    {
      *--wcp = btowc ('0' + val % 10);
      val /= 10;
    }
  if (wcp == &buf[29])
    *--wcp = L'0';
  return wcp;
}
```

Why is it necessary to use such a complicated implementation and not simply cast `'0' + val % 10` to a wide character? The answer is that there is no guarantee that one can perform this kind of arithmetic on the character of the character set used for `wchar_t` representation. In other situations, the bytes are not constant at compile time and so the compiler cannot do the work. In situations like this, it is necessary to use `btowc`.

There also is a function for the conversion in the other direction.

`int` **wctob** (`wint_t` *c*) Function

The `wctob` function ("wide character to byte") takes as the parameter a valid wide character. If the multibyte representation for this character in the initial state is exactly 1 byte long, the return value of this function is this character. Otherwise, the return value is `EOF`.

`wctob` was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

There are more general functions to convert single characters from multibyte representations to wide characters and vice versa. These functions pose no limit on the length of the multibyte representation, and they also do not require it to be in the initial state.

`size_t` **mbrtowc** (`wchar_t *restrict` *pwc*, `const char *restrict` *s*, `size_t` *n*, `mbstate_t *restrict` *ps*) Function

The `mbrtowc` function ("multibyte restartable to wide character") converts the next multibyte character in the string pointed to by *s* into a wide character and stores it in the wide-character string pointed to by *pwc*. The conversion is performed according to the locale currently selected for the `LC_CTYPE` category. If the conversion for the character set used in the locale requires a state, the multibyte string is interpreted in the state represented by the object pointed to by *ps*. If *ps* is a null pointer, a static, internal state variable used only by the `mbrtowc` function is used.

If the next multibyte character corresponds to the NUL wide character, the return value of the function is 0 and the state object is afterwards in the initial state. If the next *n* or fewer bytes form a correct multibyte character, the return value is the number of bytes starting from *s* that form the multibyte character. The conversion state is updated according to the bytes consumed in the conversion. In both cases the wide character (either the `L'\0'` or the one found in the conversion) is stored in the string pointed to by *pwc* if *pwc* is not null.

If the first *n* bytes of the multibyte string possibly form a valid multibyte character but there are more than *n* bytes needed to complete it, the return value of the function is `(size_t) -2` and no value is stored. This can happen even if *n* has a value greater than or equal to `MB_CUR_MAX` since the input might contain redundant shift sequences.

If the first `n` bytes of the multibyte string cannot possibly form a valid multibyte character, no value is stored, the global variable `errno` is set to the value `EILSEQ`, and the function returns `(size_t) -1`. The conversion state is afterwards undefined.

`mbrtowc` was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

Use of `mbrtowc` is straightforward. A function that copies a multibyte string into a wide-character string while at the same time converting all lowercase characters into uppercase could look like this (this is not the final version, just an example—it has no error checking, and sometimes leaks memory):

```
wchar_t *
mbstouwcs (const char *s)
{
  size_t len = strlen (s);
  wchar_t *result = malloc ((len + 1) * sizeof (wchar_t));
  wchar_t *wcp = result;
  wchar_t tmp[1];
  mbstate_t state;
  size_t nbytes;

  memset (&state, '\0', sizeof (state));
  while ((nbytes = mbrtowc (tmp, s, len, &state)) > 0)
```

```
      {
        if (nbytes >= (size_t) -2)
          /* Invalid input string.  */
          return NULL;
        *wcp++ = towupper (tmp[0]);
        len -= nbytes;
        s += nbytes;
      }
    return result;
  }
```

The use of `mbrtowc` should be clear. A single wide character is stored in *tmp*`[0]`, and the number of consumed bytes is stored in the variable *nbytes*. If the conversion is successful, the uppercase variant of the wide character is stored in the *result* array and the pointer to the input string and the number of available bytes is adjusted.

The only non-obvious thing about `mbrtowc` might be the way memory is allocated for the result. The above code uses the fact that there can never be more wide characters in the converted results than there are bytes in the multibyte input string. This method yields a pessimistic guess about the size of the result, and if many wide-character strings have to be constructed this way or if the strings are long, the extra memory allocation required because the input string contains multibyte characters might be significant. The allocated memory block can be resized to the correct size before returning it, but a better solution might be to allocate just the right amount of space for the result right away. Unfortunately, there is no function to compute the length of the wide-character string directly from the multibyte string. There is, however, a function that does part of the work.

`size_t` **mbrlen** (`const char *restrict` *s*`, size_t` *n*`,` `mbstate_t *`*ps*) Function

The `mbrlen` function ("multibyte restartable length") computes the number of at most *n* bytes starting at *s*, which form the next valid and complete multibyte character.

If the next multibyte character corresponds to the NUL wide character, the return value is 0. If the next *n* bytes form a valid multibyte character, the number of bytes belonging to this multibyte-character byte sequence is returned.

If the first *n* bytes possibly form a valid multibyte character but the character is incomplete, the return value is `(size_t) -2`. Otherwise, the multibyte-character sequence is invalid and the return value is `(size_t) -1`.

The multibyte sequence is interpreted in the state represented by the object pointed to by *ps*. If *ps* is a null pointer, a state object local to `mbrlen` is used.

`mbrlen` was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

The attentive reader now will note that `mbrlen` can be implemented as:

```
mbrtowc (NULL, s, n, ps != NULL ? ps : &internal)
```

This is true and in fact is mentioned in the official specification. How can this function be used to determine the length of the wide-character string created from a multibyte-character string? It is not directly usable, but we can define a function `mbslen` using it:

```
size_t
mbslen (const char *s)
{
  mbstate_t state;
  size_t result = 0;
  size_t nbytes;
  memset (&state, '\0', sizeof (state));
  while ((nbytes = mbrlen (s, MB_LEN_MAX, &state)) > 0)
    {
      if (nbytes >= (size_t) -2)
        /* Something is wrong.  */
        return (size_t) -1;
      s += nbytes;
      ++result;
    }
  return result;
}
```

This function simply calls `mbrlen` for each multibyte character in the string and counts the number of function calls. We here use `MB_LEN_MAX` as the size argument in the `mbrlen` call. This is acceptable since a) this value is larger than the length of the longest multibyte-character sequence and b) we know that the string *s* ends with a NUL byte, which cannot be part of any other multibyte-character sequence but the one representing the NUL wide character. Therefore, the `mbrlen` function will never read invalid memory.

Now that this function is available (just to make this clear, this function is *not* part of the GNU C Library), we can compute the number of wide characters required to store the converted multibyte-character string *s* using:

```
wcs_bytes = (mbslen (s) + 1) * sizeof (wchar_t);
```

Please note that the `mbslen` function is quite inefficient. The implementation of `mbstouwcs` with `mbslen` would have to perform the conversion of the multibyte-character input string twice, and this conversion might be quite expensive. So it is necessary to think about the consequences of using the easier but imprecise method before doing the work twice.

`size_t` **wcrtomb** (`char *restrict` *s*, `wchar_t` *wc*, `mbstate_t *restrict` *ps*) Function

The `wcrtomb` function ("wide character restartable to multibyte") converts a single wide character into a multibyte string corresponding to that wide character.

If *s* is a null pointer, the function resets the state stored in the objects pointed to by *ps* (or the internal `mbstate_t` object) to the initial state. This can also be achieved by a call like this:

```
wcrtombs (temp_buf, L'\0', ps)
```

since, if *s* is a null pointer, `wcrtomb` performs as if it writes into an internal buffer, which is guaranteed to be large enough.

If *wc* is the NUL wide character, `wcrtomb` emits, if necessary, a shift sequence to get the state *ps* into the initial state followed by a single NUL byte, which is stored in the string *s*.

Otherwise, a byte sequence (possibly including shift sequences) is written into the string *s*. This only happens if *wc* is a valid wide character (i.e., it has a multibyte representation in the character set selected by locale of the `LC_CTYPE` category). If *wc* is not a valid wide character, nothing is stored in the string *s*, `errno` is set to `EILSEQ`, the conversion state in *ps* is undefined, and the return value is `(size_t) -1`.

If no error occurred, the function returns the number of bytes stored in the string *s*. This includes all bytes representing shift sequences.

One word about the interface of the function: there is no parameter specifying the length of the array *s*. Instead, the function assumes that there are at least `MB_CUR_MAX` bytes available since this is the maximum length of any byte sequence representing a single character. So the caller has to make sure that there is enough space available; otherwise, buffer overruns can occur.

`wcrtomb` was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

Using `wcrtomb` is as easy as using `mbrtowc`. The following example appends a wide-character string to a multibyte-character string. Again, the code is not really useful (or correct) but is simply here to demonstrate use and some problems.

```
char *
mbscatwcs (char *s, size_t len, const wchar_t *ws)
{
  mbstate_t state;
  /* Find the end of the existing string.  */
  char *wp = strchr (s, '\0');
  len -= wp - s;
  memset (&state, '\0', sizeof (state));
  do
    {
      size_t nbytes;
      if (len < MB_CUR_LEN)
        {
          /* We cannot guarantee that the next
             character fits into the buffer, so
             return an error.  */
```

```
            errno = E2BIG;
            return NULL;
          }
        nbytes = wcrtomb (wp, *ws, &state);
        if (nbytes == (size_t) -1)
          /* Error in the conversion.  */
          return NULL;
        len -= nbytes;
        wp += nbytes;
      }
    while (*ws++ != L'\0');
    return s;
  }
```

First the function has to find the end of the string currently in the array *s*. The `strchr` call does this very efficiently, since a requirement for multibyte-character representations is that the NUL byte never be used except to represent itself (and in this context, the end of the string).

After initializing the state object, the loop is entered where the first task is to make sure there is enough room in the array *s*. We abort if there are not at least `MB_CUR_LEN` bytes available. This is not always optimal, but we have no other choice. We might have less than `MB_CUR_LEN` bytes available, but the next multibyte character might also be only 1 byte long. At the time the `wcrtomb` call returns, it is too late to decide whether the buffer was large enough. If this solution is unsuitable, there is a very slow but more accurate solution:

```
...
if (len < MB_CUR_LEN)
  {
    mbstate_t temp_state;
    memcpy (&temp_state, &state, sizeof (state));
    if (wcrtomb (NULL, *ws, &temp_state) > len)
      {
        /* We cannot guarantee that the next
           character fits into the buffer, so
           return an error.  */
        errno = E2BIG;
        return NULL;
      }
  }
...
```

Here we perform the conversion that might overflow the buffer so that we are afterwards in the position to make an exact decision about the buffer size. Please note the `NULL` argument for the destination buffer in the new `wcrtomb` call; a writeable string would usually be used, but we've used `NULL` since we're not interested in the converted text at this point. The most unusual thing about this piece

of code certainly is the duplication of the conversion-state object, but if a change of the state is necessary to emit the next multibyte character, we want to have the same shift-state change performed in the real conversion. Therefore, we have to preserve the initial shift-state information.

There are certainly many more and better solutions to this problem. This example is only provided for educational purposes.

6.3.4 Converting Multibyte- and Wide-Character Strings

The functions described in the previous section only convert a single character at a time. Most operations to be performed in real-world programs include strings and therefore the ISO C standard also defines conversions on entire strings. However, the defined set of functions is quite limited; therefore, the GNU C Library contains a few extensions that can help in some important situations.

`size_t` **mbsrtowcs** (`wchar_t *restrict` *dst*, `const char **restrict` *src*, `size_t` *len*, `mbstate_t *restrict` *ps*) Function

The `mbsrtowcs` function ("multibyte string restartable to wide-character string") converts a NUL-terminated multibyte-character string at **src* into an equivalent wide-character string, including the NUL wide character at the end. The conversion is started using the state information from the object pointed to by *ps* or from an internal object of `mbsrtowcs` if *ps* is a null pointer. Before returning, the state object is updated to match the state after the last converted character. The state is the initial state if the terminating NUL byte is reached and converted.

If *dst* is not a null pointer, the result is stored in the array pointed to by *dst*; otherwise, the conversion result is not available since it is stored in an internal buffer.

If *len* wide characters are stored in the array *dst* before reaching the end of the input string, the conversion stops and *len* is returned. If *dst* is a null pointer, *len* is never checked.

Another reason for a premature return from the function call is if the input string contains an invalid multibyte-sequence. In this case the global variable `errno` is set to `EILSEQ` and the function returns `(size_t) -1`.

In all other cases, the function returns the number of wide characters converted during this call. If *dst* is not null, `mbsrtowcs` stores in the pointer pointed to by *src* either a null pointer (if the NUL byte in the input string was reached) or the address of the byte following the last converted multibyte character.

`mbsrtowcs` was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

The definition of the `mbsrtowcs` function has one important limitation. The requirement that *dst* has to be a NUL-terminated string causes problems if you

want to convert buffers with text. A buffer is normally not a collection of NUL-terminated strings but instead a continuous collection of lines, separated by newline characters. Now assume that a function to convert one line from a buffer is needed. Since the line is not NUL-terminated, the source pointer cannot directly point into the unmodified text buffer. This means that either you insert the NUL byte at the appropriate place for the time of the `mbsrtowcs` function call (which is not doable for a read-only buffer or in a multithreaded application) or you copy the line in an extra buffer where it can be terminated by a NUL byte. It is not in general possible to limit the number of characters to convert by setting the parameter *len* to any specific value. Since it is not known how many bytes each multibyte-character sequence is in length, you can only guess.

There is still a problem with the method of NUL-terminating a line right after the newline character, which could lead to very strange results. As said in the description of the `mbsrtowcs` function above, the conversion state is guaranteed to be in the initial shift state after processing the NUL byte at the end of the input string. But this NUL byte is not really part of the text (i.e., the conversion state after the newline in the original text could be something different than the initial shift state and therefore the first character of the next line is encoded using this state). But the state in question is never accessible to the user since the conversion stops after the NUL byte (which resets the state). Most stateful character sets in use today require that the shift state after a newline be the initial state—but this is not a strict guarantee. Therefore, simply NUL-terminating a piece of a running text is not always an adequate solution and, therefore, should never be done in generally used code.

The generic conversion interface (see Section 6.5 [Generic Charset Conversion], page 157) does not have this limitation (it simply works on buffers, not strings), and the GNU C Library contains a set of functions that take additional parameters specifying the maximum number of bytes that are consumed from the input string. This way the problem of `mbsrtowcs`'s example above could be solved by determining the line length and passing this length to the function.

`size_t` **wcsrtombs** (`char *restrict` *dst*, `const wchar_t **restrict` *src*, `size_t` *len*, `mbstate_t *restrict` *ps*) — Function

The `wcsrtombs` function ("wide-character string restartable to multibyte string") converts the NUL-terminated wide-character string at **src* into an equivalent multibyte-character string and stores the result in the array pointed to by *dst*. The NUL wide character is also converted. The conversion starts in the state described in the object pointed to by *ps* or by a state object locally to `wcsrtombs` in case *ps* is a null pointer. If *dst* is a null pointer, the conversion is performed as usual, but the result is not available. If all characters of the input string were successfully converted and if *dst* is not a null pointer, the pointer pointed to by *src* gets assigned a null pointer.

If one of the wide characters in the input string has no valid multibyte-character equivalent, the conversion stops early, sets the global variable `errno` to `EILSEQ`, and returns `(size_t) -1`.

Another reason for a premature stop is if *dst* is not a null pointer and the next converted character would require more than *len* bytes in total to the array *dst*. In this case (and if *dest* is not a null pointer), the pointer pointed to by *src* is assigned a value pointing to the wide character right after the last successfully converted one.

Except in the case of an encoding error, the return value of the `wcsrtombs` function is the number of bytes in all the multibyte-character sequences stored in *dst*. Before returning, the state in the object pointed to by *ps* (or the internal object in case *ps* is a null pointer) is updated to reflect the state after the last conversion. The state is the initial shift state in case the terminating NUL wide character was converted.

The `wcsrtombs` function was introduced in Amendment 1 to ISO C90 and is declared in '`wchar.h`'.

The restriction mentioned above for the `mbsrtowcs` function applies here also. There is no possibility of directly controlling the number of input characters. You have to place the NUL wide character at the correct place, or control the consumed input indirectly via the available output array size (the *len* parameter).

`size_t` **mbsnrtowcs** (`wchar_t *restrict` *dst*, `const char **restrict` *src*, `size_t` *nmc*, `size_t` *len*, `mbstate_t *restrict` *ps*) Function

The `mbsnrtowcs` function is very similar to the `mbsrtowcs` function. All the parameters are the same except for *nmc*, which is new. The return value is the same as for `mbsrtowcs`.

This new parameter specifies how many bytes at most can be used from the multibyte-character string. In other words, the multibyte-character string **src* need not be NUL-terminated. But if a NUL byte is found within the *nmc* first bytes of the string, the conversion stops there.

This function is a GNU extension. It is meant to work around the problems mentioned above. Now it is possible to convert a buffer with multibyte-character text piece-for-piece without having to care about inserting NUL bytes and the effect of NUL bytes on the conversion state.

A function to convert a multibyte string into a wide-character string and display it could be written like this (this example is useful only to show the syntax):

```
void
showmbs (const char *src, FILE *fp)
{
  mbstate_t state;
  int cnt = 0;
  memset (&state, '\0', sizeof (state));
```

```
      while (1)
        {
          wchar_t linebuf[100];
          const char *endp = strchr (src, '\n');
          size_t n;

          /* Exit if there is no more line.  */
          if (endp == NULL)
            break;

          n = mbsnrtowcs (linebuf, &src, endp - src, 99, &state);
          linebuf[n] = L'\0';
          fprintf (fp, "line %d: \"%S\"\n", linebuf);
        }
    }
```

There is no problem with the state after a call to `mbsnrtowcs`. Since we don't insert characters in the strings that were not there from the beginning, and we use *state* only for the conversion of the given buffer, there is no problem with altering the state.

`size_t` **wcsnrtombs** (`char *restrict` *dst*, `const wchar_t **restrict` *src*, `size_t` *nwc*, `size_t` *len*, `mbstate_t *restrict` *ps*) — Function

The `wcsnrtombs` function implements the conversion from wide-character strings to multibyte-character strings. It is similar to `wcsrtombs`, but like `mbsnrtowcs`, it takes an extra parameter, which specifies the length of the input string.

No more than *nwc* wide characters from the input string **src* are converted. If the input string contains a NUL wide character in the first *nwc* characters, the conversion stops at this place.

The `wcsnrtombs` function is a GNU extension and like `mbsnrtowcs` helps in situations where no NUL-terminated input strings are available.

6.3.5 A Complete Multibyte Conversion Example

The example programs given in the last sections are only brief and do not contain things like error checking. Presented here is a complete and documented example. It features the `mbrtowc` function, but it should be easy to derive versions using the other functions:

```
int
file_mbsrtowcs (int input, int output)
{
  /* Note the use of MB_LEN_MAX.
     MB_CUR_MAX cannot portably be used here.  */
```

```
char buffer[BUFSIZ + MB_LEN_MAX];
mbstate_t state;
int filled = 0;
int eof = 0;

/* Initialize the state.  */
memset (&state, '\0', sizeof (state));

while (!eof)
  {
    ssize_t nread;
    ssize_t nwrite;
    char *inp = buffer;
    wchar_t outbuf[BUFSIZ];
    wchar_t *outp = outbuf;

    /* Fill up the buffer from the input file.  */
    nread = read (input, buffer + filled, BUFSIZ);
    if (nread < 0)
      {
        perror ("read");
        return 0;
      }
    /* If we reach end of file, make a note to read no more.  */
    if (nread == 0)
      eof = 1;

    /* filled is now the number of bytes in buffer.  */
    filled += nread;

    /* Convert those bytes to wide characters—as many as we can.  */
    while (1)
      {
        size_t thislen = mbrtowc (outp, inp, filled, &state);
        /* Stop converting at invalid character;
           this can mean we have read just the first part
           of a valid character.  */
        if (thislen == (size_t) -1)
          break;
        /* We want to handle embedded NUL bytes
           but the return value is 0.  Correct this.  */
        if (thislen == 0)
          thislen = 1;
        /* Advance past this character.  */
```

```
            inp += thislen;
            filled -= thislen;
            ++outp;
          }

        /* Write the wide characters we just made.  */
        nwrite = write (output, outbuf,
                        (outp - outbuf) * sizeof (wchar_t));
        if (nwrite < 0)
          {
            perror ("write");
            return 0;
          }

        /* See if we have a real invalid character.  */
        if ((eof && filled > 0) || filled >= MB_CUR_MAX)
          {
            error (0, 0, "invalid multibyte character");
            return 0;
          }

        /* If any characters must be carried forward,
           put them at the beginning of buffer.  */
        if (filled > 0)
          memmove (inp, buffer, filled);
      }

    return 1;
  }
```

6.4 Nonreentrant Conversion Function

The functions described in the previous chapter are defined in Amendment 1 to ISO C90, but the original ISO C90 standard also contained functions for character-set conversion. The reason that these original functions are not described first is that they are almost entirely useless.

The problem is that all the conversion functions described in the original ISO C90 use a local state. Using a local state implies that multiple conversions at the same time (not only when using threads) cannot be done, and that you cannot first convert single characters and then strings since you cannot tell the conversion functions which state to use.

These original functions are therefore usable only in a very limited set of situations. You have to finish converting the entire string before starting a new one, and each string or text must be converted with the same function (there is no problem

with the library itself; it is guaranteed that no library function changes the state of any of these functions). *For the above reasons, it is strongly recommended that the functions described in the previous section be used in place of nonreentrant conversion functions.*

6.4.1 Nonreentrant Conversion of Single Characters

`int` **mbtowc** (`wchar_t *restrict` *result*`, const char *restrict` *string*`, size_t` *size*) Function

The `mbtowc` ("multibyte to wide character") function, when called with nonnull *string*, converts the first multibyte character beginning at *string* to its corresponding wide-character code. It stores the result in **result*.

`mbtowc` never examines more than *size* bytes. (The idea is to supply for *size* the number of bytes of data you have in hand.)

`mbtowc` with nonnull *string* distinguishes three possibilities: either the first *size* bytes at *string* start with valid multibyte characters, they start with an invalid byte-sequence or just part of a character, or *string* points to an empty string (a null character).

For a valid multibyte character, `mbtowc` converts it to a wide character and stores that in **result*, and returns the number of bytes in that character (always at least 1 and never more than *size*).

For an invalid byte-sequence, `mbtowc` returns -1. For an empty string, it returns 0, also storing `'\0'` in **result*.

If the multibyte-character code uses shift characters, then `mbtowc` maintains and updates a shift state as it scans. If you call `mbtowc` with a null pointer for *string*, that initializes the shift state to its standard initial value. It also returns nonzero if the multibyte-character code in use actually has a shift state (see Section 6.4.3 [States in Nonreentrant Functions], page 155).

`int` **wctomb** (`char *`*string*`, wchar_t` *wchar*) Function

The `wctomb` ("wide character to multibyte") function converts the wide-character code *wchar* to its corresponding multibyte-character sequence, and stores the result in bytes starting at *string*. At most `MB_CUR_MAX` characters are stored.

`wctomb` with nonnull *string* distinguishes three possibilities for *wchar*: a valid wide-character code (one that can be translated to a multibyte character), an invalid code, and `L'\0'`.

Given a valid code, `wctomb` converts it to a multibyte character, storing the bytes starting at *string*. Then it returns the number of bytes in that character (always at least 1 and never more than `MB_CUR_MAX`).

If *wchar* is an invalid wide-character code, `wctomb` returns -1. If *wchar* is `L'\0'`, it returns `0`, also storing `'\0'` in **string*.

If the multibyte-character code uses shift characters, then `wctomb` maintains and updates a shift state as it scans. If you call `wctomb` with a null pointer for

string, that initializes the shift state to its standard initial value. It also returns nonzero if the multibyte-character code in use actually has a shift state (see Section 6.4.3 [States in Nonreentrant Functions], page 155).

Calling this function with a *wchar* argument of zero when *string* is not null has the side effect of reinitializing the stored shift state *as well as* storing the multibyte character `'\0'` and returning 0.

Similar to `mbrlen`, there is also a nonreentrant function that computes the length of a multibyte character. It can be defined in terms of `mbtowc`.

`int` **mblen** (`const char *`*string*`, size_t` *size*) Function

The `mblen` function with a nonnull *string* argument returns the number of bytes that make up the multibyte character beginning at *string*, never examining more than *size* bytes. (The idea is to supply for *size* the number of bytes of data you have in hand.)

The return value of `mblen` distinguishes three possibilities: either the first *size* bytes at *string* start with valid multibyte characters, they start with an invalid byte-sequence or just part of a character, or *string* points to an empty string (a null character).

For a valid multibyte character, `mblen` returns the number of bytes in that character (always at least `1` and never more than *size*). For an invalid byte-sequence, `mblen` returns -1. For an empty string, it returns 0.

If the multibyte-character code uses shift characters, then `mblen` maintains and updates a shift state as it scans. If you call `mblen` with a null pointer for *string*, that initializes the shift state to its standard initial value. It also returns a nonzero value if the multibyte-character code in use actually has a shift state (see Section 6.4.3 [States in Nonreentrant Functions], page 155).

The function `mblen` is declared in '`stdlib.h`'.

6.4.2 Nonreentrant Conversion of Strings

For convenience, the ISO C90 standard also defines functions to convert entire strings instead of single characters. These functions suffer from the same problems as their reentrant counterparts from Amendment 1 to ISO C90 (see Section 6.3.4 [Converting Multibyte- and Wide-Character Strings], page 147).

`size_t` **mbstowcs** (`wchar_t *`*wstring*`, const char *`*string*`, size_t` *size*) Function

The `mbstowcs` ("multibyte string to wide-character string") function converts the null-terminated string of multibyte characters *string* to an array of wide-character codes, storing not more than *size* wide characters into the array beginning at *wstring*. The terminating null character counts toward the size, so if *size* is less than the actual number of wide characters resulting from *string*, no terminating null character is stored.

The conversion of characters from *string* begins in the initial shift state.

If an invalid multibyte-character sequence is found, the `mbstowcs` function returns a value of -1. Otherwise, it returns the number of wide characters stored in the array *wstring*. This number does not include the terminating null character, which is present if the number is less than *size*.

Here is an example showing how to convert a string of multibyte characters, allocating enough space for the result:

```
wchar_t *
mbstowcs_alloc (const char *string)
{
  size_t size = strlen (string) + 1;
  wchar_t *buf = xmalloc (size * sizeof (wchar_t));

  size = mbstowcs (buf, string, size);
  if (size == (size_t) -1)
    return NULL;
  buf = xrealloc (buf, (size + 1) * sizeof (wchar_t));
  return buf;
}
```

`size_t` **wcstombs** (`char *`*string*`, const wchar_t *`*wstring*`, size_t` *size*) Function

The `wcstombs` ("wide-character string to multibyte string") function converts the null-terminated wide-character array *wstring* into a string containing multibyte characters, storing not more than *size* bytes starting at *string*, followed by a terminating null character if there is room. The conversion of characters begins in the initial shift state.

The terminating null character counts toward the size, so if *size* is less than or equal to the number of bytes needed in *wstring*, no terminating null character is stored.

If a code that does not correspond to a valid multibyte character is found, the `wcstombs` function returns a value of -1. Otherwise, the return value is the number of bytes stored in the array *string*. This number does not include the terminating null character, which is present if the number is less than *size*.

6.4.3 States in Nonreentrant Functions

In some multibyte-character codes, the *meaning* of any particular byte sequence is not fixed; it depends on what other sequences have come earlier in the same string. Typically there are just a few sequences that can change the meaning of other sequences; these few are called *shift sequences* and we say that they set the *shift state* for other sequences that follow.

To illustrate shift state and shift sequences, suppose we decide that the sequence `0200` (just one byte) enters Japanese mode, in which pairs of bytes in the range `0240` to `0377` are single characters, while `0201` enters Latin-1 mode, in which

single bytes in the range `0240` to `0377` are characters, and interpreted according to the ISO Latin-1 character set. This is a multibyte code that has two alternative shift states ("Japanese mode" and "Latin-1 mode"), and two shift sequences that specify particular shift states.

When the multibyte-character code in use has shift states, then `mblen`, `mbtowc`, and `wctomb` must maintain and update the current shift state as they scan the string. To make this work properly, you must follow these rules:

- Before starting to scan a string, call the function with a null pointer for the multibyte-character address—for example, `mblen (NULL, 0)`. This initializes the shift state to its standard initial value.
- Scan the string one character at a time, in order. Do not "back up" and rescan characters already scanned, and do not intersperse the processing of different strings.

Here is an example of using `mblen` following these rules:

```
void
scan_string (char *s)
{
  int length = strlen (s);

  /* Initialize shift state.  */
  mblen (NULL, 0);

  while (1)
    {
      int thischar = mblen (s, length);
      /* Deal with end of string and invalid characters.  */
      if (thischar == 0)
        break;
      if (thischar == -1)
        {
          error ("invalid multibyte character");
          break;
        }
      /* Advance past this character.  */
      s += thischar;
      length -= thischar;
    }
}
```

The functions `mblen`, `mbtowc` and `wctomb` are not reentrant when using a multibyte code that uses a shift state. However, no other library functions call these functions, so you don't have to worry that the shift state might be mysteriously changed.

6.5 Generic Charset Conversion

The conversion functions mentioned so far in this chapter all have in common that they operate on character sets that are not directly specified by the functions. The multibyte encoding used is specified by the currently selected locale for the `LC_CTYPE` category. The wide-character set is fixed by the implementation (in the case of GNU C library it is always UCS-4 encoded ISO 10646.

This has of course several problems when it comes to general character conversion:

- For every conversion where neither the source nor the destination character set is the character set of the locale for the `LC_CTYPE` category, one has to change the `LC_CTYPE` locale using `setlocale`.

  Changing the `LC_TYPE` locale introduces major problems for the rest of the programs since several more functions, like the character-classification functions (see Section 4.1 [Classification of Characters], page 79), use the `LC_CTYPE` category.

- Parallel conversions to and from different character sets are not possible since the `LC_CTYPE` selection is global and shared by all threads.

- If neither the source nor the destination character set is the character set used for `wchar_t` representation, there is at least a two-step process necessary to convert a text using the functions above. One would have to select the source character set as the multibyte encoding, convert the text into a `wchar_t` text, select the destination character set as the multibyte encoding, and convert the wide-character text to the multibyte (= destination) character set.

  Even if this is possible (which is not guaranteed), it is a very tiring work. Plus it suffers from the other two aforementioned problems even more, due to the steady changing of the locale.

The XPG2 standard defines a completely new set of functions, which has none of these limitations. They are not at all coupled to the selected locales, and they have no constraints on the character sets selected for source and destination. Only the set of available conversions limits them. The standard does not specify that any conversion at all must be available. Such availability is a measure of the quality of the implementation.

In the following text, first the interface to `iconv` and then the conversion function will be described. Comparisons with other implementations will show what obstacles stand in the way of portable applications. Finally, the implementation is described in so far as might interest the advanced user who wants to extend conversion capabilities.

6.5.1 Generic Character-Set Conversion Interface

This set of functions follows the traditional cycle of using a resource—open, use, close. The interface consists of three functions, each of which implements one step.

Before the interfaces are described, it is necessary to introduce a data type. Just like other open-use-close interfaces, the functions introduced here work using handles, and the ‘iconv.h’ header defines a special type for the handles used.

iconv_t Data Type

This data type is an abstract type defined in ‘iconv.h’. The user must not assume anything about the definition of this type; it must be completely opaque.

Objects of this type can get assigned handles for the conversions using the iconv functions. The objects themselves need not be freed, but the conversions that the handles stand for do.

The first step is the function to create a handle.

iconv_t **iconv_open** (const char *tocode, const char *fromcode) Function

The iconv_open function has to be used before starting a conversion. The two parameters this function takes determine the source and destination character set for the conversion, and if the implementation is able to perform such a conversion, the function returns a handle.

If the wanted conversion is not available, the iconv_open function returns (iconv_t) -1. In this case, the global variable errno can have the following values:

EMFILE The process already has OPEN_MAX file descriptors open.

ENFILE The system limit of open files is reached.

ENOMEM There is not enough memory to carry out the operation.

EINVAL The conversion from *fromcode* to *tocode* is not supported.

It is not possible to use the same descriptor in different threads to perform independent conversions. The data structures associated with the descriptor include information about the conversion state. This must not be messed up by using it in different conversions.

An iconv descriptor is like a file descriptor, as for every use a new descriptor must be created. The descriptor does not stand for all of the conversions from *fromset* to *toset*.

The GNU C Library implementation of iconv_open has one significant extension to other implementations. To ease the extension of the set of available conversions, the implementation allows storing the necessary files with data and code in an arbitrary number of directories. How this extension must be written will be explained below (see Section 6.5.4 [The iconv Implementation in the GNU C Library], page 165). Here it is only important to say that all directories mentioned in the GCONV_PATH environment variable are considered only if they contain a file ‘gconv-modules’. These directories need not necessarily be created by the system administrator. In fact, this extension is introduced to help users writing and using their own, new conversions. Of course,

this does not work for security reasons in SUID binaries; in this case only the system directory is considered—this is normally '*prefix*`/lib/gconv`'. The `GCONV_PATH` environment variable is examined exactly once at the first call of the `iconv_open` function. Later modifications of the variable have no effect.

The `iconv_open` function was introduced early in the *X/Open Portability Guide,* Issue 2.[1] It is supported by all commercial Unices, as it is required for the Unix branding. However, the quality and completeness of the implementation varies widely. The `iconv_open` function is declared in '`iconv.h`'.

The `iconv` implementation can associate large data structures with the handle returned by `iconv_open`. Therefore, it is crucial to free all the resources once all conversions are carried out and the conversion is not needed anymore.

`int` **iconv_close** (`iconv_t` *cd*) Function

The `iconv_close` function frees all resources associated with the handle *cd*, which must have been returned by a successful call to the `iconv_open` function.

If the function call was successful, the return value is 0. Otherwise, it is -1 and `errno` is set appropriately. Defined errors are

`EBADF` The conversion descriptor is invalid.

The `iconv_close` function was introduced together with the rest of the `iconv` functions in XPG2 and is declared in '`iconv.h`'.

The standard defines only one actual conversion function. This has, therefore, the most general interface: it allows conversion from one buffer to another. Conversion from a file to a buffer, from a buffer to a file, or even from a file to file can be implemented on top of it.

`size_t` **iconv** (`iconv_t` *cd*, `char` `**`*inbuf*, `size_t` `*`*inbytesleft*, `char` `**`*outbuf*, `size_t` `*`*outbytesleft*) Function

The `iconv` function converts the text in the input buffer according to the rules associated with the descriptor *cd* and stores the result in the output buffer. It is possible to call the function for the same text several times in a row, since for stateful character sets the necessary state information is kept in the data structures associated with the descriptor.

The input buffer is specified by `*`*inbuf* and it contains `*`*inbytesleft* bytes. The extra indirection is necessary for communicating the used input back to the caller (see below). It is important to note that the buffer pointer is of type `char` and the length is measured in bytes even if the input text is encoded in wide characters.

The output buffer is specified in a similar way. `*`*outbuf* points to the beginning of the buffer with at least `*`*outbytesleft* bytes of room for the result. The buffer

[1] X/Open Company, *X/Open Portability Guide,* Issue 2 (Reading, UK: X/Open Company, Ltd., 1987).

pointer again is of type `char` and the length is measured in bytes. If *outbuf* or **outbuf* is a null pointer, the conversion is performed but no output is available.

If *inbuf* is a null pointer, the `iconv` function performs the necessary action to put the state of the conversion into the initial state. This is obviously a no-op for nonstateful encodings, but if the encoding has a state, such a function call might put some byte sequences in the output buffer, which perform the necessary state changes. The next call with *inbuf* not being a null pointer then goes on from the initial state. It is important that the programmer never make any assumption as to whether the conversion has to deal with states. Even if the input and output character sets are not stateful, the implementation might still have to keep states. This is due to the implementation chosen for the GNU C Library as it is described below. Therefore an `iconv` call to reset the state should always be performed if some protocol requires this for the output text.

The conversion stops for one of three reasons. The first is that all characters from the input buffer are converted. This actually can mean two things: either all bytes from the input buffer are consumed or there are some bytes at the end of the buffer that can possibly form a complete character, but the input is incomplete. The second reason for a stop is that the output buffer is full. The third reason is that the input contains invalid characters.

In all of these cases, the buffer pointers after the last successful conversion, for input and output buffer, are stored in *inbuf* and *outbuf*, and the available room in each buffer is stored in *inbytesleft* and *outbytesleft*.

Since the character sets selected in the `iconv_open` call can be almost arbitrary, there can be situations where the input buffer contains valid characters, which have no identical representation in the output character set. The behavior in this situation is undefined. The *current* behavior of the GNU C Library in this situation is to return with an error immediately. This certainly is not the most desirable solution. Future versions will provide better solutions, but they are not yet finished.

If all input from the input buffer is successfully converted and stored in the output buffer, the function returns the number of nonreversible conversions performed. In all other cases, the return value is `(size_t) -1` and `errno` is set appropriately. In such cases, the value pointed to by *inbytesleft* is nonzero.

`EILSEQ` The conversion stopped because of an invalid byte-sequence in the input. After the call, **inbuf* points at the first byte of the invalid byte-sequence.

`E2BIG` The conversion stopped because it ran out of space in the output buffer.

`EINVAL` The conversion stopped because of an incomplete byte-sequence at the end of the input buffer.

`EBADF` The *cd* argument is invalid.

The `iconv` function was introduced in the XPG2 standard and is declared in the '`iconv.h`' header.

The definition of the `iconv` function is quite good overall. It provides flexible functionality. The only problems lie in the boundary cases, which are incomplete byte-sequences at the end of the input buffer and invalid input. A third problem, which is not really a design problem, is the way conversions are selected. The standard does not say anything about the legitimate names—a minimal set of available conversions. We will see how this negatively impacts other implementations, as demonstrated below.

6.5.2 A Complete `iconv` Example

The example below features a solution for a common problem. Given that we know the internal encoding used by the system for `wchar_t` strings, we often are in the position to read text from a file and store it in wide-character buffers. We can do this using `mbsrtowcs`, but then we run into the problems discussed above.

```
int
file2wcs (int fd, const char *charset, wchar_t *outbuf, size_t avail)
{
  char inbuf[BUFSIZ];
  size_t insize = 0;
  char *wrptr = (char *) outbuf;
  int result = 0;
  iconv_t cd;

  cd = iconv_open ("WCHAR_T", charset);
  if (cd == (iconv_t) -1)
    {
      /* Something went wrong.  */
      if (errno == EINVAL)
        error (0, 0, "conversion from '%s' to wchar_t not available",
               charset);
      else
        perror ("iconv_open");

      /* Terminate the output string.  */
      *outbuf = L'\0';

      return -1;
    }

  while (avail > 0)
    {
      size_t nread;
      size_t nconv;
      char *inptr = inbuf;
```

```
  /* Read more input.  */
  nread = read (fd, inbuf + insize, sizeof (inbuf) - insize);
  if (nread == 0)
    {
      /* When we come here the file is completely read.
         This still could mean there are some unused
         characters in the inbuf.  Put them back.  */
      if (lseek (fd, -insize, SEEK_CUR) == -1)
        result = -1;

      /* Now write out the byte sequence to get into the
         initial state if this is necessary.  */
      iconv (cd, NULL, NULL, &wrptr, &avail);

      break;
    }
  insize += nread;

  /* Do the conversion.  */
  nconv = iconv (cd, &inptr, &insize, &wrptr, &avail);
  if (nconv == (size_t) -1)
    {
      /* Not everything went right.  It might only be
         an unfinished byte-sequence at the end of the
         buffer, or it could be a real problem.  */
      if (errno == EINVAL)
        /* This is harmless.  Simply move the unused
           bytes to the beginning of the buffer so that
           they can be used in the next round.  */
        memmove (inbuf, inptr, insize);
      else
        {
          /* It is a real problem.  Maybe we ran out of
             space in the output buffer or we have invalid
             input.  In any case back the file pointer to
             the position of the last processed byte.  */
          lseek (fd, -insize, SEEK_CUR);
          result = -1;
          break;
        }
    }
}
```

```
  /* Terminate the output string.  */
  if (avail >= sizeof (wchar_t))
    *((wchar_t *) wrptr) = L'\0';

  if (iconv_close (cd) != 0)
    perror ("iconv_close");

  return (wchar_t *) wrptr - outbuf;
}
```

This example shows the most important aspects of using the `iconv` functions. It shows how successive calls to `iconv` can be used to convert large amounts of text. The user does not have to care about stateful encodings, since the functions take care of everything.

An interesting point is the case where `iconv` returns an error and `errno` is set to `EINVAL`. This is not really an error in the transformation. It can happen whenever the input character set contains byte sequences of more than 1 byte for some character, and texts are not processed in one piece. In this case, there is a chance that a multibyte sequence is cut. The caller can then simply read the remainder of the takes and feed the offending bytes together with new characters from the input to `iconv` and continue the work. The internal state kept in the descriptor is *not* unspecified after such an event, as is the case with the conversion functions from the ISO C standard.

The example also shows the problem of using wide-character strings with `iconv`. As explained in the description of the `iconv` function above, the function always takes a pointer to a `char` array, and the available space is measured in bytes. In the example, the output buffer is a wide-character buffer; therefore, we use a local variable *wrptr* of type `char *`, which is used in the `iconv` calls.

This looks rather innocent but can lead to problems on platforms that have tight restrictions on alignment. Therefore, the caller of `iconv` has to make sure that the pointers passed are suitable for accessing characters from the appropriate character set. Since, in the above case, the input parameter to the function is a `wchar_t` pointer, this is the case (unless the user violates alignment when computing the parameter). In other situations, especially when writing generic functions where you do not know what type of character set you are using, `iconv` treats text as a sequence of bytes, which might become tricky.

6.5.3 Some Details About Other `iconv` Implementations

This is not really the place to discuss the `iconv` implementation of other systems, but it is necessary to know a bit about them to write portable programs. The aforementioned problems with the specification of the `iconv` functions can lead to portability issues.

The first thing to notice is that, due to the large number of character sets in use, it is certainly not practical to encode the conversions directly in the C library.

Therefore, the conversion information must come from files outside the C library. This is usually done in one or both of the following ways:

- The C library contains a set of generic conversion functions that can read the needed conversion tables and other information from data files. These files get loaded when necessary.

  This solution is problematic, since it requires a great deal of effort to apply to all character sets (potentially an infinite set). The differences in the structures of the different character sets are so large that many different variants of the table-processing functions must be developed. In addition, the generic nature of these functions makes them slower than specifically implemented functions.

- The C library only contains a framework that can dynamically load object files and execute the conversion functions contained therein.

  This solution provides much more flexibility. The C library itself contains only very little code and therefore reduces the general memory footprint. Also, with a documented interface between the C library and the loadable modules, it is possible for third parties to extend the set of available conversion modules. A drawback of this solution is that dynamic loading must be available.

Some implementations in commercial Unices implement a mixture of these possibilities; the majority implement only the second solution. Using loadable modules moves the code out of the library itself and keeps the door open for extensions and improvements, but this design is also limiting on some platforms, since not many platforms support dynamic loading in statically linked programs. On platforms without this capability, it is therefore not possible to use this interface in statically linked programs. The GNU C Library has, on ELF platforms, no problems with dynamic loading in these situations; therefore, this point is moot. The danger is that you get acquainted with this situation and forget about the restrictions on other systems.

A second thing to know about other `iconv` implementations is that the number of available conversions is often very limited. Some implementations provide, in the standard release (not special international or developer releases), at most 100 to 200 conversion possibilities. This does not mean 200 different character sets are supported; for example, conversions from one character set to a set of ten others might count as ten conversions. Together with the other direction, this makes twenty conversion possibilities used up by one character set. You can imagine the thin coverage these platform provide. Some Unix vendors even provide only a handful of conversions, which renders them useless for almost all uses.

This leads directly to the third and probably most problematic point. The way the `iconv` conversion functions are implemented on all known Unix systems and the availability of the conversion functions from character set $\mathcal{A}$ to $\mathcal{B}$ and the conversion from $\mathcal{B}$ to $\mathcal{C}$ does *not* imply that the conversion from $\mathcal{A}$ to $\mathcal{C}$ is available.

This might not seem unreasonable or problematic at first, but it is a big problem. To show the problem, we assume a program that has to convert from $\mathcal{A}$ to $\mathcal{C}$. A call like:

```
cd = iconv_open ("C", "A");
```

fails according to the assumption above. But what does the program do now? The conversion is necessary; therefore, simply giving up is not an option.

This is a nuisance. The `iconv` function should take care of this. But how should the program proceed from here on? If it tries to convert to character set $\mathcal{B}$, first the two `iconv_open` calls:

```
cd1 = iconv_open ("B", "A");
```

and

```
cd2 = iconv_open ("C", "B");
```

will succeed, but how to find $\mathcal{B}$?

Unfortunately, the answer is that there is no general solution. On some systems, guessing might help. On those systems, most character sets can convert to and from UTF-8 encoded ISO 10646 or Unicode text. Besides this, only some very system-specific methods can help. Since the conversion functions come from loadable modules and these modules must be stored somewhere in the file system, one *could* try to find them and determine from the available file which conversions are available and whether there is an indirect route from $\mathcal{A}$ to $\mathcal{C}$.

This example shows one of the design errors of `iconv` mentioned above. It should at least be possible to determine the list of available conversions programmatically so that if `iconv_open` says there is no such conversion, one could make sure this also is true for indirect routes.

6.5.4 The `iconv` Implementation in the GNU C Library

After reading about the problems of `iconv` implementations in the last section, it is certainly good to note that the implementation in the GNU C Library has none of the problems mentioned above. What follows is a step-by-step analysis of the points raised above. The evaluation is based on the current state of the development (as of January 1999). The development of the `iconv` functions is not complete, but basic functionality has solidified.

The GNU C Library's `iconv` implementation uses shared loadable modules to implement the conversions. A very small number of conversions are built into the library itself, but these are only rather trivial conversions.

All the benefits of loadable modules are available in the GNU C Library implementation. This is especially appealing since the interface is well documented (see below); therefore, it is easy to write new conversion modules. The drawback of using loadable objects is not a problem in the GNU C Library, at least on ELF systems. Since the library is able to load shared objects even in statically linked binaries, static linking need not be forbidden in case one wants to use `iconv`.

The second mentioned problem is the number of supported conversions. Currently, the GNU C Library supports more than 150 character sets. The way the implementation is designed the number of supported conversions is greater than 22350 (150 times 149). If any conversion from or to a character set is missing, it can be added easily.

Particularly impressive as it may be, this high number is due to the fact that the GNU C Library implementation of `iconv` does not have the third problem mentioned above (i.e., whenever there is a conversion from a character set $\mathcal{A}$ to $\mathcal{B}$ and from $\mathcal{B}$ to $\mathcal{C}$ it is always possible to convert from $\mathcal{A}$ to $\mathcal{C}$ directly). If the `iconv_open` returns an error and sets `errno` to `EINVAL`, there is no known way, directly or indirectly, to perform the wanted conversion.

Triangulation is achieved by providing for each character set a conversion from and to UCS-4 encoded ISO 10646. Using ISO 10646 as an intermediate representation, it is possible to *triangulate* (i.e., convert with an intermediate representation).

There is no inherent requirement to provide a conversion to ISO 10646 for a new character set, and it is also possible to provide other conversions where neither source nor destination character set is ISO 10646. The existing set of conversions is simply meant to cover all conversions that might be of interest.

All currently available conversions use the triangulation method above, making conversion run unnecessarily slow. If, for example, somebody often needs the conversion from ISO-2022-JP to EUC-JP, a quicker solution would involve direct conversion between the two character sets, skipping the input to ISO 10646 first. The two character sets of interest are much more similar to each other than to ISO 10646.

In such a situation, one easily can write a new conversion and provide it as a better alternative. The GNU C Library `iconv` implementation would automatically use the module implementing the conversion if it is specified to be more efficient.

6.5.4.1 Format of ‘`gconv-modules`’ Files

All information about the available conversions comes from a file named ‘`gconv-modules`’, which can be found in any of the directories along the `GCONV_PATH`. The ‘`gconv-modules`’ files are line-oriented text files, where each of the lines has one of the following formats:

- If the first non-white-space character is a ‘`#`’, the line contains only comments and is ignored.
- Lines starting with `alias` define an alias name for a character set. Two more words are expected on the line. The first word defines the alias name, and the second defines the original name of the character set. The effect is that it is possible to use the alias name in the *fromset* or *toset* parameters of `iconv_open` and achieve the same result as when using the real character-set name.

  This is quite important, since a character set often has many different names. There is normally an official name, but this need not correspond to the most popular name. Besides this, many character sets have special names that are somehow constructed. For example, all character sets specified by the ISO have an alias of the form `ISO-IR-`*nnn*, where *nnn* is the registration number. This allows programs that know about the registration number to construct character-set names and use them in `iconv_open` calls. More on the available names and aliases follows below.

- Lines starting with `module` introduce an available conversion module. These lines must contain three or four more words.

  The first word specifies the source character set, the second word the destination character set of conversion implemented in this module, and the third word is the name of the loadable module. The file name is constructed by appending the usual shared-object suffix (normally '`.so`') and this file is then supposed to be found in the same directory the '`gconv-modules`' file is in. The last word on the line, which is optional, is a numeric value representing the cost of the conversion. If this word is missing, a cost of 1 is assumed. The numeric value itself does not matter that much; the relative values of the sums of costs for all possible conversion paths are what counts. Below is a more precise description of the use of the cost value.

Returning to the example above of a module written to directly convert from ISO-2022-JP to EUC-JP and back, all that has to be done is to put the new module (let its name be ISO2022JP-EUCJP.so) in a directory and add a file '`gconv-modules`' with the following content in the same directory:

```
module  ISO-2022-JP//   EUC-JP//        ISO2022JP-EUCJP    1
module  EUC-JP//        ISO-2022-JP//   ISO2022JP-EUCJP    1
```

To see why this is sufficient, it is necessary to understand how the conversion used by `iconv` (and described in the descriptor) is selected. The approach to this problem is quite simple.

At the first call of the `iconv_open` function, the program reads all available '`gconv-modules`' files and builds up two tables—one containing all the known aliases and another that contains the information about the conversions and which shared object implements them.

6.5.4.2 Finding the Conversion Path in `iconv`

The set of available conversions form a directed graph with weighted edges. The weights on the edges are the costs specified in the '`gconv-modules`' files. The `iconv_open` function uses an algorithm suitable for searching for the best path in such a graph and so constructs a list of conversions that must be performed in succession to get the transformation from the source to the destination character set.

Explaining why the above '`gconv-modules`' file allows the `iconv` implementation to resolve the specific ISO-2022-JP to EUC-JP conversion module instead of the conversion coming with the library itself is straightforward. Since the latter conversion takes two steps (from ISO-2022-JP to ISO 10646 and then from ISO 10646 to EUC-JP), the cost is 1 + 1 = 2. The above '`gconv-modules`' file, however, specifies that the new conversion modules can perform this conversion with only the cost of 1.

A mysterious item about the '`gconv-modules`' file above (and also the file that comes with the GNU C Library) is the names of the character sets specified in the `module` lines. Almost all of the names end in `//`. Some names are actually

regular expressions. The part of the implementation where this is used is not yet finished. For now please simply follow the existing examples. It should become clearer as you read on.

A last remark about the 'gconv-modules' is about the names not ending with //. A character set named INTERNAL is often mentioned. From the discussion above and the chosen name, it should have become clear that this is the name for the representation used in the intermediate step of the triangulation. We have said that this is UCS-4, but actually that is not quite right. The UCS-4 specification also includes the specification of the byte ordering used. Since a UCS-4 value consists of 4 bytes, a stored value is effected by byte ordering. The internal representation is *not* the same as UCS-4 in case the byte ordering of the processor (or at least the running process) is not the same as the one required for UCS-4. This is done for performance reasons as one does not want to perform unnecessary byte-swapping operations if one is not interested in actually seeing the result in UCS-4. To avoid trouble with endianess, the internal representation consistently is named INTERNAL even on big-endian systems where the representations are identical.

6.5.4.3 `iconv` Module Data Structures

So far, this section has described how modules are located and considered to be used. What remains to be described is the interface of the modules so that you can write new ones. This section describes the interface as it is in use in January 1999. The interface will change a bit in the future but, with luck, only in an upwardly compatible way.

The definitions necessary to write new modules are publicly available in the non-standard header 'gconv.h'. The following text, therefore, describes the definitions from this header file. First, however, it is necessary to get an overview.

From the perspective of the user of iconv, the interface is quite simple; the iconv_open function returns a handle that can be used in calls to iconv, and then the handle is freed with a call to iconv_close. The problem is that the handle has to be able to represent the possibly long sequences of conversion steps and also the state of each conversion, since the handle is all that is passed to the iconv function. Therefore, the data structures are really the elements necessary to understanding the implementation.

We need two different kinds of data structures. The first describes the conversion and the second describes the state, etc. There are really two type definitions like this in 'gconv.h'.

struct __gconv_step Data type

This data structure describes one conversion a module can perform. For each function in a loaded module with conversion functions, there is exactly one object of this type. This object is shared by all users of the conversion (i.e., this object does not contain any information corresponding to an actual conversion; it only describes the conversion itself).

`struct __gconv_loaded_object *__shlib_handle`
`const char *__modname`
`int __counter`
: All these elements of the structure are used internally in the C library to coordinate loading and unloading the shared object. You should not expect any of the other elements to be available or initialized.

`const char *__from_name`
`const char *__to_name`
: `__from_name` and `__to_name` contain the names of the source and destination character sets. They can be used to identify the actual conversion to be carried out, since one module might implement conversions for more than one character set and/or direction.

`gconv_fct __fct`
`gconv_init_fct __init_fct`
`gconv_end_fct __end_fct`
: These elements contain pointers to the functions in the loadable module. The interface will be explained below.

`int __min_needed_from`
`int __max_needed_from`
`int __min_needed_to`
`int __max_needed_to;`
: These values have to be supplied in the init function of the module. The `__min_needed_from` value specifies the minimum number of bytes a character of the source character set needs. The `__max_needed_from` specifies the maximum value that also includes possible shift sequences.

  The `__min_needed_to` and `__max_needed_to` values serve the same purpose as `__min_needed_from` and `__max_needed_from` but this time for the destination character set.

  It is crucial that these values be accurate, since otherwise the conversion functions will have problems or not work at all.

`int __stateful`
: This element must also be initialized by the init function. `int __stateful` is nonzero if the source character set is stateful. Otherwise, it is zero.

`void *__data`
: This element can be used freely by the conversion functions in the module. `void *__data` can be used to communicate extra information from one call to another. `void *__data` need not be initialized if it is not needed at all. If `void *__data` element is assigned a pointer to dynamically allocated memory (presum-

ably in the init function), it has to be ensured that the end function de-allocates the memory. Otherwise, the application will leak memory.

It is important to be aware that this data structure is shared by all users of this specification conversion and therefore, the `__data` element must not contain data specific to any particular use of the conversion function.

struct __gconv_step_data Data type

This is the data structure that contains the information specific to each use of the conversion functions.

`char *__outbuf`
`char *__outbufend`

These elements specify the output buffer for the conversion step. The `__outbuf` element points to the beginning of the buffer, and `__outbufend` points to the byte following the last byte in the buffer. The conversion function must not assume anything about the size of the buffer, but it can be safely assumed that there is room for at least one complete character in the output buffer.

Once the conversion is finished, if the conversion is the last step, the `__outbuf` element must be modified to point after the last byte written into the buffer to signal how much output is available. If this conversion step is not the last one, the element must not be modified. The `__outbufend` element must not be modified.

`int __is_last`

This element is nonzero if this conversion step is the last one. This information is necessary for the recursion. See the description of the conversion function internals below. This element must never be modified.

`int __invocation_counter`

The conversion function can use this element to see how many calls of the conversion function already happened. Some character sets require a certain prolog when generating output, and by comparing this value with zero, one can find out whether it is the first call and whether, therefore, the prolog should be emitted. This element must never be modified.

`int __internal_use`

This element is another one rarely used but needed in certain situations. It is assigned a nonzero value in case the conversion functions are used to implement `mbsrtowcs` or others like it (i.e., the function is not used directly through the `iconv` interface).

This sometimes makes a difference, since it is expected that the `iconv` functions are used to translate entire texts, while the

`mbsrtowcs` functions are normally used only to convert single strings and might be used multiple times to convert entire texts.

But in this situation we would have problems complying with some rules of the character-set specification. Some character sets require a prolog, which must appear exactly once for an entire text. If a number of `mbsrtowcs` calls are used to convert the text, only the first call must add the prolog. However, because there is no communication between the different calls of `mbsrtowcs`, the conversion functions have no chance to find this out. The situation is different for sequences of `iconv` calls, since the handle allows access to the needed information.

The `int __internal_use` element is mostly used together with `__invocation_counter` as follows:

```
if (!data->__internal_use
      && data->__invocation_counter == 0)
  /* Emit prolog.  */
  ...
```

This element must never be modified.

`mbstate_t *__statep`
: The `__statep` element points to an object of type `mbstate_t` (see Section 6.3.2 [Representing the State of the Conversion], page 139). The conversion of a stateful character set must use the object pointed to by `__statep` to store information about the conversion state. The `__statep` element itself must never be modified.

`mbstate_t __state`
: This element must *never* be used directly. It is only part of this structure to have the needed space allocated.

6.5.4.4 `iconv` Module Interfaces

With the knowledge about the data structures, we can now describe the conversion function itself. To understand the interface, a bit of knowledge about the functionality in the C library that loads the objects with the conversions is necessary.

It is often the case that one conversion is used more than once (i.e., there are several `iconv_open` calls for the same set of character sets during one program run). The `mbsrtowcs` functions in the GNU C Library also use the `iconv` functionality, which increases the number of uses of the same functions even more.

Because of this multiple use of conversions, the modules do not get loaded exclusively for one conversion. Instead, a module once loaded can be used by an arbitrary number of `iconv` or `mbsrtowcs` calls at the same time. The splitting of the information between conversion-function-specific information and conver-

sion data makes this possible. The last section showed the two data structures used to do this.

This is also reflected in the interface and semantics of the functions that the modules must provide. There are three functions, which must have the following names:

`gconv_init`
: The `gconv_init` function initializes the conversion-function-specific data structure. This very same object is shared by all conversions that use this conversion and, therefore, no state information about the conversion itself must be stored in here. If a module implements more than one conversion, the `gconv_init` function will be called multiple times.

`gconv_end`
: The `gconv_end` function is responsible for freeing all resources allocated by the `gconv_init` function. If there is nothing to do, this function can be missing. Special care must be taken if the module implements more than one conversion and the `gconv_init` function does not allocate the same resources for all conversions.

`gconv`
: This is the actual conversion function. It is called to convert one block of text. It gets passed the conversion-step information initialized by `gconv_init` and the conversion data, specific to this use of the conversion functions.

There are three data types defined for the three module interface functions, and these define the interface.

`int` **(*__gconv_init_fct) (struct __gconv_step *)** Data type

This specifies the interface of the initialization function of the module. It is called exactly once for each conversion the module implements.

As explained in the description of the `struct __gconv_step` data structure above, the initialization function has to initialize parts of it.

`__min_needed_from`
`__max_needed_from`
`__min_needed_to`
`__max_needed_to`
: These elements must be initialized to the exact numbers of the minimum and maximum number of bytes used by one character in the source and destination character sets, respectively. If the characters all have the same size, the minimum and maximum values are the same.

`__stateful`
: This element must be initialized to a nonzero value if the source character set is stateful. Otherwise, it must be zero.

If the initialization function needs to communicate some information to the conversion function, this communication can happen using the `__data` element of the `__gconv_step` structure. But since this data is shared by all the conversions, it must not be modified by the conversion function. The example below shows how this can be used.

```
#define MIN_NEEDED_FROM         1
#define MAX_NEEDED_FROM         4
#define MIN_NEEDED_TO           4
#define MAX_NEEDED_TO           4

int
gconv_init (struct __gconv_step *step)
{
  /* Determine which direction.  */
  struct iso2022jp_data *new_data;
  enum direction dir = illegal_dir;
  enum variant var = illegal_var;
  int result;

  if (__strcasecmp (step->__from_name, "ISO-2022-JP//") == 0)
    {
      dir = from_iso2022jp;
      var = iso2022jp;
    }
  else if (__strcasecmp (step->__to_name, "ISO-2022-JP//") == 0)
    {
      dir = to_iso2022jp;
      var = iso2022jp;
    }
  else if (__strcasecmp (step->__from_name, "ISO-2022-JP-2//") == 0)
    {
      dir = from_iso2022jp;
      var = iso2022jp2;
    }
  else if (__strcasecmp (step->__to_name, "ISO-2022-JP-2//") == 0)
    {
      dir = to_iso2022jp;
      var = iso2022jp2;
    }

  result = __GCONV_NOCONV;
  if (dir != illegal_dir)
    {
      new_data = (struct iso2022jp_data *)
```

```
            malloc (sizeof (struct iso2022jp_data));

          result = __GCONV_NOMEM;
          if (new_data != NULL)
            {
              new_data->dir = dir;
              new_data->var = var;
              step->__data = new_data;

              if (dir == from_iso2022jp)
                {
                  step->__min_needed_from = MIN_NEEDED_FROM;
                  step->__max_needed_from = MAX_NEEDED_FROM;
                  step->__min_needed_to = MIN_NEEDED_TO;
                  step->__max_needed_to = MAX_NEEDED_TO;
                }
              else
                {
                  step->__min_needed_from = MIN_NEEDED_TO;
                  step->__max_needed_from = MAX_NEEDED_TO;
                  step->__min_needed_to = MIN_NEEDED_FROM;
                  step->__max_needed_to = MAX_NEEDED_FROM + 2;
                }

              /* Yes, this is a stateful encoding.  */
              step->__stateful = 1;

              result = __GCONV_OK;
            }
        }

      return result;
    }
```

The function first checks which conversion is wanted. The module from which this function is taken implements four different conversions; the one that is selected can be determined by comparing the names. The comparison should always be done without paying attention to the case.

Next, a data structure, which contains the necessary information about which conversion is selected, is allocated. The data structure `struct iso2022jp_data` is locally defined since, outside the module, this data is not used at all. If all four conversions this module supports are requested, there are four data blocks.

The initialization of the `__min_` and `__max_` elements of the step-data object is interesting. A single ISO-2022-JP character can consist of 1 to 4 bytes.

Therefore, the `MIN_NEEDED_FROM` and `MAX_NEEDED_FROM` macros are defined this way. The output is always the `INTERNAL` character set (aka UCS-4), and therefore each character consists of exactly 4 bytes. For the conversion from `INTERNAL` to ISO-2022-JP, we have to take into account that escape sequences might be necessary to switch the character sets. Therefore, the `__max_needed_to` element for this direction gets assigned `MAX_NEEDED_FROM + 2`. This takes into account the 2 bytes needed for the escape sequences to signal the switching. The asymmetry in the maximum values for the two directions can be explained easily, when reading ISO-2022-JP text, escape sequences can be handled alone (i.e., it is not necessary to process a real character, since the effect of the escape sequence can be recorded in the state information). The situation is different for the other direction. Since it is in general not known which character comes next, you cannot emit escape sequences to change the state in advance. This means the escape sequences have to be emitted together with the next character. Therefore, you need room for more than only the character itself.

The possible return values of the initialization function are

`__GCONV_OK`
: The initialization succeeded.

`__GCONV_NOCONV`
: The requested conversion is not supported in the module. This can happen if the '`gconv-modules`' file has errors.

`__GCONV_NOMEM`
: Memory required to store additional information could not be allocated.

The function called before the module is unloaded is significantly easier. It often has nothing at all to do, in which case it can be left out completely.

`void` **(*__gconv_end_fct) (struct gconv_step *)** Data type

The task of this function is to free all resources allocated in the initialization function. Therefore, only the `__data` element of the object pointed to by the argument is of interest. Continuing the example from the initialization function, the finalization function looks like this:

```
void
gconv_end (struct __gconv_step *data)
{
  free (data->__data);
}
```

The most important function is the conversion function itself, which can get quite complicated for complex character sets. But since this is not of interest here, we will only describe a possible skeleton for the conversion function.

`int` Data type

(*__gconv_fct) (struct __gconv_step *, struct __gconv_step_data *, const char **, const char *, size_t *, int)

The conversion function can be called for two basic reasons: to convert text or to reset the state. From the description of the `iconv` function, it can be seen why the flushing mode is necessary. The mode selected is determined by the sixth argument, an integer. This argument being nonzero means that flushing is selected.

Common to both modes is the location of the output buffer. The information about this buffer is stored in the conversion-step data. A pointer to this information is passed as the second argument to this function. The description of the `struct __gconv_step_data` structure has more information on the conversion-step data.

What has to be done for flushing depends on the source character set. If the source character set is not stateful, nothing has to be done. Otherwise, the function has to emit a byte sequence to bring the state object into the initial state. Once this has all happened, the other conversion modules in the chain of conversions have to get the same chance. Whether another step follows can be determined from the `__is_last` element of the step data structure to which the first parameter points.

The more interesting mode is when actual text has to be converted. The first step in this case is to convert as much text as possible from the input buffer and store the result in the output buffer. The start of the input buffer is determined by the third argument, which is a pointer to a pointer variable referencing the beginning of the buffer. The fourth argument is a pointer to the byte right after the last byte in the buffer.

The conversion has to be performed according to the current state if the character set is stateful. The state is stored in an object pointed to by the `__statep` element of the step data (second argument). Once either the input buffer is empty or the output buffer is full, the conversion stops. At this point, the pointer variable referenced by the third parameter must point to the byte following the last processed byte (i.e., if all of the input is consumed, this pointer and the fourth parameter have the same value).

What happens now depends on whether this step is the last one. If it is the last step, the only thing to be done is to update the `__outbuf` element of the step data structure to point after the last written byte. This update gives the caller the information on how much text is available in the output buffer. In addition, the variable pointed to by the fifth parameter, which is of type `size_t`, must be incremented by the number of characters (*not bytes*) that were converted in a nonreversible way. Then, the function can return.

In case the step is not the last one, the later conversion functions have to get a chance to do their work. Therefore, the appropriate conversion function has to be called. The information about the functions is stored in the conversion data structures, passed as the first parameter. This information and the step data

are stored in arrays, so the next element in both cases can be found by simple pointer arithmetic:

```
int
gconv (struct __gconv_step *step, struct __gconv_step_data *data,
       const char **inbuf, const char *inbufend, size_t *written,
       int do_flush)
{
  struct __gconv_step *next_step = step + 1;
  struct __gconv_step_data *next_data = data + 1;
  ...
```

The `next_step` pointer references the next step information, and `next_data` references the next data record. The call of the next function therefore will look similar to this:

```
next_step->__fct (next_step, next_data, &outerr, outbuf,
                  written, 0)
```

But this is not yet all. Once the function call returns, the conversion function might have some more to do. If the return value of the function is `__GCONV_EMPTY_INPUT`, more room is available in the output buffer. Unless the input buffer is empty, the conversion functions start all over again and process the rest of the input buffer. If the return value is not `__GCONV_EMPTY_INPUT`, something went wrong and we will have to recover from this.

A requirement for the conversion function is that the input buffer pointer (the third argument) always point to the last character that was put in converted form into the output buffer. This is true after the conversion performed in the current step, but if the conversion functions deeper downstream stop prematurely, not all characters from the output buffer are consumed and, therefore, the input buffer pointers must be backed off to the right position.

Correcting the input buffers is easy to do if the input and output character sets have a fixed width for all characters. In this situation, we can compute how many characters are left in the output buffer and, therefore, can correct the input buffer pointer appropriately with a similar computation. Things are getting tricky if either character set has characters represented with variable-length byte sequences, and it gets even more complicated if the conversion has to take care of the state. In these cases, the conversion has to be performed once again, from the known state before the initial conversion (i.e., if necessary the state of the conversion has to be reset and the conversion loop has to be executed again). The difference now is that it is known how much input must be created, and the conversion can stop before converting the first unused character. Once this is done, the input buffer pointers must be updated again and the function can return.

One final thing should be mentioned. If it is necessary for the conversion to know whether it is the first invocation (in case a prolog has to be emitted), the conversion function should increment the `__invocation_counter` element of the step data structure just before returning to the caller. See the de-

scription of the `struct __gconv_step_data` structure in Section 6.5.4.3 [iconv Module Data Structures], page 168 for more information on how this can be used.

The return value must be one of the following values:

`__GCONV_EMPTY_INPUT`
: All input was consumed and there is room left in the output buffer.

`__GCONV_FULL_OUTPUT`
: No more room in the output buffer. In case this is not the last step, this value is propagated down from the call of the next conversion function in the chain.

`__GCONV_INCOMPLETE_INPUT`
: The input buffer is not entirely empty, since it contains an incomplete character sequence.

The following example provides a framework for a conversion function. In case a new conversion has to be written, the following is a template that can be filled in to do the job:

```
int
gconv (struct __gconv_step *step, struct __gconv_step_data *data,
       const char **inbuf, const char *inbufend, size_t *written,
       int do_flush)
{
  struct __gconv_step *next_step = step + 1;
  struct __gconv_step_data *next_data = data + 1;
  gconv_fct fct = next_step->__fct;
  int status;

  /* If the function is called with no input, we have
     to reset to the initial state.  The potentially partly
     converted input is dropped.  */
  if (do_flush)
    {
      status = __GCONV_OK;

      /* Possibly emit a byte sequence that puts the state object
         into the initial state.  */

      /* Call the steps down the chain if there are any but only
         if we successfully emitted the escape sequence.  */
      if (status == __GCONV_OK && ! data->__is_last)
        status = fct (next_step, next_data, NULL, NULL,
                      written, 1);
    }
  else
```

```
{
  /* We preserve the initial values of the pointer variables.  */
  const char *inptr = *inbuf;
  char *outbuf = data->__outbuf;
  char *outend = data->__outbufend;
  char *outptr;

  do
    {
      /* Remember the start value for this round.  */
      inptr = *inbuf;
      /* The outbuf buffer is empty.  */
      outptr = outbuf;

      /* For stateful encodings the state must be safe here.  */

      /* Run the conversion loop.  status is set
         appropriately afterwards.  */

      /* If this is the last step, leave the loop. There is
         nothing we can do.  */
      if (data->__is_last)
        {
          /* Store information about how many bytes are
             available.  */
          data->__outbuf = outbuf;

          /* If any nonreversible conversions were performed,
             add the number to *written.  */

          break;
        }

      /* Write out all output that was produced.  */
      if (outbuf > outptr)
        {
          const char *outerr = data->__outbuf;
          int result;

          result = fct (next_step, next_data, &outerr,
                        outbuf, written, 0);

          if (result != __GCONV_EMPTY_INPUT)
            {
```

```
                    if (outerr != outbuf)
                      {
                        /* Reset the input buffer pointer. Here we
                           document the complex case.  */
                        size_t nstatus;

                        /* Reload the pointers.  */
                        *inbuf = inptr;
                        outbuf = outptr;

                        /* Possibly reset the state.  */

                        /* Redo the conversion, but this time
                           the end of the output buffer is at
                           outerr.  */
                      }

                    /* Change the status.  */
                    status = result;
                  }
                else
                  /* All the output is consumed, we can make
                     another run if everything was ok.  */
                  if (status == __GCONV_FULL_OUTPUT)
                    status = __GCONV_OK;
              }
          }
        while (status == __GCONV_OK);

        /* We finished one use of this step.  */
        ++data->__invocation_counter;
      }

    return status;
  }
```

This information should be sufficient to write new modules. Anybody doing so should also take a look at the available source code in the GNU C Library sources. It contains many examples of working and optimized modules.

7 Locales and Internationalization

Different countries and cultures have varying conventions for how to communicate. These conventions range from very simple ones, such as the format for representing dates and times, to very complex ones, such as the language spoken.

Internationalization of software means programming it to be able to adapt to the user's favorite conventions. In ISO C, internationalization works by means of *locales*. Each locale specifies a collection of conventions, one convention for each purpose. The user chooses a set of conventions by specifying a locale (via environment variables).

All programs inherit the chosen locale as part of their environment. Provided the programs are written to obey the choice of locale, they will follow the conventions preferred by the user.

7.1 What Effects a Locale Has

Each locale specifies conventions for several purposes, including the following:

- What multibyte-character sequences are valid, and how they are interpreted (see Chapter 6 [Character-Set Handling], page 133)
- Classification of which characters in the local character set are considered alphabetic, and uppercase and lowercase conversion conventions (see Chapter 4 [Character Handling], page 79)
- The collating sequence for the local language and character set (see Section 5.6 [Collation Functions], page 109)
- Formatting of numbers and currency amounts (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187)
- Formatting of dates and times (see Section 10.4.5 [Formatting Calendar Time], page 291)
- What language to use for output, including error messages (see Chapter 11 [Message Translation], page 315)
- What language to use for user answers to yes-or-no questions (see Section 7.8 [Yes-or-No Questions], page 200)
- What language to use for more complex user input (the C library doesn't yet help you implement this)

Some aspects of adapting to the specified locale are handled automatically by the library subroutines. For example, all your program needs to do in order to use the collating sequence of the chosen locale is to use `strcoll` or `strxfrm` to compare strings.

Other aspects of locales are beyond the comprehension of the library. For example, the library can't automatically translate your program's output messages into other languages. The only way you can support output in the user's favorite language is to program this more or less by hand. The C library provides functions to handle translations for multiple languages easily.

This chapter discusses the mechanism by which you can modify the current locale. The effects of the current locale on specific library functions are discussed in more detail in the descriptions of those functions.

7.2 Choosing a Locale

The simplest way for the user to choose a locale is to set the environment variable `LANG`. This specifies a single locale to use for all purposes. For example, a user could specify a hypothetical locale named '`espana-castellano`' to use the standard conventions of most of Spain.

The set of locales supported depends on the operating system you are using, and so do their names. We can't make any promises about what locales will exist, except for one standard locale called '`C`' or '`POSIX`'. Later we will describe how to construct locales.

A user also has the option of specifying different locales for different purposes—in effect, choosing a mixture of multiple locales.

For example, the user might specify the locale '`espana-castellano`' for most purposes, but specify the locale '`usa-english`' for currency formatting. This might make sense if the user is a Spanish-speaking American, working in Spanish, but representing monetary amounts in US dollars.

Both locales '`espana-castellano`' and '`usa-english`', like all locales, would include conventions for all of the purposes to which locales apply. However, the user can choose to use each locale for a particular subset of those purposes.

7.3 Categories of Activities That Locales Affect

The purposes that locales serve are grouped into *categories*, so that a user or a program can choose the locale for each category independently. Here is a table of categories; each name is both an environment variable that a user can set, and a macro name that you can use as an argument to `setlocale`.

`LC_COLLATE`
: This category applies to collation of strings (functions `strcoll` and `strxfrm`) (see Section 5.6 [Collation Functions], page 109).

`LC_CTYPE`
: This category applies to classification and conversion of characters, and to multibyte and wide characters (see Chapter 4 [Character Handling], page 79, and Chapter 6 [Character-Set Handling], page 133).

`LC_MONETARY`
: This category applies to formatting monetary values (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187).

`LC_NUMERIC`
: This category applies to formatting numeric values that are not monetary (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187).

`LC_TIME`
: This category applies to formatting date and time values (see Section 10.4.5 [Formatting Calendar Time], page 291).

`LC_MESSAGES`
: This category applies to selecting the language used in the user interface for message translation (see Section 11.2 [The Uniforum Approach to Message Translation], page 325, and Section 11.1 [X/Open Message Catalog Handling], page 315) and contains regular expressions for affirmative and negative responses.

`LC_ALL`
: This is not an environment variable; it is only a macro that you can use with `setlocale` to set a single locale for all purposes. Setting this environment variable overwrites all selections by the other `LC_*` variables or `LANG`.

`LANG`
: If this environment variable is defined, its value specifies the locale to use for all purposes except as overridden by the variables above.

When developing the message translation functions it was felt that the functionality provided by the variables above is not sufficient. For example, it should be possible to specify more than one locale name. Take a Swedish user who speaks German better than English, and a program whose messages are output in English by default. It should be possible to specify that the first choice of language is Swedish, the second German, and if this also fails, to use English. This is possible with the variable `LANGUAGE`. For further description of this GNU extension, see Section 11.2.1.6 [User Influence on `gettext`], page 338.

7.4 How Programs Set the Locale

A C program inherits its locale environment variables when it starts up. This happens automatically. However, these variables do not automatically control the locale used by the library functions, because ISO C says that all programs start by default in the standard 'C' locale. To use the locales specified by the environment, you must call `setlocale`. Call it as follows:

```
setlocale (LC_ALL, "");
```

to select a locale based on the user choice of the appropriate environment variables.

You can also use `setlocale` to specify a particular locale, for general use or for a specific category.

The symbols in this section are defined in the header file '`locale.h`'.

`char *` **setlocale** (`int` *category*, `const char *`*locale*) Function

The function `setlocale` sets the current locale for category *category* to *locale*. A list of all the locales the system provides can be created by running:

```
locale -a
```

If *category* is `LC_ALL`, this specifies the locale for all purposes. The other possible values of *category* specify a single purpose (see Section 7.3 [Categories of Activities That Locales Affect], page 182).

You can also use this function to find out the current locale by passing a null pointer as the *locale* argument. In this case, `setlocale` returns a string that is the name of the locale currently selected for category *category*.

The string returned by `setlocale` can be overwritten by subsequent calls, so you should make a copy of the string (see Section 5.4 [Copying and Concatenation], page 93) if you want to save it past any further calls to `setlocale`. (The standard library is guaranteed never to call `setlocale` itself.)

You should not modify the string returned by `setlocale`. It might be the same string that was passed as an argument in a previous call to `setlocale`. One requirement is that the *category* must be the same in the call the string was returned from and the call where the string is passed in as *locale* parameter.

When you read the current locale for category `LC_ALL`, the value encodes the entire combination of selected locales for all categories. In this case, the value is not just a single locale name. In fact, we don't make any promises about what it looks like. But if you specify the same locale name with `LC_ALL` in a subsequent call to `setlocale`, it restores the same combination of locale selections.

To be sure you can use the returned string encoding the currently selected locale at a later time, you must make a copy of the string. It is not guaranteed that the returned pointer remains valid over time.

When the *locale* argument is not a null pointer, the string returned by `setlocale` reflects the newly-modified locale.

If you specify an empty string for *locale*, this means to read the appropriate environment variable and use its value to select the locale for *category*.

If a nonempty string is given for *locale*, then the locale of that name is used if possible.

If you specify an invalid locale name, `setlocale` returns a null pointer and leaves the current locale unchanged.

Here is an example showing how you might use `setlocale` to temporarily switch to a new locale:

```
#include <stddef.h>
#include <locale.h>
#include <stdlib.h>
#include <string.h>

void
```

```
with_other_locale (char *new_locale,
                   void (*subroutine) (int),
                   int argument)
{
  char *old_locale, *saved_locale;

  /* Get the name of the current locale.  */
  old_locale = setlocale (LC_ALL, NULL);

  /* Copy the name so it won't be clobbered by setlocale. */
  saved_locale = strdup (old_locale);
  if (saved_locale == NULL)
    fatal ("Out of memory");

  /* Now change the locale and do some stuff with it. */
  setlocale (LC_ALL, new_locale);
  (*subroutine) (argument);

  /* Restore the original locale. */
  setlocale (LC_ALL, saved_locale);
  free (saved_locale);
}
```

Portability Note: Some ISO C systems may define additional locale categories, and future versions of the library will do so. For portability, assume that any symbol beginning with '`LC_`' might be defined in '`locale.h`'.

7.5 Standard Locales

The only locale names you can count on finding on all operating systems are these three standard ones:

`''C''` This is the standard C locale. The attributes and behavior it provides are specified in the ISO C standard. When your program starts up, it initially uses this locale by default.

`''POSIX''` This is the standard POSIX locale. Currently, it is an alias for the standard C locale.

`''''` The empty name says to select a locale based on environment variables (see Section 7.3 [Categories of Activities That Locales Affect], page 182).

Defining and installing named locales is normally a responsibility of the system administrator at your site (or the person who installed the GNU C Library). It is also possible for the user to create private locales. All this will be discussed later when describing the tool for such creation.

If your program needs to use something other than the 'C' locale, it will be more portable if you use whatever locale the user specifies with the environment, rather than trying to specify some nonstandard locale explicitly by name. Remember, different machines might have different sets of locales installed.

7.6 Accessing Locale Information

There are several ways to access locale information. The simplest way is to let the C library itself do the work. Several of the functions in this library implicitly access the locale data, and use what information is provided by the currently selected locale. This is how the locale model is meant to work normally.

As an example, take the `strftime` function, which is meant to nicely format date and time information (see Section 10.4.5 [Formatting Calendar Time], page 291). Part of the standard information contained in the `LC_TIME` category is the names of the months. Instead of requiring the programmer to take care of providing the translations, the `strftime` function does this all by itself. `%A` in the format string is replaced by the appropriate weekday name of the locale currently selected by `LC_TIME`. This is an easy example, and wherever possible, functions do things automatically in this way.

But there are quite often situations when there is simply no function to perform the task, or it is simply not possible to do the work automatically. For these cases it is necessary to access the information in the locale directly. To do this, the C library provides two functions: `localeconv` and `nl_langinfo`. The former is part of ISO C and is therefore portable, but has a brain-damaged interface. The second is part of the Unix interface and is portable in as far as the system follows the Unix standards.

7.6.1 `localeconv`: "It is portable, but ..."

Together with the `setlocale` function, the ISO C people invented the `localeconv` function. It is a masterpiece of poor design. It is expensive to use, not extendable, and not generally usable, since it provides access to only `LC_MONETARY` and `LC_NUMERIC` related information. Nevertheless, if it is applicable to a given situation, it should be used since it is very portable. The function `strfmon` formats monetary amounts according to the selected locale using this information.

`struct lconv *` **localeconv** (`void`) Function

The `localeconv` function returns a pointer to a structure whose components contain information about how numeric and monetary values should be formatted in the current locale.

You should not modify the structure or its contents. The structure might be overwritten by subsequent calls to `localeconv`, or by calls to `setlocale`, but no other function in the library overwrites this value.

struct lconv Data Type

`localeconv`'s return value is of this data type. Its elements are described in the following subsections.

If a member of the structure `struct lconv` has type `char`, and the value is `CHAR_MAX`, it means that the current locale has no value for that parameter.

7.6.1.1 Generic Numeric Formatting Parameters

These are the standard members of `struct lconv`; there may be others.

`char *decimal_point`
`char *mon_decimal_point`

These are the decimal-point separators used in formatting nonmonetary and monetary quantities, respectively. In the 'C' locale, the value of `decimal_point` is `"."`, and the value of `mon_decimal_point` is `""`.

`char *thousands_sep`
`char *mon_thousands_sep`

These are the separators used to delimit groups of digits to the left of the decimal point in formatting nonmonetary and monetary quantities, respectively. In the 'C' locale, both members have a value of `""` (the empty string).

`char *grouping`
`char *mon_grouping`

These are strings that specify how to group the digits to the left of the decimal point. `grouping` applies to nonmonetary quantities, and `mon_grouping` applies to monetary quantities. Use either `thousands_sep` or `mon_thousands_sep` to separate the digit groups.

Each member of these strings is to be interpreted as an integer value of type `char`. Successive numbers (from left to right) give the sizes of successive groups (from right to left, starting at the decimal point.) The last member is either `0`, in which case the previous member is used over and over again for all the remaining groups, or `CHAR_MAX`, in which case there is no more grouping—or, put another way, any remaining digits form one large group without separators.

For example, if `grouping` is `"\04\03\02"`, the correct grouping for the number `123456787654321` is '`12`', '`34`', '`56`', '`78`', '`765`', '`4321`'. This uses a group of four digits at the end, preceded by a group of three digits, preceded by groups of two digits (as many as needed). With a separator of '`,`', the number would be printed as '`12,34,56,78,765,4321`'.

A value of `"\03"` indicates repeated groups of three digits, as normally used in the United States.

In the standard 'C' locale, both `grouping` and `mon_grouping` have a value of `""`. This value specifies no grouping at all.

`char int_frac_digits`
`char frac_digits`
These are small integers indicating how many fractional digits (to the right of the decimal point) should be displayed in a monetary value in international and local formats, respectively. (Most often, both members have the same value.)

In the standard 'C' locale, both of these members have the value `CHAR_MAX`, meaning "unspecified". The ISO standard doesn't say what to do when you find this value; we recommend printing no fractional digits. (This locale also specifies the empty string for `mon_decimal_point`, so printing any fractional digits would be confusing!)

7.6.1.2 Printing the Currency Symbol

These members of the `struct lconv` structure specify how to print the symbol to identify a monetary value—the international analog of '$' for US dollars.

Each country has two standard currency symbols. The *local currency symbol* is used commonly within the country, while the *international currency symbol* is used internationally to refer to that country's currency when it is necessary to indicate the country unambiguously.

For example, many countries use the dollar as their monetary unit, and when dealing with international currencies it's important to specify that one is dealing with (say) Canadian dollars instead of US dollars or Australian dollars. But when the context is known to be Canada, there is no need to make this explicit—dollar amounts are implicitly assumed to be in Canadian dollars.

`char *currency_symbol`
The local currency symbol for the selected locale.

In the standard 'C' locale, this member has a value of `""` (the empty string), meaning "unspecified". The ISO standard doesn't say what to do when you find this value; we recommend you simply print the empty string as you would print any other string pointed to by this variable.

`char *int_curr_symbol`
The international currency symbol for the selected locale.

The value of `int_curr_symbol` should normally consist of a three-letter abbreviation determined by the international standard ISO *4217 Codes for the Representation of Currency and Funds*, followed by a one-character separator (often a space).

In the standard 'C' locale, this member has a value of `""` (the empty string), meaning "unspecified". We recommend you simply print the

empty string as you would print any other string pointed to by this variable.

`char p_cs_precedes`
`char n_cs_precedes`
`char int_p_cs_precedes`
`char int_n_cs_precedes`

These members are `1` if the `currency_symbol` or `int_curr_symbol` strings should precede the value of a monetary amount, or `0` if the strings should follow the value. The `p_cs_precedes` and `int_p_cs_precedes` members apply to positive amounts (or zero), and the `n_cs_precedes` and `int_n_cs_precedes` members apply to negative amounts.

In the standard '`C`' locale, all of these members have a value of `CHAR_MAX`, meaning "unspecified". The ISO standard doesn't say what to do when you find this value. We recommend printing the currency symbol before the amount, which is right for most countries. In other words, treat all nonzero values alike in these members.

The members with the `int_` prefix apply to the `int_curr_symbol` while the other two apply to `currency_symbol`.

`char p_sep_by_space`
`char n_sep_by_space`
`char int_p_sep_by_space`
`char int_n_sep_by_space`

These members are `1` if a space should appear between the `currency_symbol` or `int_curr_symbol` strings and the amount, or `0` if no space should appear. The `p_sep_by_space` and `int_p_sep_by_space` members apply to positive amounts (or zero), and the `n_sep_by_space` and `int_n_sep_by_space` members apply to negative amounts.

In the standard '`C`' locale, all of these members have a value of `CHAR_MAX`, meaning "unspecified". The ISO standard doesn't say what you should do when you find this value; we suggest you treat it as 1 (print a space). In other words, treat all nonzero values alike in these members.

The members with the `int_` prefix apply to the `int_curr_symbol` while the other two apply to `currency_symbol`. There is one special case with the `int_curr_symbol`, though. Since all legal values contain a space at the end the string one either adds this space with printf (if the currency symbol must appear in front and must be separated) or one has to avoid printing this character at all (especially when at the end of the string).

7.6.1.3 Printing the Sign of a Monetary Amount

These members of the `struct lconv` structure specify how to print the sign (if any) of a monetary value.

`char *positive_sign`
`char *negative_sign`

These are strings used to indicate positive (or zero) and negative monetary quantities, respectively.

In the standard 'C' locale, both of these members have a value of `""` (the empty string), meaning "unspecified".

The ISO standard doesn't say what to do when you find this value; we recommend printing `positive_sign` as you find it, even if it is empty. For a negative value, print `negative_sign` as you find it unless both it and `positive_sign` are empty, in which case print '–' instead. (Failing to indicate the sign at all seems rather unreasonable.)

`char p_sign_posn`
`char n_sign_posn`
`char int_p_sign_posn`
`char int_n_sign_posn`

These members are small integers that indicate how to position the sign for nonnegative and negative monetary quantities, respectively. (The string used by the sign is what was specified with `positive_sign` or `negative_sign`.) The possible values are as follows:

`0` The currency symbol and quantity should be surrounded by parentheses.

`1` Print the sign string before the quantity and currency symbol.

`2` Print the sign string after the quantity and currency symbol.

`3` Print the sign string right before the currency symbol.

`4` Print the sign string right after the currency symbol.

`CHAR_MAX` "Unspecified". Both members have this value in the standard 'C' locale.

The ISO standard doesn't say what you should do when the value is `CHAR_MAX`. We recommend you print the sign after the currency symbol.

The members with the `int_` prefix apply to the `int_curr_symbol` while the other two apply to `currency_symbol`.

7.6.2 Pinpoint Access to Locale Data

When writing the *X/Open Portability Guide*,[1] the authors realized that the `localeconv` function is not enough to provide reasonable access to locale information. The information that was meant to be available in the locale (as later specified in the POSIX.1 standard) requires more ways to access it. Therefore, the `nl_langinfo` function was introduced.

`char *` **nl_langinfo** (`nl_item` *item*) Function

The `nl_langinfo` function can be used to access individual elements of the locale categories. Unlike the `localeconv` function, which returns all the information, `nl_langinfo` lets the caller select what information it requires. This is very fast and it is not a problem to call this function multiple times.

A second advantage is that in addition to the numeric and monetary formatting information, information from the `LC_TIME` and `LC_MESSAGES` categories is available.

The type `nl_type` is defined in '`nl_types.h`'. The argument *item* is a numeric value defined in the header '`langinfo.h`'. The X/Open standard defines the following values:

`CODESET` `nl_langinfo` returns a string with the name of the coded character set used in the selected locale.

`ABDAY_1`
`ABDAY_2`
`ABDAY_3`
`ABDAY_4`
`ABDAY_5`
`ABDAY_6`
`ABDAY_7` `nl_langinfo` returns the abbreviated weekday name. `ABDAY_1` corresponds to Sunday.

`DAY_1`
`DAY_2`
`DAY_3`
`DAY_4`
`DAY_5`
`DAY_6`
`DAY_7` Similar to `ABDAY_1`, etc., but here the return value is the unabbreviated weekday name.

[1] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

`ABMON_1`
`ABMON_2`
`ABMON_3`
`ABMON_4`
`ABMON_5`
`ABMON_6`
`ABMON_7`
`ABMON_8`
`ABMON_9`
`ABMON_10`
`ABMON_11`
`ABMON_12`
: The return value is the abbreviated name of the month. `ABMON_1` corresponds to January.

`MON_1`
`MON_2`
`MON_3`
`MON_4`
`MON_5`
`MON_6`
`MON_7`
`MON_8`
`MON_9`
`MON_10`
`MON_11`
`MON_12`
: Similar to `ABMON_1`, etc., but here the month names are not abbreviated. Here the first value `MON_1` also corresponds to January.

`AM_STR`
`PM_STR`
: The return values are strings that can be used in the representation of time as an hour from 1 to 12 plus an a.m. or p.m. specifier.

  In locales that do not use this time representation, these strings might be empty, in which case the a.m./p.m. format cannot be used at all.

`D_T_FMT`
: The return value can be used as a format string for `strftime` to represent time and date in a locale-specific way.

`D_FMT`
: The return value can be used as a format string for `strftime` to represent a date in a locale-specific way.

`T_FMT`
: The return value can be used as a format string for `strftime` to represent time in a locale-specific way.

`T_FMT_AMPM`
: The return value can be used as a format string for `strftime` to represent time in the a.m./p.m. format.

If the a.m./p.m. format does not make any sense for the selected locale, the return value might be the same as the one for `T_FMT`.

`ERA` The return value represents the era used in the current locale.

Most locales do not define this value. An example of a locale that does define this value is the Japanese locale. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the `strftime` functions to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

`ERA_YEAR`
The return value gives the year in the relevant era of the locale. As for `ERA`, it should not be necessary to use this value directly.

`ERA_D_T_FMT`
This return value can be used as a format string for `strftime` to represent dates and times in a locale-specific era-based way.

`ERA_D_FMT`
This return value can be used as a format string for `strftime` to represent a date in a locale-specific era-based way.

`ERA_T_FMT`
This return value can be used as a format string for `strftime` to represent time in a locale-specific era-based way.

`ALT_DIGITS`
The return value is a representation of up to 100 values used to represent the values 0 to 99. As for `ERA`, this value is not intended to be used directly, but instead indirectly through the `strftime` function. When the modifier `O` is used in a format that would otherwise use numerals to represent hours, minutes, seconds, weekdays, months, or weeks, the appropriate value for the locale is used instead.

`INT_CURR_SYMBOL`
This is the same as the value returned by `localeconv` in the `int_curr_symbol` element of the `struct lconv`.

`CURRENCY_SYMBOL`
`CRNCYSTR`
This is the same as the value returned by `localeconv` in the `currency_symbol` element of the `struct lconv`.

`CRNCYSTR` is a deprecated alias still required by Unix98.

`MON_DECIMAL_POINT`
: This is the same as the value returned by `localeconv` in the `mon_decimal_point` element of the `struct lconv`.

`MON_THOUSANDS_SEP`
: This is the same as the value returned by `localeconv` in the `mon_thousands_sep` element of the `struct lconv`.

`MON_GROUPING`
: This is the same as the value returned by `localeconv` in the `mon_grouping` element of the `struct lconv`.

`POSITIVE_SIGN`
: This is the same as the value returned by `localeconv` in the `positive_sign` element of the `struct lconv`.

`NEGATIVE_SIGN`
: This is the same as the value returned by `localeconv` in the `negative_sign` element of the `struct lconv`.

`INT_FRAC_DIGITS`
: This is the same as the value returned by `localeconv` in the `int_frac_digits` element of the `struct lconv`.

`FRAC_DIGITS`
: This is the same as the value returned by `localeconv` in the `frac_digits` element of the `struct lconv`.

`P_CS_PRECEDES`
: This is the same as the value returned by `localeconv` in the `p_cs_precedes` element of the `struct lconv`.

`P_SEP_BY_SPACE`
: This is the same as the value returned by `localeconv` in the `p_sep_by_space` element of the `struct lconv`.

`N_CS_PRECEDES`
: This is the same as the value returned by `localeconv` in the `n_cs_precedes` element of the `struct lconv`.

`N_SEP_BY_SPACE`
: This is the same as the value returned by `localeconv` in the `n_sep_by_space` element of the `struct lconv`.

`P_SIGN_POSN`
: This is the same as the value returned by `localeconv` in the `p_sign_posn` element of the `struct lconv`.

`N_SIGN_POSN`
: This is the same as the value returned by `localeconv` in the `n_sign_posn` element of the `struct lconv`.

`INT_P_CS_PRECEDES`
: This is the same as the value returned by `localeconv` in the `int_p_cs_precedes` element of the `struct lconv`.

`INT_P_SEP_BY_SPACE`
: This is the same as the value returned by `localeconv` in the `int_p_sep_by_space` element of the `struct lconv`.

`INT_N_CS_PRECEDES`
: This is the same as the value returned by `localeconv` in the `int_n_cs_precedes` element of the `struct lconv`.

`INT_N_SEP_BY_SPACE`
: This is the same as the value returned by `localeconv` in the `int_n_sep_by_space` element of the `struct lconv`.

`INT_P_SIGN_POSN`
: This is the same as the value returned by `localeconv` in the `int_p_sign_posn` element of the `struct lconv`.

`INT_N_SIGN_POSN`
: This is the same as the value returned by `localeconv` in the `int_n_sign_posn` element of the `struct lconv`.

`DECIMAL_POINT`
`RADIXCHAR`
: This is the same as the value returned by `localeconv` in the `decimal_point` element of the `struct lconv`.

  The name `RADIXCHAR` is a deprecated alias still used in Unix98.

`THOUSANDS_SEP`
`THOUSEP`
: This is the same as the value returned by `localeconv` in the `thousands_sep` element of the `struct lconv`.

  The name `THOUSEP` is a deprecated alias still used in Unix98.

`GROUPING`
: This is the same as the value returned by `localeconv` in the `grouping` element of the `struct lconv`.

`YESEXPR`
: The return value is a regular expression that can be used with the `regex` function to recognize a positive response to a yes-or-no question. The GNU C Library provides the `rpmatch` function for easier handling in applications.

`NOEXPR`
: The return value is a regular expression that can be used with the `regex` function to recognize a negative response to a yes-or-no question.

`YESSTR` The return value is a locale-specific translation of the positive response to a yes-or-no question.

Using this symbol or its value is deprecated, since it is a very special case of message translation and is better handled by the message translation functions (see Chapter 11 [Message Translation], page 315).

`NOSTR` The return value is a locale-specific translation of the negative response to a yes-or-no question. Use of this symbol or its value is deprecated in the same fashion as `YESSTR`. Instead, message translation should be used (see Chapter 11 [Message Translation], page 315).

The file '`langinfo.h`' defines a many more symbols, but none of them are official. Using them is not portable, and the format of the return values might change. Therefore, we recommend you do not use them.

The return value for any valid argument can be used for `strftime` in all situations (with the possible exception of the a.m. and p.m. time-formatting codes). If the user has not selected any locale for the appropriate category, `nl_langinfo` returns the information from the `"C"` locale. It is therefore possible to use this function as shown in the example below.

If the argument *item* is not valid, a pointer to an empty string is returned.

An example of `nl_langinfo` usage is a function that has to print a given date and time in a locale-specific way. At first you might think that, since `strftime` internally uses the locale information, writing something like the following is enough:

```
size_t
i18n_time_n_data (char *s, size_t len, const struct tm *tp)
{
  return strftime (s, len, "%X %D", tp);
}
```

The format contains no weekday or month names and therefore is internationally usable. This is incorrect. The output produced is something like `"hh:mm:ss MM/DD/YY"`. This format is only recognizable in the United States. Other countries use different formats. Therefore, the function should be rewritten like this:

```
size_t
i18n_time_n_data (char *s, size_t len, const struct tm *tp)
{
  return strftime (s, len, nl_langinfo (D_T_FMT), tp);
}
```

Now it uses the date and time format of the locale selected when the program runs. If the user selects the locale correctly, there should never be a misunderstanding over the time and date format.

7.7 A Dedicated Function to Format Numbers

We have seen that the structure returned by `localeconv` as well as the values given to `nl_langinfo` allow you to retrieve the various pieces of locale-specific information to format numbers and monetary amounts. We have also seen that the underlying rules are quite complex.

Therefore, the X/Open standards introduce a function that uses such locale information, making it easier for the user to format numbers according to these rules.

`ssize_t` **strfmon** (`char *`*s*`, size_t` *maxsize*`, const char *`*format*`, ...`) Function

The `strfmon` function is similar to the `strftime` function in that it takes a buffer, its size, a format string and values to write into the buffer as text in a form specified by the format string. Like `strftime`, the function also returns the number of bytes written into the buffer.

There are two differences: `strfmon` can take more than one argument, and the format specification is different. Like `strftime`, the format string consists of normal text, which is output as is, and format specifiers, which are indicated by a '`%`'. Immediately after the '`%`', you can optionally specify various flags and formatting information before the main formatting character, in a similar way to `printf`:

- Immediately following the '`%`' there can be one or more of the following flags:

  '=*f*' The single-byte character *f* is used for this field as the numeric fill character. By default, this character is a space character. Filling with this character is only performed if a left precision is specified. It is not just to fill to the given field width.

  '^' The number is printed without grouping the digits according to the rules of the current locale. By default, grouping is enabled.

  '+', '(' At most one of these flags can be used. They select which format to represent the sign of a currency amount. By default, and if '+' is given, the locale equivalent of +/− is used. If '(' is given, negative amounts are enclosed in parentheses. The exact format is determined by the values of the `LC_MONETARY` category of the locale selected at program run time.

  '!' The output will not contain the currency symbol.

  '−' The output will be formatted left-justified instead of right-justified if it does not fill the entire field width.

The next part of a specification is an optional field width. If no width is specified, 0 is taken. During output, the function first determines how much space is required. If it requires at least as many characters as given by the field width, it

is output using as much space as necessary. Otherwise, it is extended to use the full width by filling with the space character. The presence or absence of the '-' flag determines the side at which such padding occurs. If present, the spaces are added at the right, making the output left-justified, and vice versa.

So far the format looks familiar, being similar to the `printf` and `strftime` formats. However, the next two optional fields introduce something new. The first one is a '`#`' character followed by a decimal digit string. The value of the digit string specifies the number of *digit* positions to the left of the decimal point (or equivalent). This does *not* include the grouping character when the '`^`' flag is not given. If the space needed to print the number does not fill the whole width, the field is padded at the left side with the fill character, which can be selected using the '`=`' flag and by default is a space. For example, if the field width is selected as 6 and the number is 123, the fill character is '`*`', and the result will be '`***123`'.

The second optional field starts with a '`.`' (period) and consists of another decimal digit string. Its value describes the number of characters printed after the decimal point. The default is selected from the current locale (`frac_digits`, `int_frac_digits`; see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187). If the exact representation needs more digits than given by the field width, the displayed value is rounded. If the number of fractional digits is selected to be zero, no decimal point is printed.

As a GNU extension, the `strfmon` implementation in the GNU libc allows an optional '`L`' next as a format modifier. If this modifier is given, the argument is expected to be a `long double` instead of a `double` value.

Finally, the last component is a format specifier. There are three specifiers defined:

'`i`' Use the locale's rules for formatting an international currency value.

'`n`' Use the locale's rules for formatting a national currency value.

'`%`' Place a '`%`' in the output. There must be no flag, width specifier or modifier given; only '`%%`' is allowed.

As for `printf`, the function reads the format string from left to right and uses the values passed to the function following the format string. The values are expected to be either of type `double` or `long double`, depending on the presence of the modifier '`L`'. The result is stored in the buffer pointed to by *s*. At most *maxsize* characters are stored.

The return value of the function is the number of characters stored in *s*, including the terminating `NULL` byte. If the number of characters stored would exceed *maxsize*, the function returns -1 and the content of the buffer *s* is unspecified. In this case `errno` is set to `E2BIG`.

A few examples should make clear how the function works. It is assumed that all the following pieces of code are executed in a program that uses the US locale (`en_US`). The simplest form of the format is this:

```
strfmon (buf, 100, "@%n@%n@%n@", 123.45, -567.89, 12345.678);
```

The output produced is

```
"@$123.45@-$567.89@$12,345.68@"
```

We can notice several things here. First, the widths of the output numbers are different, since we have not specified a width in the format string. Second, the third number is printed using thousands separators. The thousands separator for the `en_US` locale is a comma. The number is also rounded. The number .678 is rounded to .68 since the format does not specify a precision and the default value in the locale is 2. Finally, note that the national currency symbol is printed since '`%n`' was used, not '`i`'. The next example shows how we can align the output:

```
strfmon (buf, 100, "@%=*11n@%=*11n@%=*11n@", 123.45, -567.89, 12345.678);
```

The output this time is

```
"@    $123.45@   -$567.89@ $12,345.68@"
```

Two things stand out. First, all fields have the same width (eleven characters) since this is the width given in the format, and since no number required more characters to be printed. The second important point is that the fill character is not used. This is correct since the white space was not used to achieve a precision given by a '`#`' modifier, but instead to fill to the given width. The difference becomes obvious if we now add a width specification:

```
strfmon (buf, 100, "@%=*11#5n@%=*11#5n@%=*11#5n@",
         123.45, -567.89, 12345.678);
```

The output is

```
"@ $***123.45@-$***567.89@ $12,456.68@"
```

Here we can see that all the currency symbols are now aligned, and that the space between the currency sign and the number is filled with the selected fill character. Although the width is selected to be 5 and 123.45 has 3 digits left of the decimal point, the space is filled with 3 asterisks. This is correct since, as explained above, the width does not include the positions used to store thousands separators. One last example should explain the remaining functionality:

```
strfmon (buf, 100, "@%=0(16#5.3i@%=0(16#5.3i@%=0(16#5.3i@",
         123.45, -567.89, 12345.678);
```

This rather complex format string produces the following output:

```
"@ USD 000123,450 @(USD 000567.890)@ USD 12,345.678 @"
```

The most noticeable change is the alternative way of representing negative numbers. In financial circles this is often done using parentheses, and this is what the '`(`' flag selected. The fill character is now '`0`'. This '`0`' character is not regarded as a numeric zero, and therefore the first and second numbers are not printed using a thousands separator. Since we used the format specifier '`i`' instead of '`n`', the international form of the currency symbol is used. This is a four-letter string, in this case `"USD "`. The last point is that since the precision right of the decimal point is selected to be three, the first and second numbers are printed with an extra zero at the end and the third number is printed without rounding.

7.8 Yes-or-No Questions

Some non-GUI programs ask a yes-or-no question. If the messages (especially the questions) are translated into foreign languages, be sure that you localize the answers too. It would be very bad habit to ask a question in one language and request the answer in another, often English.

The GNU C Library contains `rpmatch` to give applications easy access to the corresponding locale definitions.

`int` **rpmatch** (`const char *`*response*) Function

The function `rpmatch` checks the string in *response* to see whether it is a correct yes-or-no answer and if yes, which one. The check uses the `YESEXPR` and `NOEXPR` data in the `LC_MESSAGES` category of the currently selected locale. The return value is as follows:

`1` The user entered an affirmative answer.

`0` The user entered a negative answer.

`-1` The answer matched neither the `YESEXPR` nor the `NOEXPR` regular expression.

This function is not standardized, but is available in GNU libc and at least also in the IBM AIX library.

This function would normally be used like this:

```
...
/* Use a safe default.  */
_Bool doit = false;

fputs (gettext ("Do you really want to do this? "), stdout);
fflush (stdout);
/* Prepare the getline call.  */
line = NULL;
len = 0;
while (getline (&line, &len, stdout) >= 0)
  {
    /* Check the response.  */
    int res = rpmatch (line);
    if (res >= 0)
      {
        /* We got a definitive answer.  */
        if (res > 0)
          doit = true;
        break;
      }
  }
```

```
/* Free what getline allocated. */
free (line);
```

Note that the loop continues until a read error is detected or until a definitive (positive or negative) answer is read.

8 Mathematics

This chapter contains information about functions for performing mathematical computations, such as trigonometric functions. Most of these functions have prototypes declared in the header file 'math.h'. The complex-valued functions are defined in 'complex.h'.

All mathematical functions that take a floating-point argument have three variants, one each for double, float and long double arguments. The double versions are mostly defined in ISO C89. The float and long double versions are from the numeric extensions to C included in ISO C99.

Which of the three versions of a function should be used depends on the situation. For most calculations, the float functions are the fastest. On the other hand, the long double functions have the highest precision. double is somewhere in between. It is usually wise to pick the narrowest type that can accommodate your data. Not all machines have a distinct long double type; it may be the same as double.

8.1 Predefined Mathematical Constants

The header 'math.h' defines several useful mathematical constants. All values are defined as preprocessor macros starting with M_. The values provided are

M_E The base of natural logarithms

M_LOG2E The logarithm to base 2 of M_E

M_LOG10E
The logarithm to base 10 of M_E

M_LN2 The natural logarithm of 2

M_LN10 The natural logarithm of 10

M_PI Pi, the ratio of a circle's circumference to its diameter

M_PI_2 Pi divided by two

M_PI_4 Pi divided by four

M_1_PI The reciprocal of pi (1/pi)

M_2_PI Two times the reciprocal of pi

M_2_SQRTPI
Two times the reciprocal of the square root of pi

M_SQRT2 The square root of two

`M_SQRT1_2`
: The reciprocal of the square root of two (also the square root of 1/2)

These constants come from the Unix98 standard and were also available in 4.4BSD; therefore they are only defined if `_BSD_SOURCE` or `_XOPEN_SOURCE=500` or a more general feature-select macro is defined. The default set of features includes these constants (see Section 1.3.4 [Feature-Test Macros], page 8).

All values are of type `double`. As an extension, the GNU C library also defines these constants with type `long double`. The `long double` macros have a lowercase '`l`' appended to their names: `M_El`, `M_PIl`, and so forth. These are only available if `_GNU_SOURCE` is defined.

Some programs use a constant named `PI` that has the same value as `M_PI`. This constant is not standard; it may have appeared in some old AT&T headers, and is mentioned in Stroustrup's book on C++. It infringes on the user's name space, so the GNU C Library does not define it. Fixing programs written to expect it is simple; replace `PI` with `M_PI` throughout, or put '`-DPI=M_PI`' on the compiler command line.

8.2 Trigonometric Functions

These are the familiar `sin`, `cos` and `tan` functions. The arguments to all of these functions are in units of radians; recall that pi radians equals 180 degrees.

The math library normally defines `M_PI` to a `double` approximation of pi. If strict ISO and/or POSIX compliance are requested, this constant is not defined, but you can easily define it yourself:

```
#define M_PI 3.14159265358979323846264338327
```

You can also compute the value of pi with the expression `acos (-1.0)`.

`double` **sin** (`double` *x*) Function
`float` **sinf** (`float` *x*) Function
`long double` **sinl** (`long double` *x*) Function
: These functions return the sine of *x*, where *x* is given in radians. The return value is in the range `-1` to `1`.

`double` **cos** (`double` *x*) Function
`float` **cosf** (`float` *x*) Function
`long double` **cosl** (`long double` *x*) Function
: These functions return the cosine of *x*, where *x* is given in radians. The return value is in the range `-1` to `1`.

`double` **tan** (`double` *x*) Function
`float` **tanf** (`float` *x*) Function
`long double` **tanl** (`long double` *x*) Function
: These functions return the tangent of *x*, where *x* is given in radians.

Mathematically, the tangent function has singularities at odd multiples of pi/2. If the argument *x* is too close to one of these singularities, `tan` will signal overflow.

In many applications where `sin` and `cos` are used, the sine and cosine of the same angle are needed at the same time. It is more efficient to compute them simultaneously, so the library provides a function to do that.

`void` **sincos** (`double` *x*, `double` `*`*sinx*, `double` `*`*cosx*) Function
`void` **sincosf** (`float` *x*, `float` `*`*sinx*, `float` `*`*cosx*) Function
`void` **sincosl** (`long double` *x*, `long double` `*`*sinx*, `long double` `*`*cosx*) Function

These functions return the sine of *x* in `*`*sinx* and the cosine of *x* in `*`*cos*, where *x* is given in radians. Both values, `*`*sinx* and `*`*cosx*, are in the range of `-1` to `1`.

This function is a GNU extension. Portable programs should be prepared to cope with its absence.

ISO C99 defines variants of the trig functions that work on complex numbers. The GNU C Library provides these functions, but they are only useful if your compiler supports the new complex types defined by the standard. (As of this writing, GCC supports complex numbers, but there are bugs in the implementation.)

`complex double` **csin** (`complex double` *z*) Function
`complex float` **csinf** (`complex float` *z*) Function
`complex long double` **csinl** (`complex long double` *z*) Function

These functions return the complex sine of *z*. The mathematical definition of the complex sine is

$$\sin(z) = \frac{1}{2i}(e^{zi} - e^{-zi})$$

`complex double` **ccos** (`complex double` *z*) Function
`complex float` **ccosf** (`complex float` *z*) Function
`complex long double` **ccosl** (`complex long double` *z*) Function

These functions return the complex cosine of *z*. The mathematical definition of the complex cosine is

$$\cos(z) = \frac{1}{2}(e^{zi} + e^{-zi})$$

`complex double` **ctan** (`complex double` *z*) Function
`complex float` **ctanf** (`complex float` *z*) Function
`complex long double` **ctanl** (`complex long double` *z*) Function

These functions return the complex tangent of *z*. The mathematical definition of the complex tangent is

$$\tan(z) = -i \cdot \frac{e^{zi} - e^{-zi}}{e^{zi} + e^{-zi}}$$

The complex tangent has poles at $pi/2 + 2n$, where n is an integer. `ctan` may signal overflow if *z* is too close to a pole.

8.3 Inverse Trigonometric Functions

These are the usual arc sine, arc cosine and arc tangent functions, which are the inverses of the sine, cosine and tangent functions respectively.

`double` **asin** (`double` *x*) Function
`float` **asinf** (`float` *x*) Function
`long double` **asinl** (`long double` *x*) Function

These functions compute the arc sine of *x*—that is, the value whose sine is *x*. The value is in units of radians. Mathematically, there are infinitely many such values; the one actually returned is the one between `-pi/2` and `pi/2` (inclusive).

The arc sine function is defined mathematically only over the domain `-1` to `1`. If *x* is outside the domain, `asin` signals a domain error.

`double` **acos** (`double` *x*) Function
`float` **acosf** (`float` *x*) Function
`long double` **acosl** (`long double` *x*) Function

These functions compute the arc cosine of *x*—the value whose cosine is *x*. The value is in units of radians. Mathematically, there are infinitely many such values; the one actually returned is the one between `0` and `pi` (inclusive).

The arc cosine function is defined mathematically only over the domain `-1` to `1`. If *x* is outside the domain, `acos` signals a domain error.

`double` **atan** (`double` *x*) Function
`float` **atanf** (`float` *x*) Function
`long double` **atanl** (`long double` *x*) Function

These functions compute the arc tangent of *x*—the value whose tangent is *x*. The value is in units of radians. Mathematically, there are infinitely many such values; the one actually returned is the one between `-pi/2` and `pi/2` (inclusive).

`double` **atan2** (`double` *y*, `double` *x*) Function
`float` **atan2f** (`float` *y*, `float` *x*) Function
`long double` **atan2l** (`long double` *y*, `long double` *x*) Function

This function computes the arc tangent of *y*/*x*, but the signs of both arguments are used to determine the quadrant of the result, and *x* is permitted to be zero. The return value is given in radians and is in the range `-pi` to `pi`, inclusive.

If *x* and *y* are coordinates of a point in the plane, `atan2` returns the signed angle between the line from the origin to that point and the x-axis. Thus, `atan2` is useful for converting Cartesian coordinates to polar coordinates. To compute the radial coordinate, use `hypot` (see Section 8.4 [Exponentiation and Logarithms], page 207).

If both *x* and *y* are zero, `atan2` returns zero.

ISO C99 defines complex versions of the inverse trig functions.

`complex double` **casin** (`complex double` *z*) Function
`complex float` **casinf** (`complex float` *z*) Function
`complex long double` **casinl** (`complex long double` *z*) Function
These functions compute the complex arc sine of *z*—the value whose sine is *z*. The value returned is in radians.

Unlike the real-valued functions, `casin` is defined for all values of *z*.

`complex double` **cacos** (`complex double` *z*) Function
`complex float` **cacosf** (`complex float` *z*) Function
`complex long double` **cacosl** (`complex long double` *z*) Function
These functions compute the complex arc cosine of *z*—the value whose cosine is *z*. The value returned is in radians.

Unlike the real-valued functions, `cacos` is defined for all values of *z*.

`complex double` **catan** (`complex double` *z*) Function
`complex float` **catanf** (`complex float` *z*) Function
`complex long double` **catanl** (`complex long double` *z*) Function
These functions compute the complex arc tangent of *z*—the value whose tangent is *z*. The value is in units of radians.

8.4 Exponentiation and Logarithms

`double` **exp** (`double` *x*) Function
`float` **expf** (`float` *x*) Function
`long double` **expl** (`long double` *x*) Function
These functions compute `e` (the base of natural logarithms) raised to the power *x*.

If the magnitude of the result is too large to be representable, `exp` signals overflow.

`double` **exp2** (`double` *x*) Function
`float` **exp2f** (`float` *x*) Function
`long double` **exp2l** (`long double` *x*) Function
These functions compute `2` raised to the power *x*. Mathematically, `exp2 (x)` is the same as `exp (x * log (2))`.

`double` **exp10** (`double` *x*) Function
`float` **exp10f** (`float` *x*) Function
`long double` **exp10l** (`long double` *x*) Function
`double` **pow10** (`double` *x*) Function
`float` **pow10f** (`float` *x*) Function
`long double` **pow10l** (`long double` *x*) Function

These functions compute `10` raised to the power *x*. Mathematically, `exp10 (x)` is the same as `exp (x * log (10))`.

These functions are GNU extensions. The name `exp10` is preferred, since it is analogous to `exp` and `exp2`.

`double` **log** (`double` *x*) Function
`float` **logf** (`float` *x*) Function
`long double` **logl** (`long double` *x*) Function

These functions compute the natural logarithm of *x*. `exp (log (x))` equals *x*, exactly in mathematics and approximately in C.

If *x* is negative, `log` signals a domain error. If *x* is zero, it returns negative infinity; if *x* is too close to zero, it may signal overflow.

`double` **log10** (`double` *x*) Function
`float` **log10f** (`float` *x*) Function
`long double` **log10l** (`long double` *x*) Function

These functions return the base-10 logarithm of *x*. `log10 (x)` equals `log (x) / log (10)`.

`double` **log2** (`double` *x*) Function
`float` **log2f** (`float` *x*) Function
`long double` **log2l** (`long double` *x*) Function

These functions return the base-2 logarithm of *x*. `log2 (x)` equals `log (x) / log (2)`.

`double` **logb** (`double` *x*) Function
`float` **logbf** (`float` *x*) Function
`long double` **logbl** (`long double` *x*) Function

These functions extract the exponent of *x* and return it as a floating-point value. If `FLT_RADIX` is two, `logb` is equal to `floor (log2 (x))`, except it's probably faster.

If *x* is de-normalized, `logb` returns the exponent *x* would have if it were normalized. If *x* is infinity (positive or negative), `logb` returns ∞. If *x* is zero, `logb` returns ∞. It does not signal.

`int` **ilogb** (`double` *x*) Function
`int` **ilogbf** (`float` *x*) Function
`int` **ilogbl** (`long double` *x*) Function

These functions are equivalent to the corresponding `logb` functions, except that they return signed integer values.

Since integers cannot represent infinity and NaN, `ilogb` instead returns an integer that can't be the exponent of a normal floating-point number. '`math.h`' defines constants so you can check for this.

`int` **FP_ILOGB0** Macro

`ilogb` returns this value if its argument is `0`. The numeric value is either `INT_MIN` or `-INT_MAX`.

This macro is defined in ISO C99.

`int` **FP_ILOGBNAN** Macro

`ilogb` returns this value if its argument is `NaN`. The numeric value is either `INT_MIN` or `INT_MAX`.

This macro is defined in ISO C99.

These values are system specific. They might even be the same. The proper way to test the result of `ilogb` is as follows:

```
i = ilogb (f);
if (i == FP_ILOGB0 || i == FP_ILOGBNAN)
  {
    if (isnan (f))
      {
        /* Handle NaN.  */
      }
    else if (f  == 0.0)
      {
        /* Handle 0.0.  */
      }
    else
      {
        /* Some other value with large exponent,
           perhaps +Inf.  */
      }
  }
```

`double` **pow** (`double` *base*, `double` *power*) Function
`float` **powf** (`float` *base*, `float` *power*) Function
`long double` **powl** (`long double` *base*, `long double` *power*) Function

These are general exponentiation functions, returning *base* raised to *power*.

Mathematically, `pow` would return a complex number when *base* is negative and *power* is not an integral value. `pow` can't do that, so instead it signals a domain error. `pow` may also underflow or overflow the destination type.

`double` **sqrt** (`double` *x*) Function
`float` **sqrtf** (`float` *x*) Function
`long double` **sqrtl** (`long double` *x*) Function
These functions return the nonnegative square root of *x*.

If *x* is negative, `sqrt` signals a domain error. Mathematically, it should return a complex number.

`double` **cbrt** (`double` *x*) Function
`float` **cbrtf** (`float` *x*) Function
`long double` **cbrtl** (`long double` *x*) Function
These functions return the cube root of *x*. They cannot fail; every representable real value has a representable real cube root.

`double` **hypot** (`double` *x*, `double` *y*) Function
`float` **hypotf** (`float` *x*, `float` *y*) Function
`long double` **hypotl** (`long double` *x*, `long double` *y*) Function
These functions return `sqrt (`*x*`*`*x* + *y*`*`*y*`)`. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller (see also the function `cabs` in Section 9.8.1 [Absolute Value], page 258).

`double` **expm1** (`double` *x*) Function
`float` **expm1f** (`float` *x*) Function
`long double` **expm1l** (`long double` *x*) Function
These functions return a value equivalent to `exp (`*x*`) - 1`. They are computed in a way that is accurate even if *x* is near zero—a case where `exp (`*x*`) - 1` would be inaccurate owing to subtraction of two numbers that are nearly equal.

`double` **log1p** (`double` *x*) Function
`float` **log1pf** (`float` *x*) Function
`long double` **log1pl** (`long double` *x*) Function
These functions returns a value equivalent to `log (1 +` *x*`)`. They are computed in a way that is accurate even if *x* is near zero.

ISO C99 defines complex variants of some of the exponentiation and logarithm functions.

`complex double` **cexp** (`complex double` *z*) Function
`complex float` **cexpf** (`complex float` *z*) Function
`complex long double` **cexpl** (`complex long double` *z*) Function
These functions return `e` (the base of natural logarithms) raised to the power of *z*. Mathematically, this corresponds to the value:

$$\exp(z) = e^z = e^{\mathrm{Re}\, z}(\cos(\mathrm{Im}\, z) + i \sin(\mathrm{Im}\, z))$$

`complex double` **clog** (`complex double` *z*) Function
`complex float` **clogf** (`complex float` *z*) Function
`complex long double` **clogl** (`complex long double` *z*) Function
These functions return the natural logarithm of *z*. Mathematically, this corresponds to the value:

$$\log(z) = \log |z| + i \arg z$$

`clog` has a pole at 0, and will signal overflow if *z* equals or is very close to 0. It is well-defined for all other values of *z*.

`complex double` **clog10** (`complex double` *z*) Function
`complex float` **clog10f** (`complex float` *z*) Function
`complex long double` **clog10l** (`complex long double` *z*) Function
These functions return the base-10 logarithm of the complex value *z*. Mathematically, this corresponds to the value:

$$\log_{10}(z) = \log_{10} |z| + i \arg z$$

These functions are GNU extensions.

`complex double` **csqrt** (`complex double` *z*) Function
`complex float` **csqrtf** (`complex float` *z*) Function
`complex long double` **csqrtl** (`complex long double` *z*) Function
These functions return the complex square root of the argument *z*. Unlike the real-valued functions, they are defined for all values of *z*.

`complex double` **cpow** (`complex double` *base*, `complex double` *power*) Function
`complex float` **cpowf** (`complex float` *base*, `complex float` *power*) Function
`complex long double` **cpowl** (`complex long double` *base*, `complex long double` *power*) Function
These functions return *base* raised to the power of *power*. This is equivalent to `cexp (y * clog (x))`

8.5 Hyperbolic Functions

The functions in this section are related to the exponential functions (see Section 8.4 [Exponentiation and Logarithms], page 207).

`double` **sinh** (`double` *x*) Function
`float` **sinhf** (`float` *x*) Function
`long double` **sinhl** (`long double` *x*) Function
These functions return the hyperbolic sine of *x*, defined mathematically as `(exp (x) - exp (-x)) / 2`. They may signal overflow if *x* is too large.

`double` **cosh** (`double` *x*) Function
`float` **coshf** (`float` *x*) Function
`long double` **coshl** (`long double` *x*) Function
These functions return the hyperbolic cosine of *x*, defined mathematically as `(exp (x) + exp (-x)) / 2`. They may signal overflow if *x* is too large.

`double` **tanh** (`double` *x*) Function
`float` **tanhf** (`float` *x*) Function
`long double` **tanhl** (`long double` *x*) Function
These functions return the hyperbolic tangent of *x*, defined mathematically as `sinh (x) / cosh (x)`. They may signal overflow if *x* is too large.

There are counterparts for the hyperbolic functions that take complex arguments.

`complex double` **csinh** (`complex double` *z*) Function
`complex float` **csinhf** (`complex float` *z*) Function
`complex long double` **csinhl** (`complex long double` *z*) Function
These functions return the complex hyperbolic sine of *z*, defined mathematically as `(exp (z) - exp (-z)) / 2`.

`complex double` **ccosh** (`complex double` *z*) Function
`complex float` **ccoshf** (`complex float` *z*) Function
`complex long double` **ccoshl** (`complex long double` *z*) Function
These functions return the complex hyperbolic cosine of *z*, defined mathematically as `(exp (z) + exp (-z)) / 2`.

`complex double` **ctanh** (`complex double` *z*) Function
`complex float` **ctanhf** (`complex float` *z*) Function
`complex long double` **ctanhl** (`complex long double` *z*) Function
These functions return the complex hyperbolic tangent of *z*, defined mathematically as `csinh (z) / ccosh (z)`.

`double` **asinh** (`double` *x*) Function
`float` **asinhf** (`float` *x*) Function
`long double` **asinhl** (`long double` *x*) Function
These functions return the inverse hyperbolic sine of *x*—the value whose hyperbolic sine is *x*.

`double` **acosh** (`double` *x*) Function
`float` **acoshf** (`float` *x*) Function
`long double` **acoshl** (`long double` *x*) Function
These functions return the inverse hyperbolic cosine of *x*—the value whose hyperbolic cosine is *x*. If *x* is less than `1`, `acosh` signals a domain error.

`double` **atanh** (`double` *x*) Function
`float` **atanhf** (`float` *x*) Function
`long double` **atanhl** (`long double` *x*) Function
These functions return the inverse hyperbolic tangent of *x*—the value whose hyperbolic tangent is *x*. If the absolute value of *x* is greater than `1`, `atanh` signals a domain error; if it is equal to 1, `atanh` returns infinity.

`complex double` **casinh** (`complex double` *z*) Function
`complex float` **casinhf** (`complex float` *z*) Function
`complex long double` **casinhl** (`complex long double` *z*) Function
These functions return the inverse complex hyperbolic sine of *z*—the value whose complex hyperbolic sine is *z*.

`complex double` **cacosh** (`complex double` *z*) Function
`complex float` **cacoshf** (`complex float` *z*) Function
`complex long double` **cacoshl** (`complex long double` *z*) Function
These functions return the inverse complex hyperbolic cosine of *z*—the value whose complex hyperbolic cosine is *z*. Unlike the real-valued functions, there are no restrictions on the value of *z*.

`complex double` **catanh** (`complex double` *z*) Function
`complex float` **catanhf** (`complex float` *z*) Function
`complex long double` **catanhl** (`complex long double` *z*) Function
These functions return the inverse complex hyperbolic tangent of *z*—the value whose complex hyperbolic tangent is *z*. Unlike the real-valued functions, there are no restrictions on the value of *z*.

8.6 Special Functions

These are some more exotic mathematical functions that are sometimes useful. Currently they only have real-valued versions.

`double` **erf** (`double` *x*) Function
`float` **erff** (`float` *x*) Function
`long double` **erfl** (`long double` *x*) Function
`erf` returns the error function of *x*. The error function is defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \cdot \int_0^x e^{-t^2} \mathrm{d}t$$

`double` **erfc** (`double` *x*) Function
`float` **erfcf** (`float` *x*) Function
`long double` **erfcl** (`long double` *x*) Function
`erfc` returns `1.0 - erf(`*x*`)`, but computed in a fashion that avoids round-off error when *x* is large.

`double` **lgamma** (`double` *x*) Function
`float` **lgammaf** (`float` *x*) Function
`long double` **lgammal** (`long double` *x*) Function
`lgamma` returns the natural logarithm of the absolute value of the gamma function of *x*. The gamma function is defined as

$$\mathrm{lgamma}(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t$$

The sign of the gamma function is stored in the global variable *signgam*, which is declared in '`math.h`'. It is `1` if the intermediate result was positive or zero, or `-1` if it was negative.

To compute the real gamma function, you can use the `tgamma` function or you can compute the values as follows:

```
lgam = lgamma(x);
gam  = signgam*exp(lgam);
```

The gamma function has singularities at the nonpositive integers. `lgamma` will raise the zero divide exception if evaluated at a singularity.

`double` **lgamma_r** (`double` *x*`, int *`*signp*) Function
`float` **lgammaf_r** (`float` *x*`, int *`*signp*) Function
`long double` **lgammal_r** (`long double` *x*`, int *`*signp*) Function
`lgamma_r` is just like `lgamma`, but it stores the sign of the intermediate result in the variable pointed to by *signp* instead of in the *signgam* global. This means it is reentrant.

`double` **gamma** (`double` *x*) Function
`float` **gammaf** (`float` *x*) Function
`long double` **gammal** (`long double` *x*) Function
These functions exist for compatibility reasons. They are equivalent to `lgamma`, `lgammaf` and `lgammal`. It is better to use `lgamma` for two reasons. The name better reflects the actual computation, and `lgamma` is standardized in ISO C99 while `gamma` is not.

`double` **tgamma** (`double` *x*) Function
`float` **tgammaf** (`float` *x*) Function
`long double` **tgammal** (`long double` *x*) Function
`tgamma` applies the gamma function to *x*. The gamma function is defined as:

$$\text{tgamma}(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t$$

This function was introduced in ISO C99.

`double` **j0** (`double` *x*) Function
`float` **j0f** (`float` *x*) Function
`long double` **j0l** (`long double` *x*) Function
`j0` returns the Bessel function of the first kind of order 0 of *x*. It may signal underflow if *x* is too large.

`double` **j1** (`double` *x*) Function
`float` **j1f** (`float` *x*) Function
`long double` **j1l** (`long double` *x*) Function
`j1` returns the Bessel function of the first kind of order 1 of *x*. It may signal underflow if *x* is too large.

`double` **jn** (`int n, double` *x*) Function
`float` **jnf** (`int n, float` *x*) Function
`long double` **jnl** (`int n, long double` *x*) Function
`jn` returns the Bessel function of the first kind of order *n* of *x*. It may signal underflow if *x* is too large.

`double` **y0** (`double` *x*) Function
`float` **y0f** (`float` *x*) Function
`long double` **y0l** (`long double` *x*) Function
`y0` returns the Bessel function of the second kind of order 0 of *x*. It may signal underflow if *x* is too large. If *x* is negative, `y0` signals a domain error; if it is zero, `y0` signals overflow and returns $-\infty$.

`double` **y1** (`double` *x*) Function
`float` **y1f** (`float` *x*) Function
`long double` **y1l** (`long double` *x*) Function
`y1` returns the Bessel function of the second kind of order 1 of *x*. It may signal underflow if *x* is too large. If *x* is negative, `y1` signals a domain error; if it is zero, `y1` signals overflow and returns $-\infty$.

`double` **yn** (`int n, double` *x*) Function
`float` **ynf** (`int n, float` *x*) Function
`long double` **ynl** (`int n, long double` *x*) Function
`yn` returns the Bessel function of the second kind of order *n* of *x*. It may signal underflow if *x* is too large. If *x* is negative, `yn` signals a domain error; if it is zero, `yn` signals overflow and returns $-\infty$.

8.7 Known Maximum Errors in Math Functions

This section lists the known errors of the functions in the math library. Errors are measured in "units of the last place". This is a measure for the relative error. For a number z with the representation $d.d\ldots d{\cdot}2^e$ (we assume IEEE floating-point numbers with base 2), the ULP is represented by:

$$\frac{|d.d\ldots d - (z/2^e)|}{2^{p-1}}$$

where p is the number of bits in the mantissa of the floating-point number representation. Ideally the error for all functions is always less than 0.5ulps. Using rounding bits, this is also possible and normally implemented for the basic operations. To achieve the same for the complex math functions requires a lot more work and this has not yet been done.

Therefore many of the functions in the math library have errors. The table lists the maximum error for each function that is exposed by one of the existing tests in the test suite. The table tries to cover as much as possible and list the actual maximum error (or at least a ballpark figure) but this is often not achieved due to the large search space.

The table lists the ULP values for different architectures. Different architectures have different results since their hardware support for floating-point operations varies and also the existing hardware support is different.

Function	*Alpha*	*ARM*	*Generic*	*hppa/fpu*	*ix86*
acosf	-	-	-	-	-
acos	-	-	-	-	-
acosl	-	-	-	-	622
acoshf	-	-	-	-	-
acosh	-	-	-	-	-
acoshl	-	-	-	-	-
asinf	-	2	-	-	-
asin	-	1	-	-	-
asinl	-	-	-	-	1
asinhf	-	-	-	-	-
asinh	-	-	-	-	-
asinhl	-	-	-	-	-
atanf	-	-	-	-	-
atan	-	-	-	-	-
atanl	-	-	-	-	-
atanhf	1	-	-	1	-
atanh	-	1	-	-	-
atanhl	-	-	-	-	1
atan2f	3	-	-	3	-
atan2	-	-	-	-	-
atan2l	-	-	-	-	-
cabsf	-	1	-	-	-
cabs	-	1	-	-	-
cabsl	-	-	-	-	-
cacosf	-	1 + i 1	-	-	0 + i 1
cacos	-	1 + i 0	-	-	-
cacosl	-	-	-	-	0 + i 2
cacoshf	7 + i 3	7 + i 3	-	7 + i 3	9 + i 4
cacosh	1 + i 1	1 + i 1	-	1 + i 1	1 + i 1
cacoshl	-	-	-	-	6 + i 1
cargf	-	-	-	-	-
carg	-	-	-	-	-
cargl	-	-	-	-	-
casinf	1 + i 0	2 + i 1	-	1 + i 0	1 + i 1
casin	1 + i 0	3 + i 0	-	1 + i 0	1 + i 0
casinl	-	-	-	-	2 + i 2
casinhf	1 + i 6	1 + i 6	-	1 + i 6	1 + i 6
casinh	5 + i 3	5 + i 3	-	5 + i 3	5 + i 3
casinhl	-	-	-	-	5 + i 5
catanf	4 + i 1	4 + i 1	-	4 + i 1	0 + i 1
catan	0 + i 1	0 + i 1	-	0 + i 1	0 + i 1
catanl	-	-	-	-	-
catanhf	0 + i 6	1 + i 6	-	0 + i 6	1 + i 0
catanh	4 + i 0	4 + i 1	-	4 + i 0	2 + i 0
catanhl	-	-	-	-	1 + i 0

cbrtf	-	-	-	-	-
cbrt	1	1	-	1	-
cbrtl	-	-	-	-	1
ccosf	1 + i 1	0 + i 1	-	1 + i 1	0 + i 1
ccos	1 + i 0	1 + i 1	-	1 + i 0	1 + i 0
ccosl	-	-	-	-	1 + i 1
ccoshf	1 + i 1	1 + i 1	-	1 + i 1	1 + i 1
ccosh	1 + i 0	1 + i 1	-	1 + i 0	1 + i 1
ccoshl	-	-	-	-	0 + i 1
ceilf	-	-	-	-	-
ceil	-	-	-	-	-
ceill	-	-	-	-	-
cexpf	1 + i 1	1 + i 1	-	1 + i 1	-
cexp	-	1 + i 0	-	-	-
cexpl	-	-	-	-	1 + i 1
cimagf	-	-	-	-	-
cimag	-	-	-	-	-
cimagl	-	-	-	-	-
clogf	1 + i 3	0 + i 3	-	1 + i 3	1 + i 0
clog	-	0 + i 1	-	-	-
clogl	-	-	-	-	1 + i 0
clog10f	1 + i 5	1 + i 5	-	1 + i 5	1 + i 1
clog10	0 + i 1	1 + i 1	-	0 + i 1	1 + i 1
clog10l	-	-	-	-	1 + i 1
conjf	-	-	-	-	-
conj	-	-	-	-	-
conjl	-	-	-	-	-
copysignf	-	-	-	-	-
copysign	-	-	-	-	-
copysignl	-	-	-	-	-
cosf	1	1	-	1	1
cos	2	2	-	2	2
cosl	-	-	-	-	1
coshf	-	-	-	-	-
cosh	-	-	-	-	-
coshl	-	-	-	-	-
cpowf	4 + i 2	4 + i 2	-	4 + i 2	4 + i 3
cpow	2 + i 2	1 + i 1.1031	-	2 + i 2	1 + i 2
cpowl	-	-	-	-	763 + i 2
cprojf	-	-	-	-	-
cproj	-	-	-	-	-
cprojl	-	-	-	-	-
crealf	-	-	-	-	-
creal	-	-	-	-	-
creall	-	-	-	-	-
csinf	-	0 + i 1	-	-	1 + i 1

csin	-	-	-	-	-
csinl	-	-	-	-	1 + i 0
csinhf	1 + i 1	1 + i 1	-	1 + i 1	1 + i 1
csinh	0 + i 1	0 + i 1	-	0 + i 1	1 + i 1
csinhl	-	-	-	-	1 + i 2
csqrtf	1 + i 0	1 + i 1	-	1 + i 0	-
csqrt	-	1 + i 0	-	-	-
csqrtl	-	-	-	-	-
ctanf	-	1 + i 1	-	-	0 + i 1
ctan	1 + i 1	1 + i 1	-	1 + i 1	1 + i 1
ctanl	-	-	-	-	439 + i 3
ctanhf	2 + i 1	2 + i 1	-	2 + i 1	1 + i 1
ctanh	1 + i 0	2 + i 2	-	1 + i 0	0 + i 1
ctanhl	-	-	-	-	5 + i 25
erff	-	-	-	-	-
erf	1	-	-	1	1
erfl	-	-	-	-	-
erfcf	-	12	-	-	1
erfc	1	24	-	1	1
erfcl	-	-	-	-	1
expf	-	-	-	-	-
exp	-	-	-	-	-
expl	-	-	-	-	-
exp10f	2	2	-	2	-
exp10	6	6	-	6	-
exp10l	-	-	-	-	8
exp2f	-	-	-	-	-
exp2	-	-	-	-	-
exp2l	-	-	-	-	-
expm1f	1	1	-	1	-
expm1	1	-	-	1	-
expm1l	-	-	-	-	-
fabsf	-	-	-	-	-
fabs	-	-	-	-	-
fabsl	-	-	-	-	-
fdimf	-	-	-	-	-
fdim	-	-	-	-	-
fdiml	-	-	-	-	-
floorf	-	-	-	-	-
floor	-	-	-	-	-
floorl	-	-	-	-	-
fmaf	-	-	-	-	-
fma	-	-	-	-	-
fmal	-	-	-	-	-
fmaxf	-	-	-	-	-
fmax	-	-	-	-	-

fmaxl	-	-	-	-	-
fminf	-	-	-	-	-
fmin	-	-	-	-	-
fminl	-	-	-	-	-
fmodf	-	1	-	-	-
fmod	-	2	-	-	-
fmodl	-	-	-	-	-
frexpf	-	-	-	-	-
frexp	-	-	-	-	-
frexpl	-	-	-	-	-
gammaf	-	-	-	-	-
gamma	-	-	-	-	1
gammal	-	-	-	-	1
hypotf	1	1	-	1	1
hypot	-	1	-	-	-
hypotl	-	-	-	-	-
ilogbf	-	-	-	-	-
ilogb	-	-	-	-	-
ilogbl	-	-	-	-	-
j0f	2	2	-	2	1
j0	2	2	-	2	1
j0l	-	-	-	-	1
j1f	2	2	-	2	1
j1	1	1	-	1	1
j1l	-	-	-	-	1
jnf	4	4	-	4	2
jn	4	6	-	4	2
jnl	-	-	-	-	2
lgammaf	2	2	-	2	2
lgamma	1	1	-	1	1
lgammal	-	-	-	-	1
lrintf	-	-	-	-	-
lrint	-	-	-	-	-
lrintl	-	-	-	-	-
llrintf	-	-	-	-	-
llrint	-	-	-	-	-
llrintl	-	-	-	-	-
logf	-	1	-	-	1
log	-	1	-	-	-
logl	-	-	-	-	-
log10f	2	1	-	2	1
log10	1	1	-	1	-
log10l	-	-	-	-	1
log1pf	1	1	-	1	-
log1p	-	1	-	-	-
log1pl	-	-	-	-	-

log2f	-	1	-	-	-
log2	-	1	-	-	-
log2l	-	-	-	-	-
logbf	-	-	-	-	-
logb	-	-	-	-	-
logbl	-	-	-	-	-
lroundf	-	-	-	-	-
lround	-	-	-	-	-
lroundl	-	-	-	-	-
llroundf	-	-	-	-	-
llround	-	-	-	-	-
llroundl	-	-	-	-	-
modff	-	-	-	-	-
modf	-	-	-	-	-
modfl	-	-	-	-	-
nearbyintf	-	-	-	-	-
nearbyint	-	-	-	-	-
nearbyintl	-	-	-	-	-
nextafterf	-	-	-	-	-
nextafter	-	-	-	-	-
nextafterl	-	-	-	-	-
nexttowardf	-	-	-	-	-
nexttoward	-	-	-	-	-
nexttowardl	-	-	-	-	-
powf	-	-	-	-	-
pow	-	-	-	-	-
powl	-	-	-	-	-
remainderf	-	-	-	-	-
remainder	-	-	-	-	-
remainderl	-	-	-	-	-
remquof	-	-	-	-	-
remquo	-	-	-	-	-
remquol	-	-	-	-	-
rintf	-	-	-	-	-
rint	-	-	-	-	-
rintl	-	-	-	-	-
roundf	-	-	-	-	-
round	-	-	-	-	-
roundl	-	-	-	-	-
scalbf	-	-	-	-	-
scalb	-	-	-	-	-
scalbl	-	-	-	-	-
scalbnf	-	-	-	-	-
scalbn	-	-	-	-	-
scalbnl	-	-	-	-	-
scalblnf	-	-	-	-	-

scalbln	-	-	-	-	-
scalblnl	-	-	-	-	-
sinf	-	-	-	-	-
sin	-	-	-	-	-
sinl	-	-	-	-	-
sincosf	1	1	-	1	1
sincos	1	1	-	1	1
sincosl	-	-	-	-	1
sinhf	-	1	-	-	-
sinh	-	1	-	-	-
sinhl	-	-	-	-	-
sqrtf	-	-	-	-	-
sqrt	-	-	-	-	-
sqrtl	-	-	-	-	-
tanf	-	-	-	-	-
tan	1	0.5	-	1	1
tanl	-	-	-	-	-
tanhf	-	1	-	-	-
tanh	-	1	-	-	-
tanhl	-	-	-	-	-
tgammaf	1	1	-	1	1
tgamma	1	1	-	1	2
tgammal	-	-	-	-	1
truncf	-	-	-	-	-
trunc	-	-	-	-	-
truncl	-	-	-	-	-
y0f	1	1	-	1	1
y0	2	2	-	2	2
y0l	-	-	-	-	1
y1f	2	2	-	2	2
y1	3	3	-	3	2
y1l	-	-	-	-	1
ynf	2	2	-	2	3
yn	3	3	-	3	2
ynl	-	-	-	-	4
Function	*IA64*	*M68k*	*MIPS*	*PowerPC*	*powerpc/nofpu*
acosf	-	-	-	-	-
acos	-	-	-	-	-
acosl	-	-	-	-	-
acoshf	-	-	-	-	-
acosh	-	-	-	-	-
acoshl	-	1	-	-	-
asinf	-	-	-	-	-
asin	-	-	-	-	-
asinl	-	-	-	-	-
asinhf	-	-	-	-	-

asinh	-	-	-	-	-
asinhl	-	1	-	-	-
atanf	-	-	-	-	-
atan	-	-	-	-	-
atanl	-	-	-	-	-
atanhf	-	-	1	1	1
atanh	-	-	-	-	-
atanhl	-	1	-	-	-
atan2f	-	-	3	3	3
atan2	-	-	-	-	-
atan2l	-	1	-	-	-
cabsf	-	-	-	-	-
cabs	-	-	-	-	-
cabsl	-	-	-	-	-
cacosf	0 + i 1	2 + i 1	-	-	-
cacos	-	-	-	-	-
cacosl	0 + i 2	1 + i 2	-	-	-
cacoshf	7 + i 0	7 + i 1	7 + i 3	7 + i 3	7 + i 3
cacosh	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
cacoshl	7 + i 1	6 + i 2	-	-	-
cargf	-	-	-	-	-
carg	-	-	-	-	-
cargl	-	-	-	-	-
casinf	1 + i 1	5 + i 1	1 + i 0	1 + i 0	1 + i 0
casin	1 + i 0	1 + i 0	1 + i 0	1 + i 0	1 + i 0
casinl	2 + i 2	3 + i 2	-	-	-
casinhf	1 + i 6	19 + i 1	1 + i 6	1 + i 6	1 + i 6
casinh	5 + i 3	6 + i 13	5 + i 3	5 + i 3	5 + i 3
casinhl	5 + i 5	5 + i 6	-	-	-
catanf	0 + i 1	0 + i 1	4 + i 1	4 + i 1	4 + i 1
catan	0 + i 1	0 + i 1	0 + i 1	0 + i 1	0 + i 1
catanl	-	1 + i 0	-	-	-
catanhf	-	-	0 + i 6	0 + i 6	0 + i 6
catanh	4 + i 0	-	4 + i 0	4 + i 0	4 + i 0
catanhl	1 + i 0	1 + i 0	-	-	-
cbrtf	-	-	-	-	-
cbrt	-	-	1	1	1
cbrtl	-	1	-	-	-
ccosf	0 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
ccos	1 + i 0	-	1 + i 0	1 + i 0	1 + i 0
ccosl	1 + i 1	1 + i 1	-	-	-
ccoshf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
ccosh	1 + i 1	-	1 + i 0	1 + i 0	1 + i 0
ccoshl	0 + i 1	0 + i 1	-	-	-
ceilf	-	-	-	-	-
ceil	-	-	-	-	-

ceill	-	-	-	-	-
cexpf	1 + i 1	2 + i 1	1 + i 1	1 + i 1	1 + i 1
cexp	-	-	-	-	-
cexpl	0 + i 1	0 + i 1	-	-	-
cimagf	-	-	-	-	-
cimag	-	-	-	-	-
cimagl	-	-	-	-	-
clogf	1 + i 0	1 + i 0	1 + i 3	1 + i 3	1 + i 3
clog	-	-	-	-	-
clogl	1 + i 0	1 + i 1	-	-	-
clog10f	1 + i 1	1 + i 1	1 + i 5	1 + i 5	1 + i 5
clog10	1 + i 1	1 + i 1	0 + i 1	0 + i 1	0 + i 1
clog10l	1 + i 1	1 + i 2	-	-	-
conjf	-	-	-	-	-
conj	-	-	-	-	-
conjl	-	-	-	-	-
copysignf	-	-	-	-	-
copysign	-	-	-	-	-
copysignl	-	-	-	-	-
cosf	1	1	1	1	1
cos	2	2	2	2	2
cosl	1	1	-	-	-
coshf	-	-	-	-	-
cosh	-	-	-	-	-
coshl	-	-	-	-	-
cpowf	5 + i 3	2 + i 6	4 + i 2	4 + i 2	4 + i 2
cpow	2 + i 2	1 + i 2	2 + i 2	2 + i 2	2 + i 2
cpowl	3 + i 4	15 + i 2	-	-	-
cprojf	-	-	-	-	-
cproj	-	-	-	-	-
cprojl	-	-	-	-	-
crealf	-	-	-	-	-
creal	-	-	-	-	-
creall	-	-	-	-	-
csinf	1 + i 1	1 + i 1	-	-	-
csin	-	-	-	-	-
csinl	1 + i 0	1 + i 0	-	-	-
csinhf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
csinh	1 + i 1	-	0 + i 1	0 + i 1	0 + i 1
csinhl	1 + i 2	1 + i 0	-	-	-
csqrtf	1 + i 0	-	1 + i 0	1 + i 0	1 + i 0
csqrt	-	-	-	-	-
csqrtl	-	-	-	-	-
ctanf	0 + i 1	-	-	-	-
ctan	1 + i 1	1 + i 0	1 + i 1	1 + i 1	1 + i 1
ctanl	2 + i 1	1 + i 2	-	-	-

ctanhf	0 + i 1	0 + i 1	2 + i 1	2 + i 1	2 + i 1
ctanh	1 + i 1	0 + i 1	1 + i 0	1 + i 0	1 + i 0
ctanhl	1 + i 24	0 + i 1	-	-	-
erff	-	-	-	-	-
erf	1	-	1	1	1
erfl	-	-	-	-	-
erfcf	1	1	-	1	-
erfc	1	-	1	1	1
erfcl	1	1	-	-	-
expf	-	-	-	-	-
exp	-	-	-	-	-
expl	-	-	-	-	-
exp10f	2	-	2	2	2
exp10	6	-	6	6	6
exp10l	3	-	-	-	-
exp2f	-	-	-	-	-
exp2	-	-	-	-	-
exp2l	-	-	-	-	-
expm1f	-	-	1	1	1
expm1	-	-	1	1	1
expm1l	1	1	-	-	-
fabsf	-	-	-	-	-
fabs	-	-	-	-	-
fabsl	-	-	-	-	-
fdimf	-	-	-	-	-
fdim	-	-	-	-	-
fdiml	-	-	-	-	-
floorf	-	-	-	-	-
floor	-	-	-	-	-
floorl	-	-	-	-	-
fmaf	-	-	-	-	-
fma	-	-	-	-	-
fmal	-	-	-	-	-
fmaxf	-	-	-	-	-
fmax	-	-	-	-	-
fmaxl	-	-	-	-	-
fminf	-	-	-	-	-
fmin	-	-	-	-	-
fminl	-	-	-	-	-
fmodf	-	-	-	-	-
fmod	-	-	-	-	-
fmodl	-	-	-	-	-
frexpf	-	-	-	-	-
frexp	-	-	-	-	-
frexpl	-	-	-	-	-
gammaf	-	-	-	-	-

gamma	-	-	-	-	-
gammal	1	1	-	-	-
hypotf	1	1	1	1	1
hypot	-	-	-	-	-
hypotl	-	-	-	-	-
ilogbf	-	-	-	-	-
ilogb	-	-	-	-	-
ilogbl	-	-	-	-	-
j0f	1	1	2	1	2
j0	2	1	2	2	2
j0l	2	1	-	-	-
j1f	2	2	2	2	2
j1	1	-	1	1	1
j1l	1	1	-	-	-
jnf	3	5	4	3	4
jn	3	1	4	3	4
jnl	2	2	-	-	-
lgammaf	2	2	2	2	2
lgamma	1	1	1	1	1
lgammal	1	1	-	-	-
lrintf	-	-	-	-	-
lrint	-	-	-	-	-
lrintl	-	-	-	-	-
llrintf	-	-	-	-	-
llrint	-	-	-	-	-
llrintl	-	-	-	-	-
logf	1	1	-	-	-
log	-	-	-	-	-
logl	-	1	-	-	-
log10f	1	1	2	2	2
log10	-	-	1	1	1
log10l	1	2	-	-	-
log1pf	-	-	1	1	1
log1p	-	-	-	-	-
log1pl	-	1	-	-	-
log2f	-	-	-	-	-
log2	-	-	-	-	-
log2l	-	1	-	-	-
logbf	-	-	-	-	-
logb	-	-	-	-	-
logbl	-	-	-	-	-
lroundf	-	-	-	-	-
lround	-	-	-	-	-
lroundl	-	-	-	-	-
llroundf	-	-	-	-	-
llround	-	-	-	-	-

llroundl	-	-	-	-	-
modff	-	-	-	-	-
modf	-	-	-	-	-
modfl	-	-	-	-	-
nearbyintf	-	-	-	-	-
nearbyint	-	-	-	-	-
nearbyintl	-	-	-	-	-
nextafterf	-	-	-	-	-
nextafter	-	-	-	-	-
nextafterl	-	-	-	-	-
nexttowardf	-	-	-	-	-
nexttoward	-	-	-	-	-
nexttowardl	-	-	-	-	-
powf	-	-	-	-	-
pow	-	-	-	-	-
powl	-	1	-	-	-
remainderf	-	-	-	-	-
remainder	-	-	-	-	-
remainderl	-	-	-	-	-
remquof	-	-	-	-	-
remquo	-	-	-	-	-
remquol	-	-	-	-	-
rintf	-	-	-	-	-
rint	-	-	-	-	-
rintl	-	-	-	-	-
roundf	-	-	-	-	-
round	-	-	-	-	-
roundl	-	-	-	-	-
scalbf	-	-	-	-	-
scalb	-	-	-	-	-
scalbl	-	-	-	-	-
scalbnf	-	-	-	-	-
scalbn	-	-	-	-	-
scalbnl	-	-	-	-	-
scalblnf	-	-	-	-	-
scalbln	-	-	-	-	-
scalblnl	-	-	-	-	-
sinf	-	-	-	-	-
sin	-	-	-	-	-
sinl	-	-	-	-	-
sincosf	1	1	1	1	1
sincos	1	1	1	1	1
sincosl	1	1	-	-	-
sinhf	-	-	-	-	-
sinh	-	-	-	-	-
sinhl	-	1	-	-	-

sqrtf	-	-	-	-	-
sqrt	-	-	-	-	-
sqrtl	-	-	-	-	-
tanf	-	-	-	-	-
tan	1	1	1	1	1
tanl	-	1	-	-	-
tanhf	-	-	-	-	-
tanh	-	-	-	-	-
tanhl	-	-	-	-	-
tgammaf	1	1	1	1	1
tgamma	1	1	1	1	1
tgammal	1	1	-	-	-
truncf	-	-	-	-	-
trunc	-	-	-	-	-
truncl	-	-	-	-	-
y0f	1	1	1	1	1
y0	2	1	2	2	2
y0l	1	2	-	-	-
y1f	2	2	2	2	2
y1	3	1	3	3	3
y1l	1	1	-	-	-
ynf	2	2	2	2	2
yn	3	1	3	3	3
ynl	2	4	-	-	-

Function	*S/390*	*SH4*	*Sparc 32-bit*	*Sparc 64-bit*	*x86_64/fpu*
acosf	-	-	-	-	-
acos	-	-	-	-	-
acosl	-	-	-	-	-
acoshf	-	-	-	-	-
acosh	-	-	-	-	-
acoshl	-	-	-	-	-
asinf	-	2	-	-	-
asin	-	1	-	-	-
asinl	-	-	-	-	1
asinhf	-	-	-	-	-
asinh	-	-	-	-	-
asinhl	-	-	-	-	-
atanf	-	-	-	-	-
atan	-	-	-	-	-
atanl	-	-	-	-	-
atanhf	1	-	1	1	1
atanh	-	1	-	-	-
atanhl	-	-	-	-	1
atan2f	3	4	3	3	3
atan2	-	-	-	-	-

atan2l	-	-	-	1	-
cabsf	-	1	-	-	-
cabs	-	1	-	-	-
cabsl	-	-	-	-	-
cacosf	-	1 + i 1	-	-	0 + i 1
cacos	-	1 + i 0	-	-	-
cacosl	-	-	-	0 + i 1	0 + i 2
cacoshf	7 + i 3	7 + i 3	7 + i 3	7 + i 3	7 + i 3
cacosh	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
cacoshl	-	-	-	5 + i 1	6 + i 1
cargf	-	-	-	-	-
carg	-	-	-	-	-
cargl	-	-	-	-	-
casinf	1 + i 0	2 + i 1	1 + i 0	1 + i 0	1 + i 1
casin	1 + i 0	3 + i 0	1 + i 0	1 + i 0	1 + i 0
casinl	-	-	-	0 + i 1	2 + i 2
casinhf	1 + i 6	1 + i 6	1 + i 6	1 + i 6	1 + i 6
casinh	5 + i 3	5 + i 3	5 + i 3	5 + i 3	5 + i 3
casinhl	-	-	-	4 + i 2	5 + i 5
catanf	4 + i 1	4 + i 1	4 + i 1	4 + i 1	4 + i 1
catan	0 + i 1	0 + i 1	0 + i 1	0 + i 1	0 + i 1
catanl	-	-	-	0 + i 1	-
catanhf	0 + i 6	1 + i 6	0 + i 6	0 + i 6	0 + i 6
catanh	4 + i 0	4 + i 1	4 + i 0	4 + i 0	4 + i 0
catanhl	-	-	-	1 + i 1	1 + i 0
cbrtf	-	-	-	-	-
cbrt	1	1	1	1	1
cbrtl	-	-	-	1	1
ccosf	1 + i 1	0 + i 1	1 + i 1	1 + i 1	1 + i 1
ccos	1 + i 0	1 + i 1	1 + i 0	1 + i 0	1 + i 0
ccosl	-	-	-	-	1 + i 1
ccoshf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
ccosh	1 + i 0	1 + i 1	1 + i 0	1 + i 0	1 + i 1
ccoshl	-	-	-	-	0 + i 1
ceilf	-	-	-	-	-
ceil	-	-	-	-	-
ceill	-	-	-	-	-
cexpf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
cexp	-	1 + i 0	-	-	-
cexpl	-	-	-	1 + i 1	0 + i 1
cimagf	-	-	-	-	-
cimag	-	-	-	-	-
cimagl	-	-	-	-	-
clogf	1 + i 3	0 + i 3	1 + i 3	1 + i 3	1 + i 3
clog	-	0 + i 1	-	-	-
clogl	-	-	-	1 + i 0	1 + i 0

clog10f	1 + i 5	1 + i 5	1 + i 5	1 + i 5	1 + i 5
clog10	0 + i 1	1 + i 1	0 + i 1	0 + i 1	1 + i 1
clog10l	-	-	-	0 + i 1	1 + i 1
conjf	-	-	-	-	-
conj	-	-	-	-	-
conjl	-	-	-	-	-
copysignf	-	-	-	-	-
copysign	-	-	-	-	-
copysignl	-	-	-	-	-
cosf	1	1	1	1	1
cos	2	2	2	2	2
cosl	-	-	-	1	1
coshf	-	-	-	-	-
cosh	-	-	-	-	-
coshl	-	-	-	-	-
cpowf	4 + i 2	4 + i 2	4 + i 2	4 + i 2	5 + i 2
cpow	2 + i 2	1 + i 1.1031	2 + i 2	2 + i 2	2 + i 2
cpowl	-	-	-	1 + i 1	5 + i 2
cprojf	-	-	-	-	-
cproj	-	-	-	-	-
cprojl	-	-	-	-	-
crealf	-	-	-	-	-
creal	-	-	-	-	-
creall	-	-	-	-	-
csinf	-	0 + i 1	-	-	0 + i 1
csin	-	-	-	-	-
csinl	-	-	-	1 + i 0	1 + i 0
csinhf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
csinh	0 + i 1	0 + i 1	0 + i 1	0 + i 1	1 + i 1
csinhl	-	-	-	-	1 + i 2
csqrtf	1 + i 0	1 + i 1	1 + i 0	1 + i 0	1 + i 0
csqrt	-	1 + i 0	-	-	-
csqrtl	-	-	-	1 + i 1	-
ctanf	-	1 + i 1	-	-	0 + i 1
ctan	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
ctanl	-	-	-	0 + i 2	439 + i 3
ctanhf	2 + i 1	2 + i 1	2 + i 1	2 + i 1	2 + i 1
ctanh	1 + i 0	2 + i 2	1 + i 0	1 + i 0	1 + i 1
ctanhl	-	-	-	-	5 + i 25
erff	-	-	-	-	-
erf	1	-	1	1	1
erfl	-	-	-	-	-
erfcf	-	12	-	-	-
erfc	1	24	1	1	1
erfcl	-	-	-	1	1
expf	-	-	-	-	-

exp	-	-	-	-	-
expl	-	-	-	-	-
exp10f	2	2	2	2	2
exp10	6	6	6	6	6
exp10l	-	-	-	1	3
exp2f	-	-	-	-	-
exp2	-	-	-	-	-
exp2l	-	-	-	-	-
expm1f	1	1	1	1	1
expm1	1	-	1	1	1
expm1l	-	-	-	1	-
fabsf	-	-	-	-	-
fabs	-	-	-	-	-
fabsl	-	-	-	-	-
fdimf	-	-	-	-	-
fdim	-	-	-	-	-
fdiml	-	-	-	-	-
floorf	-	-	-	-	-
floor	-	-	-	-	-
floorl	-	-	-	-	-
fmaf	-	-	-	-	-
fma	-	-	-	-	-
fmal	-	-	-	-	-
fmaxf	-	-	-	-	-
fmax	-	-	-	-	-
fmaxl	-	-	-	-	-
fminf	-	-	-	-	-
fmin	-	-	-	-	-
fminl	-	-	-	-	-
fmodf	-	1	-	-	-
fmod	-	2	-	-	-
fmodl	-	-	-	-	-
frexpf	-	-	-	-	-
frexp	-	-	-	-	-
frexpl	-	-	-	-	-
gammaf	-	-	-	-	-
gamma	-	-	-	-	-
gammal	-	-	-	1	1
hypotf	1	1	1	1	1
hypot	-	1	-	-	-
hypotl	-	-	-	-	-
ilogbf	-	-	-	-	-
ilogb	-	-	-	-	-
ilogbl	-	-	-	-	-
j0f	2	2	2	2	2
j0	2	2	2	2	2

j0l	-	-	-	2	1
j1f	2	2	2	2	2
j1	1	1	1	1	1
j1l	-	-	-	4	1
jnf	4	4	4	4	4
jn	4	6	4	4	4
jnl	-	-	-	4	2
lgammaf	2	2	2	2	2
lgamma	1	1	1	1	1
lgammal	-	-	-	1	1
lrintf	-	-	-	-	-
lrint	-	-	-	-	-
lrintl	-	-	-	-	-
llrintf	-	-	-	-	-
llrint	-	-	-	-	-
llrintl	-	-	-	-	-
logf	-	1	-	-	-
log	-	1	-	-	-
logl	-	-	-	-	-
log10f	2	1	2	2	2
log10	1	1	1	1	1
log10l	-	-	-	1	1
log1pf	1	1	1	1	1
log1p	-	1	-	-	-
log1pl	-	-	-	1	-
log2f	-	1	-	-	-
log2	-	1	-	-	-
log2l	-	-	-	1	-
logbf	-	-	-	-	-
logb	-	-	-	-	-
logbl	-	-	-	-	-
lroundf	-	-	-	-	-
lround	-	-	-	-	-
lroundl	-	-	-	-	-
llroundf	-	-	-	-	-
llround	-	-	-	-	-
llroundl	-	-	-	-	-
modff	-	-	-	-	-
modf	-	-	-	-	-
modfl	-	-	-	-	-
nearbyintf	-	-	-	-	-
nearbyint	-	-	-	-	-
nearbyintl	-	-	-	-	-
nextafterf	-	-	-	-	-
nextafter	-	-	-	-	-
nextafterl	-	-	-	-	-

nexttowardf	-	-	-	-	-
nexttoward	-	-	-	-	-
nexttowardl	-	-	-	-	-
powf	-	-	-	-	-
pow	-	-	-	-	-
powl	-	-	-	-	-
remainderf	-	-	-	-	-
remainder	-	-	-	-	-
remainderl	-	-	-	-	-
remquof	-	-	-	-	-
remquo	-	-	-	-	-
remquol	-	-	-	-	-
rintf	-	-	-	-	-
rint	-	-	-	-	-
rintl	-	-	-	-	-
roundf	-	-	-	-	-
round	-	-	-	-	-
roundl	-	-	-	-	-
scalbf	-	-	-	-	-
scalb	-	-	-	-	-
scalbl	-	-	-	-	-
scalbnf	-	-	-	-	-
scalbn	-	-	-	-	-
scalbnl	-	-	-	-	-
scalblnf	-	-	-	-	-
scalbln	-	-	-	-	-
scalblnl	-	-	-	-	-
sinf	-	-	-	-	-
sin	-	-	-	-	-
sinl	-	-	-	-	-
sincosf	1	1	1	1	1
sincos	1	1	1	1	1
sincosl	-	-	-	1	1
sinhf	-	1	-	-	-
sinh	-	1	-	-	-
sinhl	-	-	-	-	-
sqrtf	-	-	-	-	-
sqrt	-	-	-	-	-
sqrtl	-	-	-	1	-
tanf	-	-	-	-	-
tan	1	0.5	1	1	1
tanl	-	-	-	-	-
tanhf	-	1	-	-	-
tanh	-	1	-	-	-
tanhl	-	-	-	1	-
tgammaf	1	1	1	1	1

tgamma	1	1	1	1	1
tgammal	-	-	-	1	1
truncf	-	-	-	-	-
trunc	-	-	-	-	-
truncl	-	-	-	-	-
y0f	1	1	1	1	1
y0	2	2	2	2	2
y0l	-	-	-	3	1
y1f	2	2	2	2	2
y1	3	3	3	3	3
y1l	-	-	-	1	1
ynf	2	2	2	2	2
yn	3	3	3	3	3
ynl	-	-	-	5	4

8.8 Pseudorandom Numbers

This section describes the GNU facilities for generating a series of pseudorandom numbers. The numbers generated are not truly random; typically, they form a sequence that repeats periodically, with a period so large that you can ignore it for ordinary purposes. The random-number generator works by remembering a *seed* value, which it uses to compute the next random number and also to compute a new seed.

Although the generated numbers look unpredictable within one run of a program, the sequence of numbers is *exactly the same* from one run to the next. This is because the initial seed is always the same. This is convenient when you are debugging a program, but it is unhelpful if you want the program to behave unpredictably. If you want a different pseudorandom series each time your program runs, you must specify a different seed each time. For ordinary purposes, basing the seed on the current time works well.

You can obtain repeatable sequences of numbers on a particular machine type by specifying the same initial seed value for the random-number generator. There is no standard meaning for a particular seed value; the same seed, used in different C libraries or on different CPU types, will give you different random numbers.

The GNU library supports the standard ISO C random-number functions plus two other sets derived from BSD and SVID. The BSD and ISO C functions provide identical, somewhat limited functionality. If only a small number of random bits are required, we recommend you use the ISO C interface, `rand` and `srand`. The SVID functions provide a more flexible interface, which allows better random-number generator algorithms, provides more random bits (up to 48) per call, and can provide random floating-point numbers. These functions are required by the XPG standard and therefore will be present in all modern Unix systems.

8.8.1 ISO C Random-Number Functions

This section describes the random-number functions that are part of the ISO C standard.

To use these facilities, you should include the header file ‘stdlib.h’ in your program.

`int` **RAND_MAX** Macro

The value of this macro is an integer constant representing the largest value the `rand` function can return. In the GNU library, it is `2147483647`, which is the largest signed integer representable in 32 bits. In other libraries, it may be as low as `32767`.

`int` **rand** (`void`) Function

The `rand` function returns the next pseudorandom number in the series. The value ranges from `0` to `RAND_MAX`.

`void` **srand** (`unsigned int` *seed*) Function

This function establishes *seed* as the seed for a new series of pseudorandom numbers. If you call `rand` before a seed has been established with `srand`, it uses the value `1` as a default seed.

To produce a different pseudorandom series each time your program is run, do `srand (time (0))`.

POSIX.1 extended the C standard functions to support reproducible random numbers in multithreaded programs. However, the extension is badly designed and unsuitable for serious work.

`int` **rand_r** (`unsigned int *`*seed*) Function

This function returns a random number in the range 0 to `RAND_MAX` just as `rand` does. However, all its state is stored in the *seed* argument. This means the RNG's state can only have as many bits as the type `unsigned int` has. This is far too few to provide a good RNG.

If your program requires a reentrant RNG, we recommend you use the reentrant GNU extensions to the SVID random-number generator. The POSIX.1 interface should only be used when the GNU extensions are not available.

8.8.2 BSD Random-Number Functions

This section describes a set of random-number generation functions that are derived from BSD. There is no advantage to using these functions with the GNU C Library; we support them for BSD compatibility only.

The prototypes for these functions are in ‘stdlib.h’.

`long int` **random** (`void`) Function

This function returns the next pseudorandom number in the sequence. The value returned ranges from `0` to `RAND_MAX`.

Temporarily, this function was defined to return a `int32_t` value to indicate that the return value always contains 32 bits even if `long int` is wider. The standard demands it differently. Users must always be aware of the 32-bit limitation, though.

`void` **srandom** (`unsigned int` *seed*) Function

The `srandom` function sets the state of the random-number generator based on the integer *seed*. If you supply a *seed* value of `1`, this will cause `random` to reproduce the default set of random numbers.

To produce a different set of pseudorandom numbers each time your program runs, do `srandom (time (0))`.

`void *` **initstate** (`unsigned int` *seed*, `void *`*state*, `size_t` *size*) Function

The `initstate` function is used to initialize the random-number generator state. The argument *state* is an array of *size* bytes, used to hold the state information. It is initialized based on *seed*. The size must be between 8 and 256 bytes, and should be a power of two. The bigger the *state* array, the better.

The return value is the previous value of the state information array. You can use this value later as an argument to `setstate` to restore that state.

`void *` **setstate** (`void *`*state*) Function

The `setstate` function restores the random-number state information *state*. The argument must have been the result of a previous call to *initstate* or *setstate*.

The return value is the previous value of the state information array. You can use this value later as an argument to `setstate` to restore that state.

If the function fails, the return value is `NULL`.

The four functions described so far in this section all work on a state that is shared by all threads. The state is not directly accessible to the user and can only be modified by these functions. This makes it hard to deal with situations where each thread should have its own pseudorandom-number generator.

The GNU C Library contains four additional functions that contain the state as an explicit parameter and therefore make it possible to handle thread-local PRNGs. Besides this, there are no differences. In fact, the four functions already discussed are implemented internally using the following interfaces.

The '`stdlib.h`' header contains a definition of the following type:

struct random_data Data Type

Objects of type `struct random_data` contain the information necessary to represent the state of the PRNG. Although a complete definition of the type is present, the type should be treated as opaque.

The functions modifying the state follow exactly the already described functions.

`int` **random_r** (`struct random_data *restrict` *buf*, `int32_t *restrict` *result*) Function

The `random_r` function behaves exactly like the `random` function, except that it uses and modifies the state in the object pointed to by the first parameter instead of the global state.

`int` **srandom_r** (`unsigned int` *seed*, `struct random_data *` *buf*) Function

The `srandom_r` function behaves exactly like the `srandom` function, except that it uses and modifies the state in the object pointed to by the second parameter instead of the global state.

`int` **initstate_r** (`unsigned int` *seed*, `char *restrict` *statebuf*, `size_t` *statelen*, `struct random_data *restrict` *buf*) Function

The `initstate_r` function behaves exactly like the `initstate` function, except that it uses and modifies the state in the object pointed to by the fourth parameter instead of the global state.

`int` **setstate_r** (`char *restrict` *statebuf*, `struct random_data *restrict` *buf*) Function

The `setstate_r` function behaves exactly like the `setstate` function, except that it uses and modifies the state in the object pointed to by the first parameter instead of the global state.

8.8.3 SVID Random-Number Functions

The C library on SVID systems contains yet another kind of random-number generator functions. They use a state of 48 bits of data. The user can choose among a collection of functions that return the random bits in different forms.

Generally, there are two kinds of function. The first uses a state of the random-number generator that is shared among several functions and by all threads of the process. The second requires the user to handle the state.

All functions have in common that they use the same congruential formula with the same constants. The formula is

```
Y = (a * X + c) mod m
```

where X is the state of the generator at the beginning and Y the state at the end. `a` and `c` are constants determining the way the generator works. By default they are

```
a = 0x5DEECE66D = 25214903917
c = 0xb = 11
```

but they can also be changed by the user. `m` is of course 2^48 since the state consists of a 48-bit array.

The prototypes for these functions are in '`stdlib.h`'.

`double` **drand48** (`void`) Function
This function returns a `double` value in the range of `0.0` to `1.0` (exclusive). The random bits are determined by the global state of the random-number generator in the C library.

Since the `double` type according to IEEE 754 has a 52-bit mantissa, 4 bits are not initialized by the random-number generator. These are chosen to be the least significant bits and are initialized to `0`.

`double` **erand48** (`unsigned short int` *xsubi*`[3]`) Function
This function returns a `double` value in the range of `0.0` to `1.0` (exclusive), similarly to `drand48`. The argument is an array describing the state of the random-number generator.

This function can be called subsequently since it updates the array to guarantee random numbers. The array should have been initialized before initial use to obtain reproducible results.

`long int` **lrand48** (`void`) Function
The `lrand48` function returns an integer value in the range of `0` to `2^31` (exclusive). Even if the size of the `long int` type can take more than 32 bits, no higher numbers are returned. The random bits are determined by the global state of the random-number generator in the C library.

`long int` **nrand48** (`unsigned short int` *xsubi*`[3]`) Function
This function is similar to the `lrand48` function in that it returns a number in the range of `0` to `2^31` (exclusive), but the state of the random-number generator used to produce the random bits is determined by the array provided as the parameter to the function.

The numbers in the array are updated afterwards so that subsequent calls to this function yield different results (as is expected of a random-number generator). The array should have been initialized before the first call to obtain reproducible results.

`long int` **mrand48** (`void`) Function
The `mrand48` function is similar to `lrand48`. The only difference is that the numbers returned are in the range `-2^31` to `2^31` (exclusive).

`long int` **jrand48** (`unsigned short int` *xsubi*`[3]`) Function
The `jrand48` function is similar to `nrand48`. The only difference is that the numbers returned are in the range `-2^31` to `2^31` (exclusive). For the `xsubi` parameter the same requirements are necessary.

The internal state of the random-number generator can be initialized in several ways. The methods differ in the completeness of the information provided.

`void` **srand48** (`long int` *seedval*) Function

The `srand48` function sets the most significant 32 bits of the internal state of the random-number generator to the least significant 32 bits of the *seedval* parameter. The lower 16 bits are initialized to the value `0x330E`. Even if the `long int` type contains more than 32 bits, only the lower 32 bits are used.

Owing to this limitation, initialization of the state of this function is not very useful. But it makes it easy to use a construct like `srand48 (time (0))`.

A side effect of this function is that the values `a` and `c` from the internal state, which are used in the congruential formula, are reset to the default values given above. This is of importance once the user has called the `lcong48` function (see below).

`unsigned short int *` **seed48** (`unsigned short int` *seed16v*`[3]`) Function

The `seed48` function initializes all 48 bits of the state of the internal random-number generator from the contents of the parameter *seed16v*. Here the lower 16 bits of the first element of *see16v* initialize the least significant 16 bits of the internal state, the lower 16 bits of *seed16v*`[1]` initialize the mid-order 16 bits of the state and the 16 lower bits of *seed16v*`[2]` initialize the most significant 16 bits of the state.

Unlike `srand48` this function lets the user initialize all 48 bits of the state.

The value returned by `seed48` is a pointer to an array containing the values of the internal state before the change. This might be useful to restart the random-number generator at a certain state. Otherwise the value can simply be ignored.

As for `srand48`, the values `a` and `c` from the congruential formula are reset to the default values.

There is one more function to initialize the random-number generator that enables you to specify even more information by allowing you to change the parameters in the congruential formula.

`void` **lcong48** (`unsigned short int` *param*`[7]`) Function

The `lcong48` function allows the user to change the complete state of the random-number generator. Unlike `srand48` and `seed48`, this function also changes the constants in the congruential formula.

From the seven elements in the array *param* the least significant 16 bits of the entries *param*`[0]` to *param*`[2]` determine the initial state, the least significant 16 bits of *param*`[3]` to *param*`[5]` determine the 48-bit constant `a` and *param*`[6]` determines the 16-bit value `c`.

All the above functions have in common that they use the global parameters for the congruential formula. In multithreaded programs, it might sometimes be useful to have different parameters in different threads. For this reason, all the above functions have a counterpart that works on a description of the random-number generator in the user-supplied buffer instead of the global state.

It is no problem if several threads use the global state if all threads use the functions that take a pointer to an array containing the state. The random numbers are computed following the same loop, but if the state in the array is different, all threads will obtain an individual random-number generator.

The user-supplied buffer must be of type `struct drand48_data`. This type should be regarded as opaque and not manipulated directly.

`int` **drand48_r** (`struct drand48_data *`*buffer*`, double *`*result*) Function

This function is equivalent to the `drand48` function with the difference that it does not modify the global random-number generator parameters but instead the parameters in the buffer supplied through the pointer *buffer*. The random number is returned in the variable pointed to by *result*.

The return value of the function indicates whether the call succeeded. If the value is less than `0` an error occurred and *errno* is set to indicate the problem.

This function is a GNU extension and should not be used in portable programs.

`int` **erand48_r** (`unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, double *`*result*) Function

The `erand48_r` function works like `erand48`, but in addition it takes an argument *buffer* that describes the random-number generator. The state of the random-number generator is taken from the `xsubi` array, the parameters for the congruential formula from the global random-number generator data. The random number is returned in the variable pointed to by *result*.

The return value is nonnegative if the call succeeded.

This function is a GNU extension and should not be used in portable programs.

`int` **lrand48_r** (`struct drand48_data *`*buffer*`, double *`*result*) Function

This function is similar to `lrand48`, but in addition it takes a pointer to a buffer describing the state of the random-number generator just like `drand48`.

If the return value of the function is nonnegative, the variable pointed to by *result* contains the result. Otherwise an error occurred.

This function is a GNU extension and should not be used in portable programs.

`int` **nrand48_r** (`unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, long int *`*result*) Function

The `nrand48_r` function works like `nrand48` in that it produces a random number in the range `0` to `2^31`. But instead of using the global parameters for the congruential formula, it uses the information from the buffer pointed to by *buffer*. The state is described by the values in *xsubi*.

If the return value is nonnegative, the variable pointed to by *result* contains the result.

This function is a GNU extension and should not be used in portable programs.

`int` **mrand48_r** (`struct drand48_data *`*buffer*`, double *`*result*) Function

This function is similar to `mrand48`, but like the other reentrant functions it uses the random-number generator described by the value in the buffer pointed to by *buffer*.

If the return value is nonnegative, the variable pointed to by *result* contains the result.

This function is a GNU extension and should not be used in portable programs.

`int` **jrand48_r** (`unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, long int *`*result*) Function

The `jrand48_r` function is similar to `jrand48`. Like the other reentrant functions of this function family, it uses the congruential formula parameters from the buffer pointed to by *buffer*.

If the return value is nonnegative, the variable pointed to by *result* contains the result.

This function is a GNU extension and should not be used in portable programs.

Before any of the above functions are used, the buffer of type `struct drand48_data` should be initialized. The easiest way to do this is to fill the whole buffer with null bytes, e.g.:

```
memset (buffer, '\0', sizeof (struct drand48_data));
```

Using any of the reentrant functions of this family now will automatically initialize the random-number generator to the default values for the state and the parameters of the congruential formula.

The other possibility is to use any of the functions that explicitly initialize the buffer. Though it might be obvious how to initialize the buffer from looking at the parameter to the function, it is highly recommended to use these functions since the result might not always be what you expect.

`int` **srand48_r** (`long int` *seedval*`, struct drand48_data *`*buffer*) Function

The description of the random-number generator represented by the information in *buffer* is initialized in a similar way to that of `srand48`. The state is initialized from the parameter *seedval*, and the parameters for the congruential formula are initialized to their default values.

If the return value is nonnegative, the function call succeeded.

This function is a GNU extension and should not be used in portable programs.

`int` **seed48_r** (`unsigned short int` *seed16v*`[3], struct drand48_data *`*buffer*) Function

This function is similar to `srand48_r`, but like `seed48` it initializes all 48 bits of the state from the parameter *seed16v*.

If the return value is nonnegative, the function call succeeded. It does not return a pointer to the previous state of the random-number generator like the `seed48`

function does. If the user wants to preserve the state for a later rerun, she can copy the whole buffer pointed to by *buffer*.

This function is a GNU extension and should not be used in portable programs.

`int` **lcong48_r** (`unsigned short int` *param*`[7], struct drand48_data *`*buffer*) Function

This function initializes all aspects of the random-number generator described in *buffer* with the data in *param*. Here it is especially true that the function does more than just copying the contents of *param* and *buffer*. More work is required and therefore it is important to use this function rather than initializing the random-number generator directly.

If the return value is nonnegative, the function call succeeded.

This function is a GNU extension and should not be used in portable programs.

8.9 Is Fast Code or Small Code Preferred?

If an application uses many floating-point functions, it is often the case that the cost of the function calls themselves is not negligible. Modern processors can often execute the operations themselves very fast, but the function call disrupts the instruction pipeline.

For this reason, the GNU C Library provides optimizations for many of the frequently used math functions. When GNU CC is used and the user activates the optimizer, several new in-line functions and macros are defined. These new functions and macros have the same names as the library functions and so are used instead of the latter. In the case of in-line functions, the compiler will decide whether it is reasonable to use them, and this decision is usually correct.

This means that calls to the library functions may be unnecessary, and can increase the speed of generated code significantly. The drawback is that code size will increase, and the increase is not always negligible.

There are two kind of in-line functions: those that give the same result as the library functions, and others that might not set `errno` and might have a reduced precision or argument range in comparison with the library functions. The latter in-line functions are only available if the flag `-ffast-math` is given to GNU CC.

In cases where the in-line functions and macros are not wanted, the symbol `__NO_MATH_INLINES` should be defined before any system header is included. This will ensure that only library functions are used. Of course, it can be determined for each file in the project whether giving this option is preferable or not.

Not all hardware implements the entire IEEE 754 standard, and even if it does there may be a substantial performance penalty for using some of its features. For example, enabling traps on some processors forces the FPU to run unpipelined, which can more than double calculation time.

9 Arithmetic Functions

This chapter contains information about functions for doing basic arithmetic operations, such as splitting a float into its integer and fractional parts or retrieving the imaginary part of a complex value. These functions are declared in the header files '`math.h`' and '`complex.h`'.

9.1 Integers

The C language defines several integer data types: integer, short integer, long integer, and character; all in both signed and unsigned varieties. The GNU C Compiler extends the language to contain long long integers as well.

The C integer types were intended to allow code to be portable among machines with different inherent data sizes (word sizes), so each type may have different ranges on different machines. The problem with this is that a program often needs to be written for a particular range of integers, and sometimes must be written for a particular size of storage, regardless of what machine the program runs on.

To address this problem, the GNU C Library contains C type definitions you can use to declare integers that meet your exact needs. Because the GNU C Library header files are customized to a specific machine, your program source code doesn't have to be.

These `typedef`s are in '`stdint.h`'.

If you require that an integer be represented in exactly N bits, use one of the following types, with the obvious mapping to bit size and signedness:

- int8_t
- int16_t
- int32_t
- int64_t
- uint8_t
- uint16_t
- uint32_t
- uint64_t

If your C compiler and target machine do not allow integers of a certain size, the corresponding above type does not exist.

If you don't need a specific storage size, but want the smallest data structure with *at least* N bits, use one of these:

- int_least8_t
- int_least16_t
- int_least32_t
- int_least64_t
- uint_least8_t

- uint_least16_t
- uint_least32_t
- uint_least64_t

If you don't need a specific storage size, but want the data structure that allows the fastest access while having at least N bits (and among data structures with the same access speed, the smallest one), use one of these:

- int_fast8_t
- int_fast16_t
- int_fast32_t
- int_fast64_t
- uint_fast8_t
- uint_fast16_t
- uint_fast32_t
- uint_fast64_t

If you want an integer with the widest range possible on the platform on which it is being used, use one of the following. If you use these, you should write code that takes into account the variable size and range of the integer.

- intmax_t
- uintmax_t

The GNU C Library also provides macros that tell you the maximum and minimum possible values for each integer data type. The macro names follow these examples: `INT32_MAX`, `UINT8_MAX`, `INT_FAST32_MIN`, `INT_LEAST64_MIN`, `UINTMAX_MAX`, `INTMAX_MAX`, `INTMAX_MIN`. There are no macros for unsigned integer minima. These are always zero.

There are similar macros for use with C's built in integer types that should come with your C compiler.[1]

Don't forget that you can use the C `sizeof` function with any of these data types to get the number of bytes of storage each uses.

9.2 Integer Division

This section describes functions for performing integer division. These functions are redundant when GNU CC is used, because in GNU C the '`/`' operator always rounds toward zero. But in other C implementations, '`/`' may round differently with negative arguments. `div` and `ldiv` are useful because they specify how to round the quotient—toward zero. The remainder has the same sign as the numerator.

These functions are specified to return a result *r* such that the value *r*`.quot`**denominator* + *r*`.rem` equals *numerator*.

To use these facilities, you should include the header file '`stdlib.h`' in your program.

[1] See Loosemore, et al., "Data Type Measurements" (see chap. 1, n. 1).

div_t Data Type

This is a structure type used to hold the result returned by the `div` function. It has the following members:

`int quot` The quotient from the division

`int rem` The remainder from the division

`div_t` **div** (`int` *numerator*, `int` *denominator*) Function

This function `div` computes the quotient and remainder from the division of *numerator* by *denominator*, returning the result in a structure of type `div_t`.

If the result cannot be represented (as in a division by zero), the behavior is undefined.

Here is an example, albeit not a very useful one:

```
div_t result;
result = div (20, -6);
```

Now `result.quot` is `-3` and `result.rem` is `2`.

ldiv_t Data Type

This is a structure type used to hold the result returned by the `ldiv` function. It has the following members:

`long int quot`
The quotient from the division

`long int rem`
The remainder from the division

This is identical to `div_t`, except that the components are of type `long int` rather than `int`.

`ldiv_t` **ldiv** (`long int` *numerator*, `long int` *denominator*) Function

The `ldiv` function is similar to `div`, except that the arguments are of type `long int` and the result is returned as a structure of type `ldiv_t`.

lldiv_t Data Type

This is a structure type used to hold the result returned by the `lldiv` function. It has the following members:

`long long int quot`
The quotient from the division

`long long int rem`
The remainder from the division

This is identical to `div_t`, except that the components are of type `long long int` rather than `int`.

`lldiv_t` **lldiv** (`long long int` *numerator*, `long long int` *denominator*) Function

The `lldiv` function is like the `div` function, but the arguments are of type `long long int` and the result is returned as a structure of type `lldiv_t`.

The `lldiv` function was added in ISO C99.

imaxdiv_t Data Type

This is a structure type used to hold the result returned by the `imaxdiv` function. It has the following members:

`intmax_t quot`
: The quotient from the division

`intmax_t rem`
: The remainder from the division

This is identical to `div_t`, except that the components are of type `intmax_t` rather than `int` (see Section 9.1 [Integers], page 243 for a description of the `intmax_t` type).

`imaxdiv_t` **imaxdiv** (`intmax_t` *numerator*, `intmax_t` *denominator*) Function

The `imaxdiv` function is like the `div` function, but the arguments are of type `intmax_t` and the result is returned as a structure of type `imaxdiv_t` (see Section 9.1 [Integers], page 243 for a description of the `intmax_t` type).

The `imaxdiv` function was added in ISO C99.

9.3 Floating-Point Numbers

Most computer hardware has support for two different kinds of numbers: integers ($\ldots -3, -2, -1, 0, 1, 2, 3 \ldots$) and floating-point numbers. Floating-point numbers have three parts: the *mantissa*, the *exponent* and the *sign bit*. The real number represented by a floating-point value is given by $(s\ ?\ -1 : 1) \cdot 2^e \cdot M$ where s is the sign bit, e the exponent, and M the mantissa.[2] It is possible to have a different *base* for the exponent, but all modern hardware uses 2.

Floating-point numbers can represent a finite subset of the real numbers. While this subset is large enough for most purposes, it is important to remember that the only reals that can be represented exactly are rational numbers that have a terminating binary expansion shorter than the width of the mantissa. Even simple fractions such as $1/5$ can only be approximated by floating point.

Mathematical operations and functions frequently need to produce values that are not representable. Often these values can be approximated closely enough for practical purposes, but sometimes they can't. Historically there was no way to tell when the results of a calculation were inaccurate. Modern computers implement

[2] Ibid., "Floating-Point Representation Concepts".

the IEEE 754 standard for numerical computations, which defines a framework for indicating to the program when the results of calculation are not trustworthy. This framework consists of a set of *exceptions* that indicate why a result could not be represented, and the special values *infinity* and *not a number* (NaN).

9.4 Floating-Point Number Classification Functions

ISO C99 defines macros that let you determine what sort of floating-point number a variable holds.

`int` **fpclassify** (*float-type x*) Macro

This is a generic macro that works on all floating-point types and that returns a value of type `int`. The possible values are

`FP_NAN` The floating-point number *x* is *Not a Number* (see Section 9.5.2 [Infinity and NaN], page 250).

`FP_INFINITE` The value of *x* is either plus or minus infinity (see Section 9.5.2 [Infinity and NaN], page 250).

`FP_ZERO` The value of *x* is zero. In floating-point formats like IEEE 754, where zero can be signed, this value is also returned if *x* is negative zero.

`FP_SUBNORMAL` Numbers whose absolute value is too small to be represented in the normal format are represented in an alternate, *de-normalized* format.[3] This format is less precise but can represent values closer to zero. `fpclassify` returns this value for values of *x* in this alternate format.

`FP_NORMAL` This value is returned for all other values of *x*. It indicates that there is nothing special about the number.

`fpclassify` is most useful if more than one property of a number must be tested. There are more specific macros, which only test one property at a time. Generally these macros execute faster than `fpclassify`, since there is special hardware support for them. You should therefore use the specific macros whenever possible.

`int` **isfinite** (*float-type x*) Macro

This macro returns a nonzero value if *x* is finite—not plus or minus infinity, and not NaN. It is equivalent to:

[3] Ibid., "Floating-Point Representation Concepts".

```
(fpclassify (x) != FP_NAN && fpclassify (x) != FP_INFINITE)
```

`isfinite` is implemented as a macro that accepts any floating-point type.

`int` **isnormal** (*float-type x*) Macro

This macro returns a nonzero value if *x* is finite and normalized. It is equivalent to:

```
(fpclassify (x) == FP_NORMAL)
```

`int` **isnan** (*float-type x*) Macro

This macro returns a nonzero value if *x* is NaN. It is equivalent to:

```
(fpclassify (x) == FP_NAN)
```

Another set of floating-point classification functions was provided by BSD. The GNU C Library also supports these functions; however, we recommend that you use the ISO C99 macros in new code. Those are standard and will be available more widely. Also, since they are macros, you do not have to worry about the type of their argument.

`int` **isinf** (`double` *x*) Function
`int` **isinff** (`float` *x*) Function
`int` **isinfl** (`long double` *x*) Function

This function returns `-1` if *x* represents negative infinity, `1` if *x* represents positive infinity, and `0` otherwise.

`int` **isnan** (`double` *x*) Function
`int` **isnanf** (`float` *x*) Function
`int` **isnanl** (`long double` *x*) Function

This function returns a nonzero value if *x* is a *Not a Number* value, and zero otherwise.

The `isnan` macro defined by ISO C99 overrides the BSD function. This is normally not a problem, because the two routines behave identically. However, if you really need to get the BSD function for some reason, you can write:

```
(isnan) (x)
```

`int` **finite** (`double` *x*) Function
`int` **finitef** (`float` *x*) Function
`int` **finitel** (`long double` *x*) Function

This function returns a nonzero value if *x* is finite or a *Not a Number* value, and zero otherwise.

Portability Note: The functions listed in this section are BSD extensions.

9.5 Errors in Floating-Point Calculations

9.5.1 FP Exceptions

The IEEE 754 standard defines five *exceptions* that can occur during a calculation. Each corresponds to a particular sort of error, such as overflow.

When exceptions occur (when exceptions are *raised*, in the language of the standard), one of two things can happen. By default, the exception is simply noted in the floating-point *status word*, and the program continues as if nothing had happened. The operation produces a default value, which depends on the exception (see the table below). Your program can check the status word to find out which exceptions happened.

Alternatively, you can enable *traps* for exceptions. In that case, when an exception is raised, your program will receive the `SIGFPE` signal. The default action for this signal is to terminate the program.[4]

In the System V math library, the user-defined function `matherr` is called when certain exceptions occur inside math library functions. However, the Unix98 standard deprecates this interface. We support it for historical compatibility, but recommend that you do not use it in new programs.

The exceptions defined in IEEE 754 are

'`Invalid Operation`'
: This exception is raised if the given operands are invalid for the operation to be performed. Examples are (see IEEE 754, section 7):

  1. Addition or subtraction: $\infty - \infty$ (but $\infty + \infty = \infty$)
  2. Multiplication: $0 \cdot \infty$
  3. Division: $0/0$ or ∞/∞
  4. Remainder: x REM y, where y is zero or x is infinite
  5. Square root if the operand is less then zero; more generally, any mathematical function evaluated outside its domain produces this exception
  6. Conversion of a floating-point number to an integer or decimal string, when the number cannot be represented in the target format (due to overflow, infinity, or NaN)
  7. Conversion of an unrecognizable input string
  8. Comparison via predicates involving $<$ or $>$, when one or the other of the operands is NaN; you can prevent this exception by using the unordered comparison functions instead (see Section 9.8.6 [Floating-Point Comparison Functions], page 264)

  If the exception does not trap, the result of the operation is NaN.

[4] For how you can change the effect of the signal, see Loosemore et al., "Signal Handling".

'`Division by Zero`'
: This exception is raised when a finite nonzero number is divided by zero. If no trap occurs, the result is either $+\infty$ or $-\infty$, depending on the signs of the operands.

'`Overflow`'
: This exception is raised whenever the result cannot be represented as a finite value in the precision format of the destination. If no trap occurs, the result depends on the sign of the intermediate result and the current rounding mode (IEEE 754, section 7.3):

  1. Round to nearest carries all overflows to ∞ with the sign of the intermediate result.
  2. Round toward 0 carries all overflows to the largest representable finite number with the sign of the intermediate result.
  3. Round toward $-\infty$ carries positive overflows to the largest representable finite number and negative overflows to $-\infty$.
  4. Round toward ∞ carries negative overflows to the most negative representable finite number and positive overflows to ∞.

  Whenever the overflow exception is raised, the inexact exception is also raised.

'`Underflow`'
: The underflow exception is raised when an intermediate result is too small to be calculated accurately, or if the operation's result rounded to the destination precision is too small to be normalized.

  When no trap is installed for the underflow exception, underflow is signaled (via the underflow flag) only when both tininess and loss of accuracy have been detected. If no trap handler is installed, the operation continues with an imprecise small value, or zero if the destination precision cannot hold the small exact result.

'`Inexact`'
: This exception is signalled if a rounded result is not exact (such as when calculating the square root of two) or a result overflows without an overflow trap.

9.5.2 Infinity and NaN

IEEE 754 floating-point numbers can represent positive or negative infinity, and *NaN* (not a number). These three values arise from calculations whose result is undefined or cannot be represented accurately. You can also deliberately set a floating-point variable to any of them, which is sometimes useful. Some examples of calculations that produce infinity or NaN:

$$\frac{1}{0} = \infty$$

$$\log 0 = -\infty$$

$$\sqrt{-1} = \text{NaN}$$

When a calculation produces any of these values, an exception also occurs (see Section 9.5.1 [FP Exceptions], page 249).

The basic operations and math functions all accept infinity and NaN and produce sensible output. Infinities propagate through calculations as one would expect: for example, $2 + \infty = \infty$, $4/\infty = 0$, atan $(\infty) = \pi/2$. NaN, on the other hand, infects any calculation that involves it. Unless the calculation would produce the same result no matter what real value replaced NaN, the result is NaN.

In comparison operations, positive infinity is larger than all values except itself and NaN, and negative infinity is smaller than all values except itself and NaN. NaN is *unordered*: it is not equal to, greater than, or less than anything, *including itself*. `x == x` is false if the value of `x` is NaN. You can use this to test whether a value is NaN or not, but the recommended way to test for NaN is with the `isnan` function (see Section 9.4 [Floating-Point Number Classification Functions], page 247). In addition, <, >, <= and >= will raise an exception when applied to NaNs.

'`math.h`' defines macros that allow you to explicitly set a variable to infinity or NaN.

`float` **INFINITY** Macro

An expression representing positive infinity. It is equal to the value produced by mathematical operations like `1.0 / 0.0`. `-INFINITY` represents negative infinity.

You can test whether a floating-point value is infinite by comparing it to this macro. However, this is not recommended; you should use the `isfinite` macro instead (see Section 9.4 [Floating-Point Number Classification Functions], page 247).

This macro was introduced in the ISO C99 standard.

`float` **NAN** Macro

An expression representing a value which is *not a number*. This macro is a GNU extension, available only on machines that support the *not a number* value—that is to say, on all machines that support IEEE floating point.

You can use '`#ifdef NAN`' to test whether the machine supports NaN. (Of course, you must arrange for GNU extensions to be visible, such as by defining `_GNU_SOURCE`, and then you must include '`math.h`'.)

IEEE 754 also allows for another unusual value: negative zero. This value is produced when you divide a positive number by negative infinity, or when a negative result is smaller than the limits of representation. Negative zero behaves identically to zero in all calculations, unless you explicitly test the sign bit with `signbit` or `copysign`.

9.5.3 Examining the FPU Status Word

ISO C99 defines functions to query and manipulate the floating-point status word. You can use these functions to check for untrapped exceptions when it's convenient, rather than worrying about them in the middle of a calculation.

These constants represent the various IEEE 754 exceptions. Not all FPUs report all the different exceptions. Each constant is defined if and only if the FPU you are compiling for supports that exception, so you can test for FPU support with '`#ifdef`'. They are defined in '`fenv.h`'.

`FE_INEXACT`
: The inexact exception

`FE_DIVBYZERO`
: The divide by zero exception

`FE_UNDERFLOW`
: The underflow exception

`FE_OVERFLOW`
: The overflow exception

`FE_INVALID`
: The invalid exception

The macro `FE_ALL_EXCEPT` is the bit-wise OR of all exception macros that are supported by the FP implementation.

These functions allow you to clear exception flags, test for exceptions, and save and restore the set of exceptions flagged.

`int` **feclearexcept** (`int` *excepts*) Function

This function clears all of the supported exception flags indicated by *excepts*.

The function returns zero when the operation was successful and a nonzero value otherwise.

`int` **feraiseexcept** (`int` *excepts*) Function

This function raises the supported exceptions indicated by *excepts*. If more than one exception bit in *excepts* is set, the order in which the exceptions are raised is undefined except that overflow (`FE_OVERFLOW`) or underflow (`FE_UNDERFLOW`) are raised before inexact (`FE_INEXACT`). For overflow or underflow, whether the inexact exception is also raised is implementation dependent.

The function returns zero when the operation was successful and a nonzero value otherwise.

`int` **fetestexcept** (`int` *excepts*) Function

Test whether the exception flags indicated by the parameter *except* are currently set. If any of them are, a nonzero value is returned, which specifies which exceptions are set. Otherwise, the result is zero.

To understand these functions, imagine that the status word is an integer variable named *status*. `feclearexcept` is then equivalent to '`status &= ~excepts`' and `fetestexcept` is equivalent to '`(status & excepts)`'. The actual implementation may be very different, of course.

Exception flags are only cleared when the program explicitly requests it, by calling `feclearexcept`. If you want to check for exceptions from a set of calculations, you should clear all the flags first. Here is a simple example of the way to use `fetestexcept`:

```
{
  double f;
  int raised;
  feclearexcept (FE_ALL_EXCEPT);
  f = compute ();
  raised = fetestexcept (FE_OVERFLOW | FE_INVALID);
  if (raised & FE_OVERFLOW) { /* ... */ }
  if (raised & FE_INVALID) { /* ... */ }
  /* ... */
}
```

You cannot explicitly set bits in the status word. You can, however, save the entire status word and restore it later. This is done with the following functions:

`int` **fegetexceptflag** (`fexcept_t *`*flagp*`, int` *excepts*) Function

This function stores in the variable pointed to by *flagp* an implementation-defined value representing the current setting of the exception flags indicated by *excepts*.

The function returns zero if the operation was successful and a nonzero value otherwise.

`int` **fesetexceptflag** (`const fexcept_t *`*flagp*`, int` *excepts*) Function

This function restores the flags for the exceptions indicated by *excepts* to the values stored in the variable pointed to by *flagp*.

The function returns zero if the operation was successful and a nonzero value otherwise.

The value stored in `fexcept_t` bears no resemblance to the bit mask returned by `fetestexcept`. The type may not even be an integer. Do not attempt to modify an `fexcept_t` variable.

9.5.4 Error Reporting by Mathematical Functions

Many of the math functions are defined only over a subset of the real or complex numbers. Even if they are mathematically defined, their result may be larger or smaller than the range representable by their return type. These are known as *domain errors*, *overflows*, and *underflows*, respectively. Math functions do several

things when one of these errors occurs. In this manual we will refer to the complete response as *signalling* a domain error, overflow, or underflow.

When a math function suffers a domain error, it raises the invalid exception and returns NaN. It also sets *errno* to `EDOM`; this is for compatibility with old systems that do not support IEEE 754 exception handling. Likewise, when overflow occurs, math functions raise the overflow exception and return ∞ or $-\infty$ as appropriate. They also set *errno* to `ERANGE`. When underflow occurs, the underflow exception is raised, and zero (appropriately signed) is returned. *errno* may be set to `ERANGE`, but this is not guaranteed.

Some of the math functions are defined mathematically to result in a complex value over parts of their domains. The most familiar example of this is taking the square root of a negative number. The complex math functions, such as `csqrt`, will return the appropriate complex value in this case. The real-valued functions, such as `sqrt`, will signal a domain error.

Some older hardware does not support infinities. On that hardware, overflows instead return a particular very large number (usually the largest representable number). '`math.h`' defines macros you can use to test for overflow on both old and new hardware.

`double` **HUGE_VAL** Macro
`float` **HUGE_VALF** Macro
`long double` **HUGE_VALL** Macro

An expression representing a particular very large number. On machines that use IEEE 754 floating-point format, `HUGE_VAL` is infinity. On other machines, it's typically the largest positive number that can be represented.

Mathematical functions return the appropriately typed version of `HUGE_VAL` or `-HUGE_VAL` when the result is too large to be represented.

9.6 Rounding Modes

Floating-point calculations are carried out internally with extra precision, and then rounded to fit into the destination type. This ensures that results are as precise as the input data. IEEE 754 defines four possible rounding modes:

Round to nearest

This is the default mode. It should be used unless there is a specific need for one of the others. In this mode, results are rounded to the nearest representable value. If the result is midway between two representable values, the even representable is chosen. *Even* here means the lowest-order bit is zero. This rounding mode prevents statistical bias and guarantees numeric stability—round-off errors in a lengthy calculation will remain smaller than half of `FLT_EPSILON`.

Round toward plus Infinity.

All results are rounded to the smallest representable value that is greater than the result.

Round toward minus Infinity.
: All results are rounded to the largest representable value that is less than the result.

Round toward zero
: All results are rounded to the largest representable value whose magnitude is less than that of the result. In other words, if the result is negative, it is rounded up; if it is positive, it is rounded down.

'`fenv.h`' defines constants which you can use to refer to the various rounding modes. Each one will be defined if and only if the FPU supports the corresponding rounding mode.

`FE_TONEAREST`
: Round to nearest

`FE_UPWARD`
: Round toward $+\infty$

`FE_DOWNWARD`
: Round toward $-\infty$

`FE_TOWARDZERO`
: Round toward zero

Underflow is an unusual case. Normally, IEEE 754 floating-point numbers are always normalized.[5] Numbers smaller than 2^r (where r is the minimum exponent, `FLT_MIN_RADIX-1` for *float*) cannot be represented as normalized numbers. Rounding all such numbers to zero or 2^r would cause some algorithms to fail at 0. Therefore, they are left in de-normalized form. That produces loss of precision, since some bits of the mantissa are stolen to indicate the decimal point.

If a result is too small to be represented as a de-normalized number, it is rounded to zero. However, the sign of the result is preserved; if the calculation was negative, the result is *negative zero*. Negative zero can also result from some operations on infinity, such as $4/-\infty$. Negative zero behaves identically to zero except when the `copysign` or `signbit` functions are used to check the sign bit directly.

At any time one of the above four rounding modes is selected. You can find out which one with this function:

`int` **fegetround** (`void`) Function
: Returns the currently selected rounding mode, represented by one of the values of the defined rounding mode macros.

To change the rounding mode, use this function:

`int` **fesetround** (`int` *round*) Function
: Changes the currently selected rounding mode to *round*. If *round* does not correspond to one of the supported rounding modes nothing is changed.

[5] Ibid., "Floating-Point Representation Concepts".

`fesetround` returns zero if it changed the rounding mode and a nonzero value if the mode is not supported.

You should avoid changing the rounding mode if possible. It can be an expensive operation; also, some hardware requires you to compile your program differently for it to work. The resulting code may run slower. See your compiler documentation for details.

9.7 Floating-Point Control Functions

IEEE 754 floating-point implementations allow the programmer to decide whether traps will occur for each of the exceptions, by setting bits in the *control word*. In C, traps result in the program receiving the `SIGFPE` signal.[6]

IEEE 754 says that trap handlers are given details of the exceptional situation and can set the result value. C signals do not provide any mechanism to pass this information back and forth. Trapping exceptions in C is therefore not very useful.

It is sometimes necessary to save the state of the floating-point unit while you perform some calculation. The library provides functions that save and restore the exception flags, the set of exceptions that generate traps, and the rounding mode. This information is known as the *floating-point environment*.

The functions to save and restore the floating-point environment all use a variable of type `fenv_t` to store information. This type is defined in '`fenv.h`'. Its size and contents are implementation defined. You should not attempt to manipulate a variable of this type directly.

To save the state of the FPU, use one of these functions:

`int` **fegetenv** (`fenv_t *`*envp*) Function

Store the floating-point environment in the variable pointed to by *envp*.

The function returns zero if the operation was successful and a nonzero value otherwise.

`int` **feholdexcept** (`fenv_t *`*envp*) Function

Store the current floating-point environment in the object pointed to by *envp*. Then clear all exception flags, and set the FPU to trap no exceptions. Not all FPUs support trapping no exceptions; if `feholdexcept` cannot set this mode, it returns a nonzero value. If it succeeds, it returns zero.

The functions that restore the floating-point environment can take these kinds of arguments:

- Pointers to `fenv_t` objects that were initialized previously by a call to `fegetenv` or `feholdexcept`.
- The special macro `FE_DFL_ENV`, which represents the floating-point environment as it was available at program start.

[6] Ibid., "Signal Handling".

- Implementation-defined macros with names starting with `FE_` and having type `fenv_t *`.

  If possible, the GNU C Library defines a macro `FE_NOMASK_ENV`, which represents an environment where every exception raised causes a trap to occur. You can test for this macro using `#ifdef`. It is only defined if `_GNU_SOURCE` is defined.

  Some platforms might define other predefined environments.

To set the floating-point environment, you can use either of these functions:

`int` **fesetenv** (`const fenv_t *`*envp*) Function

Set the floating-point environment to that described by *envp*.

The function returns zero if the operation was successful and a nonzero value otherwise.

`int` **feupdateenv** (`const fenv_t *`*envp*) Function

Like `fesetenv`, this function sets the floating-point environment to that described by *envp*. However, if any exceptions were flagged in the status word before `feupdateenv` was called, they remain flagged after the call. In other words, after `feupdateenv` is called, the status word is the bit-wise OR of the previous status word and the one saved in *envp*.

The function returns zero if the operation was successful and a nonzero value otherwise.

To control for individual exceptions if raising them causes a trap to occur, you can use the following two functions.

Portability Note: These functions are all GNU extensions.

`int` **feenableexcept** (`int` *excepts*) Function

This function enables traps for each of the exceptions as indicated by the parameter *except*. The individual exceptions are described in Section 9.5.3 [Examining the FPU Status Word], page 252. Only the specified exceptions are enabled—the status of the other exceptions is not changed.

The function returns the previously enabled exceptions if the operation was successful, `-1` otherwise.

`int` **fedisableexcept** (`int` *excepts*) Function

This function disables traps for each of the exceptions as indicated by the parameter *except*. The individual exceptions are described in Section 9.5.3 [Examining the FPU Status Word], page 252. Only the specified exceptions are disabled—the status of the other exceptions is not changed.

The function returns the previously enabled exceptions if the operation was successful and `-1` otherwise.

`int` **fegetexcept** (`int` *excepts*) Function
The function returns a bitmask of all currently enabled exceptions. It returns `-1` in case of failure.

9.8 Arithmetic Functions

The C library provides functions to do basic operations on floating-point numbers. These include absolute value, maximum and minimum, normalization, bit twiddling, rounding and a few others.

9.8.1 Absolute Value

These functions are provided for obtaining the *absolute value* (or *magnitude*) of a number. The absolute value of a real number x is x if x is positive, $-x$ if x is negative. For a complex number z, whose real part is x and whose imaginary part is y, the absolute value is `sqrt (x*x + y*y)`.

Prototypes for `abs`, `labs` and `llabs` are in '`stdlib.h`'; `imaxabs` is declared in '`inttypes.h`'; `fabs`, `fabsf` and `fabsl` are declared in '`math.h`'. `cabs`, `cabsf` and `cabsl` are declared in '`complex.h`'.

`int` **abs** (`int` *number*) Function
`long int` **labs** (`long int` *number*) Function
`long long int` **llabs** (`long long int` *number*) Function
`intmax_t` **imaxabs** (`intmax_t` *number*) Function
These functions return the absolute value of *number*.

Most computers use a two's complement integer representation, in which the absolute value of `INT_MIN` (the smallest possible `int`) cannot be represented; thus, `abs (INT_MIN)` is not defined.

`llabs` and `imaxdiv` are new to ISO C99.

See Section 9.1 [Integers], page 243 for a description of the `intmax_t` type.

`double` **fabs** (`double` *number*) Function
`float` **fabsf** (`float` *number*) Function
`long double` **fabsl** (`long double` *number*) Function
This function returns the absolute value of the floating-point number *number*.

`double` **cabs** (`complex double` *z*) Function
`float` **cabsf** (`complex float` *z*) Function
`long double` **cabsl** (`complex long double` *z*) Function
These functions return the absolute value of the complex number *z* (see Section 9.9 [Complex Numbers], page 266). The absolute value of a complex number is

```
sqrt (creal (z) * creal (z) + cimag (z) * cimag (z))
```

This function should always be used instead of the direct formula because it takes special care to avoid losing precision. It may also take advantage of hardware support for this operation (see `hypot` in Section 8.4 [Exponentiation and Logarithms], page 207).

9.8.2 Normalization Functions

The functions described in this section are primarily provided as a way to efficiently perform certain low-level manipulations on floating-point numbers that are represented internally using a binary radix.[7] These functions are required to have equivalent behavior even if the representation does not use a radix of 2, but of course they are unlikely to be particularly efficient in those cases.

All these functions are declared in '`math.h`'.

`double` **frexp** (`double` *value*, `int` ⋆*exponent*) Function
`float` **frexpf** (`float` *value*, `int` ⋆*exponent*) Function
`long double` **frexpl** (`long double` *value*, `int` ⋆*exponent*) Function

These functions are used to split the number *value* into a normalized fraction and an exponent.

If the argument *value* is not zero, the return value is *value* times a power of two, and is always in the range 1/2 (inclusive) to 1 (exclusive). The corresponding exponent is stored in ⋆*exponent*; the return value multiplied by 2 raised to this exponent equals the original number *value*.

For example, `frexp (12.8, &exponent)` returns `0.8` and stores `4` in `exponent`.

If *value* is zero, then the return value is zero, and zero is stored in ⋆*exponent*.

`double` **ldexp** (`double` *value*, `int` *exponent*) Function
`float` **ldexpf** (`float` *value*, `int` *exponent*) Function
`long double` **ldexpl** (`long double` *value*, `int` *exponent*) Function

These functions return the result of multiplying the floating-point number *value* by 2 raised to the power *exponent*. (It can be used to reassemble floating-point numbers that were taken apart by `frexp`.)

For example, `ldexp (0.8, 4)` returns `12.8`.

The following functions, which come from BSD, provide facilities equivalent to those of `ldexp` and `frexp`. See also the ISO C function `logb`, which originally also appeared in BSD.

`double` **scalb** (`double` *value*, `int` *exponent*) Function
`float` **scalbf** (`float` *value*, `int` *exponent*) Function
`long double` **scalbl** (`long double` *value*, `int` *exponent*) Function

The `scalb` function is the BSD name for `ldexp`.

[7] Ibid., "Floating-Point Representation Concepts".

`long long int` **scalbn** (`double` *x*, `int` `n`) Function
`long long int` **scalbnf** (`float` *x*, `int` `n`) Function
`long long int` **scalbnl** (`long double` *x*, `int` `n`) Function
`scalbn` is identical to `scalb`, except that the exponent *n* is an `int` instead of a floating-point number.

`long long int` **scalbln** (`double` *x*, `long int` `n`) Function
`long long int` **scalblnf** (`float` *x*, `long int` `n`) Function
`long long int` **scalblnl** (`long double` *x*, `long int` `n`) Function
`scalbln` is identical to `scalb`, except that the exponent *n* is a `long int` instead of a floating-point number.

`long long int` **significand** (`double` *x*) Function
`long long int` **significandf** (`float` *x*) Function
`long long int` **significandl** (`long double` *x*) Function
`significand` returns the mantissa of *x* scaled to the range $[1, 2)$. It is equivalent to `scalb (x, (double) -ilogb (x))`.

This function exists mainly for use in certain standardized tests of IEEE 754 conformance.

9.8.3 Rounding Functions

The functions listed here perform operations such as rounding and truncation of floating-point values. Some of these functions convert floating-point numbers to integer values. They are all declared in '`math.h`'.

You can also convert floating-point numbers to integers simply by casting them to `int`. This discards the fractional part, effectively rounding toward zero. However, this only works if the result can actually be represented as an `int`—for very large numbers, this is impossible. The functions listed here return the result as a `double` instead to get around this problem:

`double` **ceil** (`double` *x*) Function
`float` **ceilf** (`float` *x*) Function
`long double` **ceill** (`long double` *x*) Function
These functions round *x* upward to the nearest integer, returning that value as a `double`. Thus, `ceil (1.5)` is `2.0`.

`double` **floor** (`double` *x*) Function
`float` **floorf** (`float` *x*) Function
`long double` **floorl** (`long double` *x*) Function
These functions round *x* downward to the nearest integer, returning that value as a `double`. Thus, `floor (1.5)` is `1.0` and `floor (-1.5)` is `-2.0`.

`double` **trunc** (`double` *x*) Function
`float` **truncf** (`float` *x*) Function
`long double` **truncl** (`long double` *x*) Function
The `trunc` functions round *x* toward zero to the nearest integer (returned in floating-point format). Thus, `trunc (1.5)` is `1.0` and `trunc (-1.5)` is `-1.0`.

`double` **rint** (`double` *x*) Function
`float` **rintf** (`float` *x*) Function
`long double` **rintl** (`long double` *x*) Function
These functions round *x* to an integer value according to the current rounding mode.[8] The default rounding mode is to round to the nearest integer; some machines support other modes, but round-to-nearest is always used unless you explicitly select another.

If *x* was not initially an integer, these functions raise the inexact exception.

`double` **nearbyint** (`double` *x*) Function
`float` **nearbyintf** (`float` *x*) Function
`long double` **nearbyintl** (`long double` *x*) Function
These functions return the same value as the `rint` functions, but do not raise the inexact exception if *x* is not an integer.

`double` **round** (`double` *x*) Function
`float` **roundf** (`float` *x*) Function
`long double` **roundl** (`long double` *x*) Function
These functions are similar to `rint`, but they round halfway cases away from zero instead of to the nearest even integer.

`long int` **lrint** (`double` *x*) Function
`long int` **lrintf** (`float` *x*) Function
`long int` **lrintl** (`long double` *x*) Function
These functions are just like `rint`, but they return a `long int` instead of a floating-point number.

`long long int` **llrint** (`double` *x*) Function
`long long int` **llrintf** (`float` *x*) Function
`long long int` **llrintl** (`long double` *x*) Function
These functions are just like `rint`, but they return a `long long int` instead of a floating-point number.

---

[8] See Loosemore et al., "Floating-Point Parameters", for information about the various rounding modes.

`long int` **lround** (`double` *x*) Function
`long int` **lroundf** (`float` *x*) Function
`long int` **lroundl** (`long double` *x*) Function
These functions are just like `round`, but they return a `long int` instead of a floating-point number.

`long long int` **llround** (`double` *x*) Function
`long long int` **llroundf** (`float` *x*) Function
`long long int` **llroundl** (`long double` *x*) Function
These functions are just like `round`, but they return a `long long int` instead of a floating-point number.

`double` **modf** (`double` *value*, `double` ⋆*integer-part*) Function
`float` **modff** (`float` *value*, `float` ⋆*integer-part*) Function
`long double` **modfl** (`long double` *value*, `long double` ⋆*integer-part*) Function
These functions break the argument *value* into an integer part and a fractional part (between `-1` and `1`, exclusive). Their sum equals *value*. Each of the parts has the same sign as *value*, and the integer part is always rounded toward zero.

`modf` stores the integer part in ⋆*integer-part* and returns the fractional part. For example, `modf (2.5, &intpart)` returns `0.5` and stores `2.0` into `intpart`.

9.8.4 Remainder Functions

The functions in this section compute the remainder on division of two floating-point numbers. Each is a little different; pick the one that suits your problem.

`double` **fmod** (`double` *numerator*, `double` *denominator*) Function
`float` **fmodf** (`float` *numerator*, `float` *denominator*) Function
`long double` **fmodl** (`long double` *numerator*, `long double` *denominator*) Function
These functions compute the remainder from the division of *numerator* by *denominator*. Specifically, the return value is *numerator* − *n* ⋆ *denominator*, where *n* is the quotient of *numerator* divided by *denominator*, rounded toward zero to an integer. Thus, `fmod (6.5, 2.3)` returns `1.9`, which is `6.5` minus `4.6`.

The result has the same sign as the *numerator* and has magnitude less than the magnitude of the *denominator*.

If *denominator* is zero, `fmod` signals a domain error.

`double` **drem** (`double` *numerator*, `double` *denominator*) Function
`float` **dremf** (`float` *numerator*, `float` *denominator*) Function
`long double` **dreml** (`long double` *numerator*, `long double` *denominator*) Function

These functions are like `fmod` except that they round the internal quotient *n* to the nearest integer instead of toward zero to an integer. For example, `drem (6.5, 2.3)` returns `-0.4`, which is `6.5` minus `6.9`.

The absolute value of the result is less than or equal to half the absolute value of the *denominator*. The difference between `fmod (`*numerator*`,` *denominator*`)` and `drem (`*numerator*`,` *denominator*`)` is always either *denominator*, minus *denominator*, or zero.

If *denominator* is zero, `drem` signals a domain error.

`double` **remainder** (`double` *numerator*, `double` *denominator*) Function
`float` **remainderf** (`float` *numerator*, `float` *denominator*) Function
`long double` **remainderl** (`long double` *numerator*, `long double` *denominator*) Function

This function is another name for `drem`.

9.8.5 Setting and Modifying Single Bits of FP Values

There are some operations that are too complicated or expensive to perform by hand on floating-point numbers. ISO C99 defines functions to do these operations, which mostly involve changing single bits.

`double` **copysign** (`double` *x*, `double` *y*) Function
`float` **copysignf** (`float` *x*, `float` *y*) Function
`long double` **copysignl** (`long double` *x*, `long double` *y*) Function

These functions return *x* but with the sign of *y*. They work even if *x* or *y* are NaN or zero. Both of these can carry a sign (although not all implementations support it), and this is one of the few operations that can tell the difference.

`copysign` never raises an exception.

This function is defined in IEC 559 (and the appendix with recommended functions in IEEE 754/IEEE 854).

`int` **signbit** (*float-type x*) Function

`signbit` is a generic macro that can work on all floating-point types. It returns a nonzero value if the value of *x* has its sign bit set.

This is not the same as `x < 0.0`, because IEEE 754 floating-point allows zero to be signed. The comparison `-0.0 < 0.0` is false, but `signbit (-0.0)` will return a nonzero value.

`double` **nextafter** (`double` *x*, `double` *y*) Function
`float` **nextafterf** (`float` *x*, `float` *y*) Function
`long double` **nextafterl** (`long double` *x*, `long double` *y*) Function

The `nextafter` function returns the next representable neighbor of *x* in the direction toward *y*. The size of the step between *x* and the result depends on the type of the result. If *x* = *y*, the function simply returns *y*. If either value is `NaN`, `NaN` is returned. Otherwise, a value corresponding to the value of the least significant bit in the mantissa is added or subtracted, depending on the direction. `nextafter` will signal overflow or underflow if the result goes outside of the range of normalized numbers.

This function is defined in IEC 559 (and the appendix with recommended functions in IEEE 754/IEEE 854).

`double` **nexttoward** (`double` *x*, `long double` *y*) Function
`float` **nexttowardf** (`float` *x*, `long double` *y*) Function
`long double` **nexttowardl** (`long double` *x*, `long double` *y*) Function

These functions are identical to the corresponding versions of `nextafter` except that their second argument is a `long double`.

`double` **nan** (`const char` ⋆*tagp*) Function
`float` **nanf** (`const char` ⋆*tagp*) Function
`long double` **nanl** (`const char` ⋆*tagp*) Function

The `nan` function returns a representation of NaN, provided that NaN is supported by the target platform. `nan ("`*n-char-sequence*`")` is equivalent to `strtod ("NAN(`*n-char-sequence*`)")`.

The argument *tagp* is used in an unspecified manner. On IEEE 754 systems, there are many representations of NaN, and *tagp* selects one. On other systems it may do nothing.

9.8.6 Floating-Point Comparison Functions

The standard C comparison operators provoke exceptions when one or other of the operands is NaN. For example:

```
int v = a < 1.0;
```

will raise an exception if *a* is NaN (this does *not* happen with == and !=; those merely return false and true, respectively, when NaN is examined). Frequently, this exception is undesirable. ISO C99 therefore defines comparison functions that do not raise exceptions when NaN is examined. All of the functions are implemented as macros that allow their arguments to be of any floating-point type. The macros are guaranteed to evaluate their arguments only once.

`int` **isgreater** (*real-floating x, real-floating y*) Macro
This macro determines whether the argument *x* is greater than *y*. It is equivalent to `(x) > (y)`, but no exception is raised if *x* or *y* are NaN.

`int` **isgreaterequal** (*real-floating x, real-floating y*) Macro
This macro determines whether the argument *x* is greater than or equal to *y*. It is equivalent to `(x) >= (y)`, but no exception is raised if *x* or *y* are NaN.

`int` **isless** (*real-floating x, real-floating y*) Macro
This macro determines whether the argument *x* is less than *y*. It is equivalent to `(x) < (y)`, but no exception is raised if *x* or *y* are NaN.

`int` **islessequal** (*real-floating x, real-floating y*) Macro
This macro determines whether the argument *x* is less than or equal to *y*. It is equivalent to `(x) <= (y)`, but no exception is raised if *x* or *y* are NaN.

`int` **islessgreater** (*real-floating x, real-floating y*) Macro
This macro determines whether the argument *x* is less than or greater than *y*. It is equivalent to `(x) < (y) || (x) > (y)` (although it only evaluates *x* and *y* once), but no exception is raised if *x* or *y* are NaN.

This macro is not equivalent to *x* `!=` *y*, because that expression is true if *x* or *y* are NaN.

`int` **isunordered** (*real-floating x, real-floating y*) Macro
This macro determines whether its arguments are unordered. In other words, it is true if *x* or *y* are NaN and false otherwise.

Not all machines provide hardware support for these operations. On machines that do not, the macros can be very slow. Therefore, you should not use these functions when NaN is not a concern.

There are no macros `isequal` or `isunequal`. They are unnecessary, because the == and `!=` operators do *not* throw an exception if one or both of the operands are NaN.

9.8.7 Miscellaneous FP Arithmetic Functions

The functions in this section perform miscellaneous but common operations that are awkward to express with C operators. On some processors, these functions can use special machine instructions to perform these operations faster than the equivalent C code.

`double` **fmin** (`double` *x*, `double` *y*) Function
`float` **fminf** (`float` *x*, `float` *y*) Function
`long double` **fminl** (`long double` *x*, `long double` *y*) Function
The `fmin` function returns the lesser of the two values *x* and *y*. It is similar to the expression:

```
((x) < (y) ? (x) : (y))
```

except that *x* and *y* are only evaluated once.

If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

`double` **fmax** (`double` *x*, `double` *y*) Function
`float` **fmaxf** (`float` *x*, `float` *y*) Function
`long double` **fmaxl** (`long double` *x*, `long double` *y*) Function

The `fmax` function returns the greater of the two values *x* and *y*.

If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

`double` **fdim** (`double` *x*, `double` *y*) Function
`float` **fdimf** (`float` *x*, `float` *y*) Function
`long double` **fdiml** (`long double` *x*, `long double` *y*) Function

The `fdim` function returns the positive difference between *x* and *y*. The positive difference is $x - y$ if *x* is greater than *y*, and 0 otherwise.

If *x*, *y*, or both are NaN, NaN is returned.

`double` **fma** (`double` *x*, `double` *y*, `double` *z*) Function
`float` **fmaf** (`float` *x*, `float` *y*, `float` *z*) Function
`long double` **fmal** (`long double` *x*, `long double` *y*, `long double` *z*) Function

The `fma` function performs floating-point multiply-add. This is the operation $(x \cdot y) + z$, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

This function was introduced because some processors have a special instruction to perform multiply-add. The C compiler cannot use it directly, because the expression '`x*y + z`' is defined to round the intermediate result. `fma` lets you choose when you want to round only once.

On processors that do not implement multiply-add in hardware, `fma` can be very slow since it must avoid intermediate rounding. '`math.h`' defines the symbols `FP_FAST_FMA`, `FP_FAST_FMAF`, and `FP_FAST_FMAL` when the corresponding version of `fma` is no slower than the expression '`x*y + z`'. In the GNU C Library, this always means the operation is implemented in hardware.

9.9 Complex Numbers

ISO C99 introduces support for complex numbers in C. This is done with a new type qualifier, `complex`. It is a keyword if and only if '`complex.h`' has been included. There are three complex types, corresponding to the three real types: `float complex`, `double complex` and `long double complex`.

To construct complex numbers, you need a way to indicate the imaginary part of a number. There is no standard notation for an imaginary floating-point constant.

Instead, ‘complex.h’ defines two macros that can be used to create complex numbers.

`const float complex` **_Complex_I** Macro

This macro is a representation of the complex number "$0 + 1i$". Multiplying a real floating-point value by `_Complex_I` gives a complex number whose value is purely imaginary. You can use this to construct complex constants:

```
3.0 + 4.0i = 3.0 + 4.0 * _Complex_I
```

Note that `_Complex_I * _Complex_I` has the value -1, but the type of that value is `complex`.

`_Complex_I` is a bit of a mouthful. ‘complex.h’ also defines a shorter name for the same constant.

`const float complex` **I** Macro

This macro has exactly the same value as `_Complex_I`. Most of the time it is preferable. However, it causes problems if you want to use the identifier `I` for something else. You can safely write:

```
#include <complex.h>
#undef I
```

if you need `I` for your own purposes. In that case, we recommend you also define some other short name for `_Complex_I`, such as `J`.

9.10 Projections, Conjugates and Decomposing of Complex Numbers

ISO C99 also defines functions that perform basic operations on complex numbers, such as decomposition and conjugation. The prototypes for all these functions are in ‘complex.h’. All functions are available in three variants, one for each of the three complex types.

`double` **creal** (`complex double` *z*) Function
`float` **crealf** (`complex float` *z*) Function
`long double` **creall** (`complex long double` *z*) Function

These functions return the real part of the complex number *z*.

`double` **cimag** (`complex double` *z*) Function
`float` **cimagf** (`complex float` *z*) Function
`long double` **cimagl** (`complex long double` *z*) Function

These functions return the imaginary part of the complex number *z*.

`complex double` **conj** (`complex double` *z*) Function
`complex float` **conjf** (`complex float` *z*) Function
`complex long double` **conjl** (`complex long double` *z*) Function
These functions return the conjugate value of the complex number *z*. The conjugate of a complex number has the same real part and a negated imaginary part. In other words, '`conj(a + bi) = a + -bi`'.

`double` **carg** (`complex double` *z*) Function
`float` **cargf** (`complex float` *z*) Function
`long double` **cargl** (`complex long double` *z*) Function
These functions return the argument of the complex number *z*. The argument of a complex number is the angle in the complex plane between the positive real axis and a line passing through zero and the number. This angle is measured in the usual fashion and ranges from 0 to 2π.
`carg` has a branch cut along the positive real axis.

`complex double` **cproj** (`complex double` *z*) Function
`complex float` **cprojf** (`complex float` *z*) Function
`complex long double` **cprojl** (`complex long double` *z*) Function
These functions return the projection of the complex value *z* onto the Riemann sphere. Values with an infinite imaginary part are projected to positive infinity on the real axis, even if the real part is NaN. If the real part is infinite, the result is equivalent to:

```
INFINITY + I * copysign (0.0, cimag (z))
```

9.11 Parsing of Numbers

This section describes functions for "reading" integer and floating-point numbers from a string. It may be more convenient in some cases to use `sscanf` or one of the related functions (see Section 17.14 [Formatted Input], page 486). But often you can make a program more robust by finding the tokens in the string by hand, then converting the numbers one by one.

9.11.1 Parsing of Integers

The '`str`' functions are declared in '`stdlib.h`', and those beginning with '`wcs`' are declared in '`wchar.h`'. You might wonder about the use of `restrict` in the prototypes of the functions in this section. It is seemingly useless, but the ISO C standard uses it (for the functions defined there), so we have to use it as well.

`long int` **strtol** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function
The `strtol` ("string-to-long") function converts the initial part of *string* to a signed integer, which is returned as a value of type `long int`.
This function attempts to decompose *string* as follows:

- A (possibly empty) sequence of white-space characters—which characters are white space is determined by the `isspace` function (see Section 4.1 [Classification of Characters], page 79). These are discarded.
- An optional plus or minus sign ('+' or '−')
- A nonempty sequence of digits in the radix specified by *base*—if *base* is zero, decimal radix is assumed unless the series of digits begins with '`0`' (specifying octal radix), or '`0x`' or '`0X`' (specifying hexadecimal radix); in other words, the same syntax used for integer constants in C.

  Otherwise *base* must have a value between `2` and `36`. If *base* is `16`, the digits may optionally be preceded by '`0x`' or '`0X`'. If base has no legal value, the value returned is `0l`, and the global variable `errno` is set to `EINVAL`.
- Any remaining characters in the string—if *tailptr* is not a null pointer, `strtol` stores a pointer to this tail in ⋆*tailptr*.

If the string is empty, contains only white space, or does not contain an initial substring that has the expected syntax for an integer in the specified *base*, no conversion is performed. In this case, `strtol` returns a value of zero and the value stored in ⋆*tailptr* is the value of *string*.

In a locale other than the standard `"C"` locale, this function may recognize additional implementation-dependent syntax.

If the string has valid syntax for an integer but the value is not representable because of overflow, `strtol` returns either `LONG_MAX` or `LONG_MIN`, as appropriate for the sign of the value.[9] It also sets `errno` to `ERANGE` to indicate there was overflow.

You should not check for errors by examining the return value of `strtol`, because the string might be a valid representation of `0l`, `LONG_MAX`, or `LONG_MIN`. Instead, check whether *tailptr* points to what you expect after the number (e.g., `'\0'` if the string should end after the number). You also need to clear *errno* before the call and check it afterward, in case there was overflow.

There is an example at the end of this section.

`long int` **wcstol** (`const wchar_t *restrict` *string*`,` `wchar_t **restrict` *tailptr*`,` `int` *base*) Function

The `wcstol` function is equivalent to the `strtol` function in nearly all aspects, but it handles wide-character strings.

The `wcstol` function was introduced in Amendment 1 of ISO C90.

`unsigned long int` **strtoul** (`const char *restrict` *string*`,` `char **restrict` *tailptr*`,` `int` *base*) Function

The `strtoul` ("string-to-unsigned-long") function is like `strtol` except it converts to an `unsigned long int` value. The syntax is the same as described above for `strtol`. The value returned on overflow is `ULONG_MAX`.[10]

[9] Ibid., "Range of an Integer Type".

[10] Ibid.

If *string* depicts a negative number, `strtoul` acts the same as *strtol* but casts the result to an unsigned integer. That means, for example, that `strtoul` on "-1" returns `ULONG_MAX`, and an input more negative than `LONG_MIN` returns (`ULONG_MAX` + 1) / 2.

`strtoul` sets *errno* to `EINVAL` if *base* is out of range, or to `ERANGE` on overflow.

`unsigned long int` **wcstoul** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstoul` function is equivalent to the `strtoul` function in nearly all aspects, but it handles wide-character strings.

The `wcstoul` function was introduced in Amendment 1 of ISO C90.

`long long int` **strtoll** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function

The `strtoll` function is like `strtol` except that it returns a `long long int` value and accepts numbers with a correspondingly larger range.

If the string has valid syntax for an integer but the value is not representable because of overflow, `strtoll` returns either `LONG_LONG_MAX` or `LONG_LONG_MIN`, as appropriate for the sign of the value.[11] It also sets `errno` to `ERANGE` to indicate there was overflow.

The `strtoll` function was introduced in ISO C99.

`long long int` **wcstoll** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstoll` function is equivalent to the `strtoll` function in nearly all aspects, but it handles wide-character strings.

The `wcstoll` function was introduced in Amendment 1 of ISO C90.

`long long int` **strtoq** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function

`strtoq` ("string-to-quad-word") is the BSD name for `strtoll`.

`long long int` **wcstoq** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstoq` function is equivalent to the `strtoq` function in nearly all aspects, but it handles wide-character strings.

The `wcstoq` function is a GNU extension.

`unsigned long long int` **strtoull** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function

The `strtoull` function is related to `strtoll` the same way `strtoul` is related to `strtol`.

The `strtoull` function was introduced in ISO C99.

[11] Ibid.

`unsigned long long int` **wcstoull** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstoull` function is equivalent to the `strtoull` function in nearly all aspects, but it handles wide-character strings.

The `wcstoull` function was introduced in Amendment 1 of ISO C90.

`unsigned long long int` **strtouq** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function

`strtouq` is the BSD name for `strtoull`.

`unsigned long long int` **wcstouq** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstouq` function is equivalent to the `strtouq` function in nearly all aspects, but it handles wide-character strings.

The `wcstoq` function is a GNU extension.

`intmax_t` **strtoimax** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function

The `strtoimax` function is like `strtol` except that it returns an `intmax_t` value and accepts numbers of a corresponding range.

If the string has valid syntax for an integer but the value is not representable because of overflow, `strtoimax` returns either `INTMAX_MAX` or `INTMAX_MIN` (see Section 9.1 [Integers], page 243), as appropriate for the sign of the value. It also sets `errno` to `ERANGE` to indicate there was overflow.

See Section 9.1 [Integers], page 243 for a description of the `intmax_t` type.
The `strtoimax` function was introduced in ISO C99.

`intmax_t` **wcstoimax** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstoimax` function is equivalent to the `strtoimax` function in nearly all aspects, but it handles wide-character strings.

The `wcstoimax` function was introduced in ISO C99.

`uintmax_t` **strtoumax** (`const char *restrict` *string*`, char **restrict` *tailptr*`, int` *base*) Function

The `strtoumax` function is related to `strtoimax` the same way that `strtoul` is related to `strtol`.

See Section 9.1 [Integers], page 243 for a description of the `intmax_t` type.
The `strtoumax` function was introduced in ISO C99.

`uintmax_t` **wcstoumax** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*) Function

The `wcstoumax` function is equivalent to the `strtoumax` function in nearly all aspects, but it handles wide-character strings.

The `wcstoumax` function was introduced in ISO C99.

`long int` **atol** (`const char *`*string*) Function

This function is similar to the `strtol` function with a *base* argument of `10`, except that it need not detect overflow errors. The `atol` function is provided mostly for compatibility with existing code; using `strtol` is more robust.

`int` **atoi** (`const char *`*string*) Function

This function is like `atol` except that it returns an `int`. The `atoi` function is also considered obsolete; use `strtol` instead.

`long long int` **atoll** (`const char *`*string*) Function

This function is similar to `atol` except that it returns a `long long int`.

The `atoll` function was introduced in ISO C99. It too is obsolete (despite having just been added); use `strtoll` instead.

All the functions mentioned in this section so far do not handle alternative representations of characters as described in the locale data. Some locales specify the thousands-separator and the way they have to be used, which can help to make large numbers more readable. To read such numbers, one has to use the `scanf` functions with the ‘’’ flag.

Here is a function that parses a string as a sequence of integers and returns the sum of them:

```
int
sum_ints_from_string (char *string)
{
  int sum = 0;

  while (1) {
    char *tail;
    int next;

    /* Skip white space by hand, to detect the end.  */
    while (isspace (*string)) string++;
    if (*string == 0)
      break;

    /* There is more non-white-space,  */
    /* so it ought to be another number.  */
    errno = 0;
    /* Parse it.  */
    next = strtol (string, &tail, 0);
    /* Add it in, if not overflow.  */
    if (errno)
      printf ("Overflow\n");
    else
      sum += next;
```

```
      /* Advance past it.  */
      string = tail;
    }

  return sum;
}
```

9.11.2 Parsing of Floats

The 'str' functions are declared in 'stdlib.h', and those beginning with 'wcs' are declared in 'wchar.h'. One might wonder about the use of `restrict` in the prototypes of the functions in this section. It is seemingly useless, but the ISO C standard uses it (for the functions defined there), so we have to use it as well.

`double` **strtod** (`const char *restrict` *string*`, char **restrict` *tailptr*) Function

The `strtod` ("string-to-double") function converts the initial part of *string* to a floating-point number, which is returned as a value of type `double`.

This function attempts to decompose *string* as follows:

- A (possibly empty) sequence of white-space characters—which characters are white space is determined by the `isspace` function (see Section 4.1 [Classification of Characters], page 79). These are discarded.
- An optional plus or minus sign ('+' or '−')
- A floating-point number in decimal or hexadecimal format; the decimal format is:
  - A nonempty sequence of digits optionally containing a decimal-point character—normally '.', but it depends on the locale (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187).
  - An optional exponent part, consisting of a character 'e' or 'E', an optional sign and a sequence of digits

  The hexadecimal format is as follows:
  - A 0x or 0X followed by a nonempty sequence of hexadecimal digits optionally containing a decimal-point character—normally '.', but it depends on the locale (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187)
  - An optional binary-exponent part, consisting of a character 'p' or 'P', an optional sign, and a sequence of digits
- Any remaining characters in the string. If *tailptr* is not a null pointer, a pointer to this tail of the string is stored in **tailptr*.

If the string is empty, contains only white space, or does not contain an initial substring that has the expected syntax for a floating-point number, no conversion is performed. In this case, `strtod` returns a value of zero and the value returned in **tailptr* is the value of *string*.

In a locale other than the standard "C" or "POSIX" locales, this function may recognize additional locale-dependent syntax.

If the string has valid syntax for a floating-point number but the value is outside the range of a double, strtod will signal overflow or underflow as described in Section 9.5.4 [Error Reporting by Mathematical Functions], page 253.

strtod recognizes four special input strings. The strings "inf" and "infinity" are converted to ∞, or to the largest representable value if the floating-point format doesn't support infinities. You can prepend a "+" or "-" to specify the sign. Case is ignored when scanning these strings.

The strings "nan" and "nan(*chars...*)" are converted to NaN. Again, case is ignored. If *chars...* are provided, they are used in some unspecified fashion to select a particular representation of NaN (there can be several).

Since zero is a valid result as well as the value returned on error, you should check for errors in the same way as for strtol, by examining *errno* and *tailptr*.

`float` **strtof** (`const char *`*string*`, char **`*tailptr*) Function
`long double` **strtold** (`const char *`*string*`, char **`*tailptr*) Function

These functions are analogous to strtod, but they return float and long double values respectively. They report errors in the same way as strtod. strtof can be substantially faster than strtod, but has less precision; conversely, strtold can be much slower, but has more precision (on systems where long double is a separate type).

These functions have been GNU extensions and are new to ISO C99.

`double` **wcstod** (`const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*) Function
`float` **wcstof** (`const wchar_t *`*string*`, wchar_t **`*tailptr*) Function
`long double` **wcstold** (`const wchar_t *`*string*`, wchar_t **`*tailptr*) Function

The wcstod, wcstof and wcstol functions are equivalent in nearly all aspect to the strtod, strtof and strtold functions, but they handle wide-character strings.

The wcstod function was introduced in Amendment 1 of ISO C90. The wcstof and wcstold functions were introduced in ISO C99.

`double` **atof** (`const char *`*string*) Function

This function is similar to the strtod function, except that it need not detect overflow and underflow errors. The atof function is provided mostly for compatibility with existing code; using strtod is more robust.

The GNU C Library also provides '_l' versions of these functions, which take an additional argument, the locale to use in conversion (see Section 9.11.1 [Parsing of Integers], page 268).

9.12 Old-fashioned System V Number-to-String Functions

The old System V C library provided three functions to convert numbers to strings, with unusual and hard-to-use semantics. The GNU C library also provides these functions and some natural extensions.

These functions are only available in glibc and on systems descended from AT&T Unix. Therefore, unless these functions do precisely what you need, it is better to use `sprintf`, which is standard.

All these functions are defined in '`stdlib.h`'.

`char *` **ecvt** (`double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*) Function

The function `ecvt` converts the floating-point number *value* to a string with at most *ndigit* decimal digits. The returned string contains no decimal point or sign. The first digit of the string is nonzero (unless *value* is actually zero) and the last digit is rounded to nearest. **decpt* is set to the index in the string of the first digit after the decimal point. **neg* is set to a nonzero value if *value* is negative and zero otherwise.

If *ndigit* decimal digits would exceed the precision of a `double`, it is reduced to a system-specific value.

The returned string is statically allocated and overwritten by each call to `ecvt`.

If *value* is zero, whether **decpt* is `0` or `1` is implementation defined.

For example: `ecvt (12.3, 5, &d, &n)` returns `"12300"` and sets *d* to `2` and *n* to `0`.

`char *` **fcvt** (`double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*) Function

The function `fcvt` is like `ecvt`, but *ndigit* specifies the number of digits after the decimal point. If *ndigit* is less than zero, *value* is rounded to the *ndigit* + 1'th place to the left of the decimal point. For example, if *ndigit* is `-1`, *value* will be rounded to the nearest 10. If *ndigit* is negative and larger than the number of digits to the left of the decimal point in *value*, *value* will be rounded to one significant digit.

If *ndigit* decimal digits would exceed the precision of a `double`, it is reduced to a system-specific value.

The returned string is statically allocated and overwritten by each call to `fcvt`.

`char *` **gcvt** (`double` *value*, `int` *ndigit*, `char *`*buf*) Function

`gcvt` is functionally equivalent to '`sprintf(buf, "%*g", ndigit, value)`'. It is provided only for compatibility's sake. It returns *buf*.

If *ndigit* decimal digits would exceed the precision of a `double`, it is reduced to a system-specific value.

As extensions, the GNU C Library provides versions of these three functions that take `long double` arguments.

`char *` **qecvt** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*) Function

This function is equivalent to `ecvt`, except that it takes a `long double` for the first parameter and that *ndigit* is restricted by the precision of a `long double`.

`char *` **qfcvt** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*) Function

This function is equivalent to `fcvt`, except that it takes a `long double` for the first parameter, and that *ndigit* is restricted by the precision of a `long double`.

`char *` **qgcvt** (`long double` *value*, `int` *ndigit*, `char *`*buf*) Function

This function is equivalent to `gcvt`, except that it takes a `long double` for the first parameter, and that *ndigit* is restricted by the precision of a `long double`.

The `ecvt` and `fcvt` functions, and their `long double` equivalents, all return a string located in a static buffer, which is overwritten by the next call to the function. The GNU C Library provides another set of extended functions that write the converted string into a user-supplied buffer. These have the conventional `_r` suffix.

`gcvt_r` is not necessary, because `gcvt` already uses a user-supplied buffer.

`char *` **ecvt_r** (`double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*, `char *`*buf*, `size_t` *len*) Function

The `ecvt_r` function is the same as `ecvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

`char *` **fcvt_r** (`double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*, `char *`*buf*, `size_t` *len*) Function

The `fcvt_r` function is the same as `fcvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

`char *` **qecvt_r** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*, `char *`*buf*, `size_t` *len*) Function

The `qecvt_r` function is the same as `qecvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

`char *` **qfcvt_r** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*, `char *`*buf*, `size_t` *len*) Function

The `qfcvt_r` function is the same as `qfcvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

10 Date and Time

This chapter describes functions for manipulating dates and times, including functions for determining what time it is and conversion between different time representations.

10.1 Time Basics

Discussing time in a technical manual can be difficult because the word *time* in English refers to lots of different things. In this manual, we use a rigorous terminology to avoid confusion, and the only thing we use the simple word *time* for is to talk about the abstract concept.

A *calendar time* is a point in the time continuum, for example November 4, 1990 at 18:02.5 UTC. Sometimes this is called *absolute time*.

We don't speak of a *date*, because that is inherent in a calendar time.

An *interval* is a contiguous part of the time continuum between two calendar times, such as the hour between 9:00 and 10:00 on July 4, 1980.

An *elapsed time* is the length of an interval, such as 35 minutes. People sometimes sloppily use the word *interval* to refer to the elapsed time of some interval.

An *amount of time* is a sum of elapsed times, which need not be of any specific intervals. For example, the amount of time it takes to read a book might be 9 hours, independently of when and in how many sittings it is read.

A *period* is the elapsed time of an interval between two events, especially when they are part of a sequence of regularly repeating events.

CPU *time* is like calendar time, except that it is based on the subset of the time continuum when a particular process is actively using a CPU. CPU time is therefore relative to a process.

Processor time is an amount of time that a CPU is in use. In fact, it's a basic system resource, since there is a limit to how much can exist in any given interval (that limit is the elapsed time of the interval multiplied by the number of CPUs in the processor). People often call this CPU time, but we reserve the latter term in this manual for the definition above.

10.2 Elapsed Time

One way to represent an elapsed time is with a simple arithmetic data type, as with the following function to compute the elapsed time between two calendar times. This function is declared in '`time.h`':

`double` **difftime** (`time_t` *time1*, `time_t` *time0*) Function

The `difftime` function returns the number of seconds of elapsed time between calendar time *time1* and calendar time *time0*, as a value of type `double`. The difference ignores leap seconds unless leap second support is enabled.

In the GNU system, you can simply subtract `time_t` values. But on other systems, the `time_t` data type might use some other encoding where subtraction doesn't work directly.

The GNU C Library provides two data types specifically for representing an elapsed time. They are used by various GNU C Library functions, and you can use them for your own purposes too. They're exactly the same, except that one has a resolution in microseconds, and the other, newer one, has a resolution in nanoseconds.

struct timeval Data Type

The `struct timeval` structure represents an elapsed time. It is declared in '`sys/time.h`' and has the following members:

`long int tv_sec`
: This represents the number of whole seconds of elapsed time.

`long int tv_usec`
: This is the rest of the elapsed time (a fraction of a second), represented as the number of microseconds. It is always less than one million.

struct timespec Data Type

The `struct timespec` structure represents an elapsed time. It is declared in '`time.h`' and has the following members:

`long int tv_sec`
: This represents the number of whole seconds of elapsed time.

`long int tv_nsec`
: This is the rest of the elapsed time (a fraction of a second), represented as the number of nanoseconds. It is always less than one billion.

It is often necessary to subtract two values of type `struct timeval` or `struct timespec`. Here is the best way to do this (it works even on some peculiar operating systems where the `tv_sec` member has an unsigned type):

```
/* Subtract the 'struct timeval' values X and Y,
   storing the result in RESULT.
   Return 1 if the difference is negative, otherwise 0.  */

int
timeval_subtract (result, x, y)
     struct timeval *result, *x, *y;
{
  /* Perform the carry for the later subtraction by updating y. */
  if (x->tv_usec < y->tv_usec) {
    int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
```

```
    y->tv_usec -= 1000000 * nsec;
    y->tv_sec += nsec;
  }
  if (x->tv_usec - y->tv_usec > 1000000) {
    int nsec = (x->tv_usec - y->tv_usec) / 1000000;
    y->tv_usec += 1000000 * nsec;
    y->tv_sec -= nsec;
  }

  /* Compute the time remaining to wait.
     tv_usec is certainly positive. */
  result->tv_sec = x->tv_sec - y->tv_sec;
  result->tv_usec = x->tv_usec - y->tv_usec;

  /* Return 1 if result is negative. */
  return x->tv_sec < y->tv_sec;
}
```

Common functions that use `struct timeval` are `gettimeofday` and `settimeofday`.

There are no GNU C Library functions specifically oriented toward dealing with elapsed times, but the calendar time, processor time and alarm and sleeping functions have a lot to do with them.

10.3 Processor and CPU Time

If you are trying to optimize your program or measure its efficiency, it's very useful to know how much processor time it uses. For that, calendar time and elapsed times are useless because a process may spend time waiting for I/O or for other processes to use the CPU. However, you can get the information with the functions in this section.

CPU time (see Section 10.1 [Time Basics], page 277) is represented by the data type `clock_t`, which is a number of *clock ticks*. It gives the total amount of time a process has actively used a CPU since some arbitrary event. On the GNU system, that event is the creation of the process. While arbitrary in general, the event is always the same event for any particular process, so you can always measure how much time on the CPU a particular computation takes by examinining the process' CPU time before and after the computation.

In the GNU system, `clock_t` is equivalent to `long int` and `CLOCKS_PER_SEC` is an integer value. But in other systems, both `clock_t` and the macro `CLOCKS_PER_SEC` can be either integer or floating-point types. Casting CPU time values to `double`, as in the example above, makes sure that operations such as arithmetic and printing work properly and consistently no matter what the underlying representation is.

Note that the clock can wrap around. On a 32-bit system with CLOCKS_PER_SEC set to one million, this function will return the same value approximately every 72 minutes.[1]

10.3.1 CPU Time Inquiry

To get a process' CPU time, you can use the clock function. This facility is declared in the header file 'time.h'.

In typical usage, you call the clock function at the beginning and end of the interval you want to time, subtract the values and then divide by CLOCKS_PER_SEC (the number of clock ticks per second) to get processor time, like this:

```
#include <time.h>

clock_t start, end;
double cpu_time_used;

start = clock();
... /* Do the work. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Do not use a single CPU time as an amount of time; it doesn't work that way. Either do a subtraction as shown above or query processor time directly (see Section 10.3.2 [Processor Time Inquiry], page 281).

Different computers and operating systems vary wildly in how they keep track of CPU time. It's common for the internal processor clock to have a resolution somewhere between a hundredth and millionth of a second.

`int` **CLOCKS_PER_SEC** Macro

The value of this macro is the number of clock ticks per second measured by the clock function. POSIX requires that this value be one million independent of the actual resolution.

`int` **CLK_TCK** Macro

This is an obsolete name for CLOCKS_PER_SEC.

clock_t Data Type

This is the type of the value returned by the clock function. Values of type clock_t are numbers of clock ticks.

[1] See Loosemore et al., "Resource Usage and Limitation", for additional functions to examine a process' use of processor time and to control it.

`clock_t` **clock** (`void`) Function

This function returns the calling process' current CPU time. If the CPU time is not available or cannot be represented, `clock` returns the value `(clock_t)(-1)`.

10.3.2 Processor Time Inquiry

The `times` function returns information about a process' consumption of processor time in a `struct tms` object, in addition to the process' CPU time (see Section 10.1 [Time Basics], page 277). You should include the header file '`sys/times.h`' to use this facility.

struct tms Data Type

The `tms` structure is used to return information about process times. It contains at least the following members:

`clock_t tms_utime`
: This is the total processor time the calling process has used in executing the instructions of its program.

`clock_t tms_stime`
: This is the processor time the system has used on behalf of the calling process.

`clock_t tms_cutime`
: This is the sum of the `tms_utime` values and the `tms_cutime` values of all terminated child processes of the calling process, whose status has been reported to the parent process by `wait` or `waitpid`.[2] In other words, it represents the total processor time used in executing the instructions of all the terminated child processes of the calling process, excluding child processes that have not yet been reported by `wait` or `waitpid`.

`clock_t tms_cstime`
: This is similar to `tms_cutime`, but represents the total processor time the system has used on behalf of all the terminated child processes of the calling process.

All of the times are given in numbers of clock ticks. Unlike CPU time, these are the actual amounts of time; they are not relative to any event.[3]

`clock_t` **times** (`struct tms *`*buffer*) Function

The `times` function stores the processor time information for the calling process in *buffer*.

The return value is the calling process' CPU time (the same value you get from `clock()`). `times` returns `(clock_t)(-1)` to indicate failure.

---

[2] Ibid., "Process Completion".

[3] Ibid., "Creating a Process".

Portability Note: The `clock` function described in Section 10.3.1 [CPU Time Inquiry], page 280, is specified by the ISO C standard. The `times` function is a feature of POSIX.1. In the GNU system, the CPU time is defined to be equivalent to the sum of the `tms_utime` and `tms_stime` fields returned by `times`.

10.4 Calendar Time

This section describes facilities for keeping track of calendar time (see Section 10.1 [Time Basics], page 277).

The GNU C Library represents calendar time three ways:

- *Simple time* (the `time_t` data type) is a compact representation, typically giving the number of seconds of elapsed time since some implementation-specific base time.
- There is also a *high-resolution time* representation. Like simple time, this represents a calendar time as an elapsed time since a base time, but instead of measuring in whole seconds, it uses a `struct timeval` data type, which includes fractions of a second. Use this time representation instead of simple time when you need greater precision.
- *Local time* or *broken-down time* (the `struct tm` data type) represents a calendar time as a set of components specifying the year, month, etc., in the Gregorian calendar for a specific time zone. This calendar time representation is usually used only to communicate with people.

10.4.1 Simple Calendar Time

This section describes the `time_t` data type for representing calendar time as simple time, and the functions that operate on simple time objects. These facilities are declared in the header file '`time.h`'.

time_t Data Type

This is the data type used to represent simple time. Sometimes, it also represents an elapsed time. When interpreted as a calendar time value, it represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (this calendar time is sometimes referred to as the *epoch*). POSIX requires that this count not include leap seconds, but on some systems this count includes leap seconds if you set `TZ` to certain values (see Section 10.4.7 [Specifying the Time Zone with `TZ`], page 306).

A simple time has no concept of local time zone. Calendar Time T is the same instant in time regardless of where on the globe the computer is.

In the GNU C Library, `time_t` is equivalent to `long int`. In other systems, `time_t` might be either an integer or floating-point type.

The function `difftime` tells you the elapsed time between two simple calendar times, which is not always as easy to compute as just subtracting (see Section 10.2 [Elapsed Time], page 277).

`time_t` **time** (`time_t *`*result*) Function

The `time` function returns the current calendar time as a value of type `time_t`. If the argument *result* is not a null pointer, the calendar time value is also stored in `*`*result*. If the current calendar time is not available, the value `(time_t)(-1)` is returned.

`int` **stime** (`time_t *`*newtime*) Function

`stime` sets the system clock, i.e., it tells the system that the current calendar time is *newtime*, where `newtime` is interpreted as described in the above definition of `time_t`.

`settimeofday` is a newer function that sets the system clock to better than one second precision. `settimeofday` is generally a better choice than `stime` (see Section 10.4.2 [High-Resolution Calendar], page 283).

Only the superuser can set the system clock.

If the function succeeds, the return value is zero. Otherwise, it is `-1` and `errno` is set accordingly:

`EPERM` The process is not superuser.

10.4.2 High-Resolution Calendar

The `time_t` data type used to represent simple times has a resolution of only 1 second. Some applications need more precision.

So, the GNU C Library also contains functions that are capable of representing calendar times to a higher resolution than 1 second. The functions and the associated data types described in this section are declared in '`sys/time.h`'.

struct timezone Data Type

The `struct timezone` structure is used to hold minimal information about the local time zone. It has the following members:

`int tz_minuteswest`
: This is the number of minutes west of UTC.

`int tz_dsttime`
: If nonzero, daylight saving time applies during some part of the year.

The `struct timezone` type is obsolete and should never be used. Instead, use the facilities described in Section 10.4.8 [Functions and Variables for Time Zones], page 308.

`int` **gettimeofday** (`struct timeval *`*tp*`, struct timezone *`*tzp*) Function

The `gettimeofday` function returns the current calendar time as the elapsed time since the epoch in the `struct timeval` structure indicated by *tp*. (see Section 10.2 [Elapsed Time], page 277 for a description of `struct`

`timeval`). Information about the time zone is returned in the structure pointed at *tzp*. If the *tzp* argument is a null pointer, time zone information is ignored.

The return value is `0` on success and `-1` on failure. The following `errno` error condition is defined for this function:

`ENOSYS` The operating system does not support getting time zone information, and *tzp* is not a null pointer. The GNU operating system does not support using `struct timezone` to represent time zone information; that is an obsolete feature of 4.3 BSD. Instead, use the facilities described in Section 10.4.8 [Functions and Variables for Time Zones], page 308.

`int` **settimeofday** (`const struct timeval *`*tp*`, const struct timezone *`*tzp*) Function

The `settimeofday` function sets the current calendar time in the system clock according to the arguments. As for `gettimeofday`, the calendar time is represented as the elapsed time since the epoch. As for `gettimeofday`, time zone information is ignored if *tzp* is a null pointer.

You must be a privileged user in order to use `settimeofday`.

Some kernels automatically set the system clock from some source such as a hardware clock when they start up. Others, including Linux, place the system clock in an *invalid* state (in which attempts to read the clock fail). A call of `stime` removes the system clock from an invalid state, and system start-up scripts typically run a program that calls `stime`.

`settimeofday` causes a sudden jump forward or backward, which can cause a variety of problems in a system. Use `adjtime` (below) to make a smooth transition from one time to another by temporarily speeding up or slowing down the clock.

With a Linux kernel, `adjtimex` does the same thing and can also make permanent changes to the speed of the system clock so that it doesn't need to be corrected as often.

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EPERM` This process cannot set the clock because it is not privileged.

`ENOSYS` The operating system does not support setting time zone information, and *tzp* is not a null pointer.

`int` **adjtime** (`const struct timeval *`*delta*`, struct timeval *`*olddelta*) Function

This function speeds up or slows down the system clock in order to make a gradual adjustment. This ensures that the calendar time reported by the system clock is always monotonically increasing, which might not happen if you simply set the clock.

The *delta* argument specifies a relative adjustment to be made to the clock time. If negative, the system clock is slowed down for a while until it has lost this much elapsed time. If positive, the system clock is speeded up for a while.

If the *olddelta* argument is not a null pointer, the `adjtime` function returns information about any previous time adjustment that has not yet been completed.

This function is typically used to synchronize the clocks of computers in a local network. You must be a privileged user to use it.

With a Linux kernel, you can use the `adjtimex` function to permanently change the clock speed.

The return value is `0` on success and `-1` on failure. The following `errno` error condition is defined for this function:

`EPERM` You do not have privilege to set the time.

Portability Note: The `gettimeofday`, `settimeofday` and `adjtime` functions are derived from BSD.

Symbols for the following function are declared in '`sys/timex.h`':

`int` **adjtimex** (`struct timex` ⋆*timex*) Function

`adjtimex` is functionally identical to `ntp_adjtime` (see Section 10.4.4 [High-Accuracy Clock], page 288).

This function is present only with a Linux kernel.

10.4.3 Broken-Down Time

Calendar time is represented by the usual GNU C Library functions as an elapsed time since a fixed base calendar time. This is convenient for computation, but has no relation to the way people normally think of calendar time. By contrast, *broken-down time* is a binary representation of calendar time separated into year, month, day and so on. Broken-down time values are not useful for calculations, but they are useful for printing human-readable time information.

A broken-down time value is always relative to a choice of time zone, and it also indicates which time zone that is.

The symbols in this section are declared in the header file '`time.h`'.

struct tm Data Type

This is the data type used to represent a broken-down time. The structure contains at least the following members, which can appear in any order:

`int tm_sec`

This is the number of full seconds since the top of the minute (normally in the range `0` through `59`, but the actual upper limit is `60`, to allow for leap seconds if leap second support is available).

`int tm_min`

This is the number of full minutes since the top of the hour (in the range `0` through `59`).

`int tm_hour`
: This is the number of full hours past midnight (in the range `0` through `23`).

`int tm_mday`
: This is the ordinal day of the month (in the range `1` through `31`). Watch out for this one! As the only ordinal number in the structure, it is inconsistent with the rest of the structure.

`int tm_mon`
: This is the number of full calendar months since the beginning of the year (in the range `0` through `11`). Watch out for this one! People usually use ordinal numbers for month-of-year (where January = 1).

`int tm_year`
: This is the number of full calendar years since 1900.

`int tm_wday`
: This is the number of full days since Sunday (in the range `0` through `6`).

`int tm_yday`
: This is the number of full days since the beginning of the year (in the range `0` through `365`).

`int tm_isdst`
: This is a flag that indicates whether daylight saving time is (or was, or will be) in effect at the time described. The value is positive if daylight saving time is in effect, zero if it is not and negative if the information is not available.

`long int tm_gmtoff`
: This field describes the time zone that was used to compute this broken-down time value, including any adjustment for daylight saving; it is the number of seconds that you must add to UTC to get local time. You can also think of this as the number of seconds east of UTC. For example, for US eastern standard time, the value is `-5*60*60`. The `tm_gmtoff` field is derived from BSD and is a GNU library extension; it is not visible in a strict ISO C environment.

`const char *tm_zone`
: This field is the name for the time zone that was used to compute this broken-down time value. Like `tm_gmtoff`, this field is a BSD and GNU extension, and is not visible in a strict ISO C environment.

`struct tm *` **localtime** (`const time_t *`*time*) Function

The `localtime` function converts the simple time pointed to by *time* to broken-down time representation, expressed relative to the user's specified time zone.

The return value is a pointer to a static broken-down time structure, which might be overwritten by subsequent calls to `ctime`, `gmtime`, or `localtime` (but no other library function overwrites the contents of this object).

The return value is the null pointer if *time* cannot be represented as a broken-down time; typically this is because the year cannot fit into an `int`.

Calling `localtime` has one other effect: it sets the variable `tzname` with information about the current time zone (see Section 10.4.8 [Functions and Variables for Time Zones], page 308).

Using the `localtime` function is a big problem in multithreaded programs. The result is returned in a static buffer and this is used in all threads. POSIX.1c introduced a variant of this function.

`struct tm *` **localtime_r** (`const time_t *`*time*`, struct tm *`*resultp*) Function

The `localtime_r` function works just like the `localtime` function. It takes a pointer to a variable containing a simple time and converts it to the broken-down time format.

But the result is not placed in a static buffer. Instead it is placed in the object of type `struct tm` to which the parameter *resultp* points.

If the conversion is successful the function returns a pointer to the object the result was written into, i.e., it returns *resultp*.

`struct tm *` **gmtime** (`const time_t *`*time*) Function

This function is similar to `localtime`, except that the broken-down time is expressed as Coordinated Universal Time (UTC) (formerly called Greenwich mean time (GMT)) rather than relative to a local time zone.

As for the `localtime` function, we have the problem that the result is placed in a static variable. POSIX.1c also provides a replacement for `gmtime`.

`struct tm *` **gmtime_r** (`const time_t *`*time*`, struct tm *`*resultp*) Function

This function is similar to `localtime_r`, except that it converts, just like `gmtime`, the given time as Coordinated Universal Time.

If the conversion is successful, the function returns a pointer to the object the result was written into, i.e., it returns *resultp*.

`time_t` **mktime** (`struct tm *`*brokentime*) Function

The `mktime` function is used to convert a broken-down time structure to a simple time representation. It also *normalizes* the contents of the broken-down time structure, by filling in the day of week and day of year based on the other date and time components.

The `mktime` function ignores the specified contents of the `tm_wday` and `tm_yday` members of the broken-down time structure. It uses the values of the

other components to determine the calendar time; it's permissible for these components to have unnormalized values outside their normal ranges. The last thing that `mktime` does is adjust the components of the *brokentime* structure (including the `tm_wday` and `tm_yday`).

If the specified broken-down time cannot be represented as a simple time, `mktime` returns a value of `(time_t)(-1)` and does not modify the contents of *brokentime*.

Calling `mktime` also sets the variable `tzname` with information about the current time zone (see Section 10.4.8 [Functions and Variables for Time Zones], page 308).

`time_t` **timelocal** (`struct tm *`*brokentime*) Function

`timelocal` is functionally identical to `mktime`, but more mnemonically named. It is the inverse of the `localtime` function.

Portability Note: `mktime` is essentially universally available. `timelocal` is rather rare.

`time_t` **timegm** (`struct tm *`*brokentime*) Function

`timegm` is functionally identical to `mktime`, except that it always takes the input values to be Coordinated Universal Time (UTC) regardless of any local time zone setting.

`timegm` is the inverse of `gmtime`.

Portability Note: `mktime` is essentially universally available. `timegm` is rather rare. For the most portable conversion from a UTC broken-down time to a simple time, set the `TZ` environment variable to UTC, call `mktime`, then set `TZ` back.

10.4.4 High-Accuracy Clock

The `ntp_gettime` and `ntp_adjtime` functions provide an interface to monitor and manipulate the system clock to maintain high-accuracy time. For example, you can fine-tune the speed of the clock or synchronize it with another time source.

A typical use of these functions is by a server implementing the Network Time Protocol to synchronize the clocks of multiple systems and high-precision clocks.

These functions are declared in '`sys/timex.h`'.

struct ntptimeval Data Type

This structure is used for information about the system clock. It contains the following members:

`struct timeval time`
: This is the current calendar time, expressed as the elapsed time since the epoch. The `struct timeval` data type is described in Section 10.2 [Elapsed Time], page 277.

`long int maxerror`
: This is the maximum error, measured in microseconds. Unless updated via `ntp_adjtime` periodically, this value will reach some platform-specific maximum value.

`long int esterror`
: This is the estimated error, measured in microseconds. This value can be set by `ntp_adjtime` to indicate the estimated offset of the system clock from the true calendar time.

`int` **ntp_gettime** (`struct ntptimeval *`*tptr*) Function

The `ntp_gettime` function sets the structure pointed to by *tptr* to current values. The elements of the structure afterwards contain the values the timer implementation in the kernel assumes. They might or might not be correct. If they are not, an `ntp_adjtime` call is necessary.

The return value is `0` on success and other values on failure. The following `errno` error conditions are defined for this function:

`TIME_ERROR`
: The precision clock model is not properly set up at the moment, thus the clock must be considered unsynchronized, and the values should be treated with care.

struct timex Data Type

This structure is used to control and monitor the system clock. It contains the following members:

`unsigned int modes`
: This variable controls whether and which values are set. Several symbolic constants have to be combined with *binary or* to specify the effective mode. These constants start with `MOD_`.

`long int offset`
: This value indicates the current offset of the system clock from the true calendar time. The value is given in microseconds. If bit `MOD_OFFSET` is set in `modes`, the offset (and possibly other dependent values) can be set. The offset's absolute value must not exceed `MAXPHASE`.

`long int frequency`
: This value indicates the difference in frequency between the true calendar time and the system clock. The value is expressed as scaled PPM (parts per million, 0.0001%). The scaling is `1 << SHIFT_USEC`. The value can be set with bit `MOD_FREQUENCY`, but the absolute value must not exceed `MAXFREQ`.

`long int maxerror`
: This is the maximum error, measured in microseconds. A new value can be set using bit `MOD_MAXERROR`. Unless updated via

`ntp_adjtime` periodically, this value will increase steadily and reach some platform-specific maximum value.

`long int esterror`
: This is the estimated error, measured in microseconds. This value can be set using bit `MOD_ESTERROR`.

`int status`
: This variable reflects the various states of the clock machinery. There are symbolic constants for the significant bits, starting with `STA_`. Some of these flags can be updated using the `MOD_STATUS` bit.

`long int constant`
: This value represents the bandwidth or stiffness of the PLL (phase locked loop) implemented in the kernel. The value can be changed using bit `MOD_TIMECONST`.

`long int precision`
: This value represents the accuracy or the maximum error when reading the system clock. The value is expressed in microseconds.

`long int tolerance`
: This value represents the maximum frequency error of the system clock in scaled PPM. This value is used to increase the `maxerror` every second.

`struct timeval time`
: This is the current calendar time.

`long int tick`
: This is the elapsed time between clock ticks in microseconds. A clock tick is a periodic timer interrupt on which the system clock is based.

`long int ppsfreq`
: This is the first of a few optional variables that are present only if the system clock can use a PPS (pulse per second) signal to discipline the system clock. The value is expressed in scaled PPM and it denotes the difference in frequency between the system clock and the PPS signal.

`long int jitter`
: This value expresses a median filtered average of the PPS signal's dispersion in microseconds.

`int shift`
: This value is a binary exponent for the duration of the PPS calibration interval, ranging from `PPS_SHIFT` to `PPS_SHIFTMAX`.

`long int stabil`
: This value represents the median filtered dispersion of the PPS frequency in scaled PPM.

`long int jitcnt`
: This counter represents the number of pulses where the jitter exceeded the allowed maximum `MAXTIME`.

`long int calcnt`
: This counter reflects the number of successful calibration intervals.

`long int errcnt`
: This counter represents the number of calibration errors (caused by large offsets or jitter).

`long int stbcnt`
: This counter denotes the number of calibrations where the stability exceeded the threshold.

`int` **ntp_adjtime** (`struct timex *`*tptr*) Function

The `ntp_adjtime` function sets the structure specified by *tptr* to current values.

In addition, `ntp_adjtime` updates some settings to match what you pass to it in **tptr*. Use the `modes` element of **tptr* to select what settings to update. You can set `offset`, `freq`, `maxerror`, `esterror`, `status`, `constant` and `tick`.

`modes` = zero means set nothing.

Only the superuser can update settings.

The return value is `0` on success and other values on failure. The following `errno` error conditions are defined for this function:

`TIME_ERROR`
: The high-accuracy clock model is not properly set up at the moment, thus the clock must be considered unsynchronized, and the values should be treated with care. Another reason could be that the specified new values are not allowed.

`EPERM` The process specified a settings update but is not superuser.

For more details see RFC1305 (Network Time Protocol, Version 3) and related documents.

Portability Note: Early versions of the GNU C Library did not have this function but did have the synonymous `adjtimex`.

10.4.5 Formatting Calendar Time

The functions described in this section format calendar time values as strings. These functions are declared in the header file '`time.h`'.

`char *` **asctime** (`const struct tm *`*brokentime*) Function

The `asctime` function converts the broken-down time value that *brokentime* points to into a string in a standard format:

```
"Tue May 21 13:46:22 1991\n"
```

The abbreviations for the days of week are: 'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri' and 'Sat'.

The abbreviations for the months are: 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov' and 'Dec'.

The return value points to a statically allocated string, which might be overwritten by subsequent calls to `asctime` or `ctime` (but no other library function overwrites the contents of this string).

`char *` **asctime_r** (`const struct tm *`*brokentime*`, char *`*buffer*) Function

This function is similar to `asctime`, but instead of placing the result in a static buffer it writes the string in the buffer pointed to by the parameter *buffer*. This buffer should have room for at least 26 bytes, including the terminating null.

If no error occurred, the function returns a pointer to the string the result was written into, i.e., it returns *buffer*. Otherwise, it returns `NULL`.

`char *` **ctime** (`const time_t *`*time*) Function

The `ctime` function is similar to `asctime`, except that you specify the calendar time argument as a `time_t` simple time value rather than in broken-down local time format. It is equivalent to:

```
asctime (localtime (time))
```

`ctime` sets the variable `tzname`, because `localtime` does so (see Section 10.4.8 [Functions and Variables for Time Zones], page 308).

`char *` **ctime_r** (`const time_t *`*time*`, char *`*buffer*) Function

This function is similar to `ctime` but places the result in the string pointed to by *buffer*. It is equivalent to:[4]

```
({ struct tm tm; asctime_r (localtime_r (time, &tm), buf); })
```

If no error occurred the function returns a pointer to the string the result was written into, i.e., it returns *buffer*. Otherwise, it returns a `NULL`.

`size_t` **strftime** (`char *`*s*`, size_t` *size*`, const char *`*template*`, const struct tm *`*brokentime*) Function

This function is similar to the `sprintf` function (see Section 17.14 [Formatted Input], page 486), but the conversion specifications that can appear in the format template *template* are specialized for printing components of the date and time *brokentime* according to the locale currently specified for time conversion (see Chapter 7 [Locales and Internationalization], page 181).

Ordinary characters appearing in the *template* are copied to the output string *s*; this can include multibyte-character sequences. Conversion specifiers are

[4] This was written using GCC extensions. See Richard M. Stallman, "Statements and Declarations in Expressions" in *Using and Porting GCC* (Boston, MA: GNU Press, July 1999), http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.htm.

introduced by a '%' character, followed by an optional flag, which can be one of the following. These flags are all GNU extensions. The first three affect only the output of numbers:

_ The number is padded with spaces.

- The number is not padded at all.

0 The number is padded with zeros even if the format specifies padding with spaces.

^ The output uses uppercase characters, but only if this is possible (see Section 4.2 [Case Conversion], page 81).

The default action is to pad the number with zeros to keep it a constant width. Numbers that do not have a range indicated below are never padded, since there is no natural width for them.

Following the flag, an optional specification of the width is possible. This is specified in decimal notation. If the natural size of the output of the field has less than the specified number of characters, the result is written right-adjusted and space padded to the given size.

An optional modifier can follow the optional flag and width specification. The modifiers, which were first standardized by POSIX.2-1992 and by ISO C99, are

E Use the locale's alternate representation for date and time. This modifier applies to the %c, %C, %x, %X, %y and %Y format specifiers. In a Japanese locale, for example, %Ex might yield a date format based on the Japanese Emperors' reigns.

O Use the locale's alternate numeric symbols for numbers. This modifier applies only to numeric format specifiers.

If the format supports the modifier but no alternate representation is available, it is ignored.

The conversion specifier ends with a format specifier taken from the following list. The whole '%' sequence is replaced in the output string as follows:

%a This is the abbreviated weekday name according to the current locale.

%A This is the full weekday name according to the current locale.

%b This is the abbreviated month name according to the current locale.

%B This is the full month name according to the current locale.

%c This is the preferred calendar time representation for the current locale.

%C This is the century of the year. This is equivalent to the greatest integer not greater than the year divided by 100. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%d` This is the day of the month as a decimal number (range `01` through `31`).

`%D` This is the date using the format `%m/%d/%y`. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%e` This is the day of the month like with `%d`, but padded with blank (range ` 1` through `31`).

This format was first standardized by POSIX.2-1992 and by ISO C99.

`%F` This is the date using the format `%Y-%m-%d`. This is the form specified in the ISO 8601 standard and is the preferred form for all uses. This format was first standardized by ISO C99 and by POSIX.1-2001.

`%g` This is the year corresponding to the ISO week number, but without the century (range `00` through `99`). This has the same format and value as `%y`, except that if the ISO week number (see `%V`) belongs to the previous or next year, that year is used instead. This format was first standardized by ISO C99 and by POSIX.1-2001.

`%G` This is the year corresponding to the ISO week number. This has the same format and value as `%Y`, except that if the ISO week number (see `%V`) belongs to the previous or next year, that year is used instead. This format was first standardized by ISO C99 and by POSIX.1-2001 but was previously available as a GNU extension.

`%h` This is the abbreviated month name according to the current locale. The action is the same as for `%b`. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%H` This is the hour as a decimal number, using a 24-hour clock (range `00` through `23`).

`%I` This is the hour as a decimal number, using a 12-hour clock (range `01` through `12`).

`%j` This is the day of the year as a decimal number (range `001` through `366`).

`%k` This is the hour as a decimal number, using a 24-hour clock like `%H`, but padded with blank (range ` 0` through `23`). This format is a GNU extension.

`%l` This is the hour as a decimal number, using a 12-hour clock like `%I`, but padded with blank (range ` 1` through `12`). This format is a GNU extension.

`%m` This is the month as a decimal number (range `01` through `12`).

`%M` This is the minute as a decimal number (range `00` through `59`).

`%n` This is a single '`\n`' (newline) character. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%p` This is either '`AM`' or '`PM`', according to the given time value; or the corresponding strings for the current locale. Noon is treated as '`PM`' and midnight as '`AM`'. In most locales '`AM`'/'`PM`' format is not supported; in such cases, "`%p`" yields an empty string.

`%P` This is either '`am`' or '`pm`', according to the given time value; or the corresponding strings for the current locale, printed in lowercase characters. Noon is treated as '`pm`' and midnight as '`am`'. In most locales '`AM`'/'`PM`' format is not supported; in such cases, "`%P`" yields an empty string. This format is a GNU extension.

`%r` This is the complete calendar time using the a.m./p.m. format of the current locale. This format was first standardized by POSIX.2-1992 and by ISO C99. In the POSIX locale, this format is equivalent to `%I:%M:%S %p`.

`%R` This is the hour and minute in decimal numbers using the format `%H:%M`. This format was first standardized by ISO C99 and by POSIX.1-2001 but was previously available as a GNU extension.

`%s` This is the number of seconds since the epoch, i.e., since 1970-01-01 00:00:00 UTC. Leap seconds are not counted unless leap second support is available. This format is a GNU extension.

`%S` This is the seconds as a decimal number (range `00` through `60`).

`%t` This is a single '`\t`' (tabulator) character. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%T` This is the time of day using decimal numbers, in the format `%H:%M:%S`. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%u` This is the day of the week as a decimal number (range `1` through `7`), Monday being `1`. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%U` This is the week number of the current year as a decimal number (range `00` through `53`), starting with the first Sunday as the first day of the first week. Days preceding the first Sunday in the year are considered to be in week `00`.

`%V` This is the ISO 8601:1988 week number as a decimal number (range `01` through `53`). ISO weeks start with Monday and end with Sunday. Week `01` of a year is the first week that has the majority of its days in that year; this is equivalent to the week containing the year's first Thursday, and it is also equivalent to the week containing January 4. Week `01` of a year can contain days from the previous year. The week before week `01` of a year is the last week

(52 or 53) of the previous year even if it contains days from the new year. This format was first standardized by POSIX.2-1992 and by ISO C99.

`%w` This is the day of the week as a decimal number (range 0 through 6), Sunday being 0.

`%W` This is the week number of the current year as a decimal number (range 00 through 53), starting with the first Monday as the first day of the first week. All days preceding the first Monday in the year are considered to be in week 00.

`%x` This is the preferred date representation for the current locale.

`%X` This is the preferred time of day representation for the current locale.

`%y` This is the year without a century as a decimal number (range 00 through 99). This is equivalent to the year modulo 100.

`%Y` This is the year as a decimal number, using the Gregorian calendar. Years before the year 1 are numbered 0, −1, and so on.

`%z` This is the RFC 822/ISO 8601:1988 numeric time zone (e.g., −0600 or +0100), or nothing if no time zone is determinable. This format was first standardized by ISO C99 and by POSIX.1-2001 but was previously available as a GNU extension.

In the POSIX locale, a full RFC 822 time stamp is generated by the format '"%a, %d %b %Y %H:%M:%S %z"' (or the equivalent '"%a, %d %b %Y %T %z"').

`%Z` This is the time zone abbreviation (empty if the time zone can't be determined).

`%%` This is a literal '%' character.

The *size* parameter can be used to specify the maximum number of characters to be stored in the array *s*, including the terminating null character. If the formatted time requires more than *size* characters, `strftime` returns zero and the contents of the array *s* are undefined. Otherwise, the return value indicates the number of characters placed in the array *s*, not including the terminating null character.

Warning: This convention for the return value, prescribed in ISO C, can lead to problems in some situations. For certain format strings and certain locales, the output really can be the empty string, and this cannot be discovered by testing the return value only. For example, in most locales the a.m./p.m. time format is not supported (most of the world uses the 24-hour time representation). In such locales, `"%p"` will return the empty string, i.e., the return value is zero. To detect situations like this, something similar to the following code should be used:

```
buf[0] = '\1';
len = strftime (buf, bufsize, format, tp);
if (len == 0 && buf[0] != '\0')
  {
    /* Something went wrong in the strftime call.  */
    ...
  }
```

If *s* is a null pointer, `strftime` does not actually write anything, but instead returns the number of characters it would have written.

According to POSIX.1, every call to `strftime` implies a call to `tzset`. So the contents of the environment variable `TZ` are examined before any output is produced.

For an example of `strftime`, see Section 10.4.9 [Time Functions Example], page 309.

`size_t` **wcsftime** (`wchar_t *`*s*`, size_t` *size*`, const wchar_t *`*template*`, const struct tm *`*brokentime*) Function

The `wcsftime` function is equivalent to the `strftime` function with the difference that it operates on wide-character strings. The buffer where the result is stored, pointed to by *s*, must be an array of wide characters. The parameter *size*, which specifies the size of the output buffer, gives the number of wide characters, not the number of bytes.

Also, the format string *template* is a wide-character string. Since all characters needed to specify the format string are in the basic character set, it is portably possible to write format strings in the C source code using the `L"..."` notation. The parameter *brokentime* has the same meaning as in the `strftime` call.

The `wcsftime` function supports the same flags, modifiers, and format specifiers as the `strftime` function.

The return value of `wcsftime` is the number of wide characters stored in `s`. When more characters would have to be written than can be placed in the buffer *s* the return value is zero, with the same problems indicated in the `strftime` documentation.

10.4.6 Convert Textual Time and Date Information Back

The ISO C standard does not specify any functions that can convert the output of the `strftime` function back into a binary format. This led to a variety of more-or-less successful implementations with different interfaces over the years. Then the Unix standard was extended by the addition of two functions: `strptime` and `getdate`. Both have strange interfaces, but at least they are widely available.

10.4.6.1 Interpret String According to Given Format

The first function is rather low-level. It is nevertheless frequently used in software since it is better known. Its interface and implementation are heavily influ-

enced by the `getdate` function, which is defined and implemented in terms of calls to `strptime`.

`char *` **strptime** (`const char *`*s*`, const char *`*fmt*`, struct tm *`*tp*) Function

The `strptime` function parses the input string *s* according to the format string *fmt* and stores its results in the structure *tp*.

The input string could be generated by a `strftime` call or obtained any other way. It does not need to be in a human-recognizable format; e.g. a date passed as `"02:1999:9"` is acceptable, even though it is ambiguous without context. As long as the format string *fmt* matches the input string, the function will succeed.

The user has to make sure, though, that the input can be parsed in a unambiguous way. The string `"1999112"` can be parsed using the format `"%Y%m%d"` as 1999-1-12, 1999-11-2, or even 19991-1-2. It is necessary to add appropriate separators to reliably get results.

The format string consists of the same components as the format string of the `strftime` function. The only difference is that the flags _, `-`, `0` and `^` are not allowed. Several of the distinct formats of `strftime` do the same work in `strptime`, since differences like case of the input do not matter. For reasons of symmetry, though, all formats are supported.

The modifiers `E` and `O` are also allowed everywhere the `strftime` function allows them.

The formats are

`%a`
`%A` This is the weekday name according to the current locale, in abbreviated form or the full name.

`%b`
`%B`
`%h` This is the month name according to the current locale, in abbreviated form or the full name.

`%c` This is the date and time representation for the current locale.

`%Ec` This is like `%c`, but the locale's alternative date and time format is used.

`%C` This is the century of the year.

It makes sense to use this format only if the format string also contains the `%y` format.

`%EC` This is the locale's representation of the period.

Unlike `%C`, it sometimes makes sense to use this format since some cultures represent years relative to the beginning of eras instead of using the Gregorian years.

`%d`

`%e` This is the day of the month as a decimal number (range `1` through `31`). Leading zeros are permitted but not required.

`%Od`

`%Oe` This is same as `%d` but using the locale's alternative numeric symbols.

Leading zeros are permitted but not required.

`%D` Equivalent to `%m/%d/%y`.

`%F` This is equivalent to `%Y-%m-%d`, which is the ISO 8601 date format.

This is a GNU extension following an ISO C99 extension to `strftime`.

`%g` This is the year corresponding to the ISO week number, but without the century (range `00` through `99`).

Currently, this is not fully implemented. The format is recognized, input is consumed, but no field in *tm* is set.

This format is a GNU extension following a GNU extension of `strftime`.

`%G` This is the year corresponding to the ISO week number.

Currently, this is not fully implemented. The format is recognized, input is consumed, but no field in *tm* is set.

This format is a GNU extension following a GNU extension of `strftime`.

`%H`

`%k` This is the hour as a decimal number, using a 24-hour clock (range `00` through `23`).

`%k` is a GNU extension following a GNU extension of `strftime`.

`%OH` This is the same as `%H` but using the locale's alternative numeric symbols.

`%I`

`%l` This is the hour as a decimal number, using a 12-hour clock (range `01` through `12`).

`%l` is a GNU extension following a GNU extension of `strftime`.

`%OI` This is the same as `%I` but using the locale's alternative numeric symbols.

`%j` This is the day of the year as a decimal number (range `1` through `366`).

Leading zeros are permitted but not required.

`%m` This is the month as a decimal number (range `1` through `12`).

Leading zeros are permitted but not required.

`%Om` This is the same as `%m` but using the locale's alternative numeric symbols.

`%M` This is the minute as a decimal number (range `0` through `59`).

Leading zeros are permitted but not required.

`%OM` This is the same as `%M` but using the locale's alternative numeric symbols.

`%n`
`%t` This matches any white space.

`%p`

`%P` This is the locale-dependent equivalent to '`AM`' or '`PM`'.

This format is not useful unless `%I` or `%l` is also used. Another complication is that the locale might not define these values at all, and therefore the conversion fails.

`%P` is a GNU extension following a GNU extension to `strftime`.

`%r` This is the complete time using the a.m./p.m. format of the current locale.

A complication is that the locale might not define this format at all, and therefore the conversion fails.

`%R` This is the hour and minute in decimal numbers using the format `%H:%M`.

`%R` is a GNU extension following a GNU extension to `strftime`.

`%s` This is the number of seconds since the epoch, i.e., since 1970-01-01 00:00:00 UTC. Leap seconds are not counted unless leap second support is available.

`%s` is a GNU extension following a GNU extension to `strftime`.

`%S` This is the seconds as a decimal number (range `0` through `60`).

Leading zeros are permitted but not required.

The Unix specification says the upper bound on this value is `61`, a result of a decision to allow double leap seconds. You will not see the value `61`, because no minute has more than one leap second, but the myth persists.

`%OS` This is the same as `%S` but using the locale's alternative numeric symbols.

`%T` This is equivalent to the use of `%H:%M:%S` in this place.

`%u` This is the day of the week as a decimal number (range `1` through `7`), Monday being `1`.

Leading zeros are permitted but not required.

Currently, this is not fully implemented. The format is recognized, input is consumed, but no field in *tm* is set.

`%U` This is the week number of the current year as a decimal number (range `0` through `53`).

Leading zeros are permitted but not required.

`%OU` This is the same as `%U` but using the locale's alternative numeric symbols.

`%V` This is the ISO 8601:1988 week number as a decimal number (range `1` through `53`).

Leading zeros are permitted but not required.

Currently, this is not fully implemented. The format is recognized, input is consumed, but no field in *tm* is set.

`%w` This is the day of the week as a decimal number (range `0` through `6`), Sunday being `0`.

Leading zeros are permitted but not required.

Currently, this is not fully implemented. The format is recognized, input is consumed but no field in *tm* is set.

`%Ow` This is the same as `%w` but using the locale's alternative numeric symbols.

`%W` This is the week number of the current year as a decimal number (range `0` through `53`).

Leading zeros are permitted but not required.

Currently, this is not fully implemented. The format is recognized, input is consumed, but no field in *tm* is set.

`%OW` This is the same as `%W` but using the locale's alternative numeric symbols.

`%x` This is the date using the locale's date format.

`%Ex` This is like `%x`, but the locale's alternative data representation is used.

`%X` This is the time using the locale's time format.

`%EX` This is like `%X`, but the locale's alternative time representation is used.

`%y` This is the year without a century as a decimal number (range `0` through `99`).

Leading zeros are permitted but not required.

It is questionable to use this format without the `%C` format. The `strptime` function does regard input values in the range 68 to 99 as the years 1969 to 1999 and the values 0 to 68 as the years 2000 to 2068. But maybe this heuristic fails for some input data.

Therefore, it is best to avoid `%y` completely and use `%Y` instead.

`%Ey` This is the offset from `%EC` in the locale's alternative representation.

`%Oy` This is the offset of the year (from `%C`) using the locale's alternative numeric symbols.

`%Y` This is the year as a decimal number, using the Gregorian calendar.

`%EY` This is the full alternative year representation.

`%z` This is the offset from GMT in ISO 8601/RFC822 format.

`%Z` This is the time zone name.

Currently, this is not fully implemented. The format is recognized, input is consumed, but no field in *tm* is set.

`%%` This is a literal '`%`' character.

All other characters in the format string must have a matching character in the input string. Exceptions are white spaces in the input string that can match zero or more white-space characters in the format string.

Portability Note: The XPG standard advises applications to use at least one white-space character (as specified by `isspace`) or other nonalphanumeric characters between any two conversion specifications. The GNU C Library does not have this limitation, but other libraries might have trouble parsing formats like `"%d%m%Y%H%M%S"`.

The `strptime` function processes the input string from right to left. Each of the three possible input elements (white space, literal, or format) are handled one after the other. If the input cannot be matched to the format string, the function stops. The remainder of the format and input strings are not processed.

The function returns a pointer to the first character it was unable to process. If the input string contains more characters than required by the format string, the return value points right after the last consumed input character. If the whole input string is consumed, the return value points to the `NULL` byte at the end of the string. If an error occurs, i.e., `strptime` fails to match all of the format string, the function returns `NULL`.

The specification of the function in the XPG standard is rather vague, leaving out a few important pieces of information. Most importantly, it does not specify what happens to those elements of *tm* that are not directly initialized by the different formats. The implementations on different Unix systems vary here.

The GNU libc implementation does not touch those fields that are not directly initialized. Exceptions are the `tm_wday` and `tm_yday` elements, which are recomputed if any of the year, month, or date elements changed. This has two implications:

- Before calling the `strptime` function for a new input string, you should prepare the *tm* structure you pass. Normally this will mean initializing all values to zero. Alternatively, you can set all fields to values like `INT_MAX`,

allowing you to determine which elements were set by the function call. Zero does not work here, since it is a valid value for many of the fields.

Careful initialization is necessary if you want to find out whether a certain field in *tm* was initialized by the function call.

- You can construct a `struct tm` value with several consecutive `strptime` calls. A useful application of this is the parsing of two separate strings, one containing date information and the other time information. By parsing one after the other without clearing the structure in between, you can construct a complete broken-down time.

The following example shows a function that parses a string that contains the date information in either US style or ISO 8601 form:

```
const char *
parse_date (const char *input, struct tm *tm)
{
  const char *cp;

  /* First clear the result structure.  */
  memset (tm, '\0', sizeof (*tm));

  /* Try the ISO format first.  */
  cp = strptime (input, "%F", tm);
  if (cp == NULL)
    {
      /* Does not match.  Try the US form.  */
      cp = strptime (input, "%D", tm);
    }

  return cp;
}
```

10.4.6.2 A More User-Friendly Way to Parse Times and Dates

The Unix standard defines another function for parsing date strings. The interface is weird, but if the function happens to suit your application, it is just fine. It is problematic to use this function in multithreaded programs or libraries, since it returns a pointer to a static variable and uses a global variable and global state (an environment variable).

getdate_err Variable

This variable of type `int` contains the error code of the last unsuccessful call to `getdate`. Defined values are:

1 The environment variable `DATEMSK` is not defined or is null.

2 The template file denoted by the `DATEMSK` environment variable cannot be opened.

3	Information about the template file cannot retrieved.
4	The template file is not a regular file.
5	An I/O error occurred while reading the template file.
6	Not enough memory is available to execute the function.
7	The template file contains no matching template.
8	The input date is invalid, but would match a template otherwise. This includes dates like February 31st, and dates that cannot be represented in a `time_t` variable.

`struct tm *` **getdate** (`const char *`*string*) Function

The interface to `getdate` is the simplest possible way for a function to parse a string and return the value. *string* is the input string, and the result is returned in a statically allocated variable.

The details about how the string is processed are hidden from the user. In fact, they can be outside the control of the program. Which formats are recognized is controlled by the file named by the environment variable `DATEMSK`. This file should contain lines of valid format strings that could be passed to `strptime`.

The `getdate` function reads these format strings one after the other and tries to match the input string. The first line that completely matches the input string is used.

Elements not initialized through the format string retain the values present at the time of the `getdate` function call.

The formats recognized by `getdate` are the same as for `strptime`. See above for an explanation. There are only a few extensions to the `strptime` behavior:

- If the `%Z` format is given, the broken-down time is based on the current time of the time zone matched, not of the current time zone of the run-time environment.

  Currently, this is not implemented. The problem is that time zone names are not unique. If a fixed time zone is assumed for a given string (say `EST` meaning US East Coast time), then uses for countries other than the United States will fail. So far, we have found no good solution to this.
- If only the weekday is specified, the selected day depends on the current date. If the current weekday is greater or equal to the `tm_wday` value, the current week's day is chosen, otherwise the day next week is chosen.
- A similar heuristic is used when only the month is given and not the year. If the month is greater than or equal to the current month, then the current year is used. Otherwise, it wraps to next year. The first day of the month is assumed if one is not explicitly specified.
- The current hour, minute and second are used if the appropriate value is not set through the format.

- If no date is given, tomorrow's date is used if the time is smaller than the current time. Otherwise, today's date is taken.

The format in the template file need not contain only format elements. The following is a list of possible format strings (taken from the Unix standard):

```
%m
%A %B %d, %Y %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

As you can see, the template list can contain very specific strings like `run job at %I %p,%B %dnd`. Using the above list of templates and assuming the current time is Mon Sep 22 12:19:47 EDT 1986, we can obtain the following results for the given input.

'Input'	Match	Result
'Mon'	*%a*	'Mon Sep 22 12:19:47 EDT 1986'
'Sun'	*%a*	'Sun Sep 28 12:19:47 EDT 1986'
'Fri'	*%a*	'Fri Sep 26 12:19:47 EDT 1986'
'September'	*%B*	'Mon Sep 1 12:19:47 EDT 1986'
'January'	*%B*	'Thu Jan 1 12:19:47 EST 1987'
'December'	*%B*	'Mon Dec 1 12:19:47 EST 1986'
'Sep Mon'	*%b %a*	'Mon Sep 1 12:19:47 EDT 1986'
'Jan Fri'	*%b %a*	'Fri Jan 2 12:19:47 EST 1987'
'Dec Mon'	*%b %a*	'Mon Dec 1 12:19:47 EST 1986'
'Jan Wed 1989'	*%b %a %Y*	'Wed Jan 4 12:19:47 EST 1989'
'Fri 9'	*%a %H*	'Fri Sep 26 09:00:00 EDT 1986'
'Feb 10:30'	*%b %H:%S*	'Sun Feb 1 10:00:30 EST 1987'
'10:30'	*%H:%M*	'Tue Sep 23 10:30:00 EDT 1986'
'13:30'	*%H:%M*	'Mon Sep 22 13:30:00 EDT 1986'

The return value of the function is a pointer to a static variable of type `struct tm`, or a null pointer if an error occurred. The result is only valid until the next `getdate` call, making this function unusable in multithreaded applications.

The `errno` variable is *not* changed. Error conditions are stored in the global variable `getdate_err`. See the description above for a list of the possible error values.

Warning: The `getdate` function should *never* be used in SUID-programs. Using the `DATEMSK` environment variable, you can get the function to open any arbitrary file, and chances are high that with some bogus input (such as a binary file), the program will crash.

`int` **getdate_r** (`const char *`*string*`, struct tm *`*tp*) Function

The `getdate_r` function is the reentrant counterpart of `getdate`. It does not use the global variable `getdate_err` to signal an error, but instead returns an error code. The same error codes as described in the `getdate_err` documentation above are used, with 0 meaning success.

Moreover, `getdate_r` stores the broken-down time in the variable of type `struct tm` pointed to by the second argument, rather than in a static variable.

This function is not defined in the Unix standard. Nevertheless, it is available on some other Unix systems as well.

The warning against using `getdate` in SUID-programs applies to `getdate_r`.

10.4.7 Specifying the Time Zone with `TZ`

In POSIX systems, a user can specify the time zone by means of the `TZ` environment variable. For information about how to set environment variables, see Section 14.4 [Environment Variables], page 418. The functions for accessing the time zone are declared in '`time.h`'.

You should not normally need to set `TZ`. If the system is configured properly, the default time zone will be correct. You might set `TZ` if you are using a computer over a network from a different time zone, and would like times reported to you in the time zone local to you, rather than what is local to the computer.

In POSIX.1 systems the value of the `TZ` variable can be in one of three formats. With the GNU C Library, the most common format is the last one, which can specify a selection from a large database of time zone information for many regions of the world. The first two formats are used to describe the time zone information directly, which is both more cumbersome and less precise. But the POSIX.1 standard only specifies the details of the first two formats, so it is good to be familiar with them in case you come across a POSIX.1 system that doesn't support a time zone information database.

The first format is used when there is no daylight saving time (or summer time) in the local time zone:

std offset

The *std* string specifies the name of the time zone. It must be three or more characters long and must not contain a leading colon, embedded digits, commas, or plus or minus signs. There is no space character separating the time zone name from the *offset*, so these restrictions are necessary to parse the specification correctly.

The *offset* specifies the time value you must add to the local time to get a Coordinated Universal Time value. It has syntax like [`+`|`-`]*hh*[`:`*mm*[`:`*ss*]]. This is positive if the local time zone is west of the prime meridian and negative if it is east. The hour must be between `0` and `23` and the minute and seconds between `0` and `59`.

For example, here is how we would specify eastern standard time, but without any daylight saving time alternative:

```
EST+5
```

The second format is used when there is daylight saving time:

std offset dst [*offset*]*,* *start*[*/time*]*,* *end*[*/time*]

The initial *std* and *offset* specify the standard time zone, as described above. The *dst* string and *offset* specify the name and offset for the corresponding daylight saving time zone; if the *offset* is omitted, it defaults to one hour ahead of standard time.

The remainder of the specification describes when daylight saving time is in effect. The *start* field is when daylight saving time goes into effect, and the *end* field is when the change is made back to standard time. The following formats are recognized for these fields:

`J`*n* This specifies the Julian day, with *n* between `1` and `365`. February 29 is never counted, even in leap years.

n This specifies the Julian day, with *n* between `0` and `365`. February 29 is counted in leap years.

`M`*m*`.`*w*`.`*d* This specifies day *d* of week *w* of month *m*. The day *d* must be between `0` (Sunday) and `6`. The week *w* must be between `1` and `5`; week `1` is the first week in which day *d* occurs, and week `5` specifies the *last d* day in the month. The month *m* should be between `1` and `12`.

The *time* fields specify when, in the local time currently in effect, the change to the other time occurs. If omitted, the default is `02:00:00`.

For example, here is how you would specify the eastern time zone in the United States, including the appropriate daylight saving time and its dates of applicability. The normal offset from UTC is five hours; since this is west of the prime meridian, the sign is positive. Summer time begins on the first Sunday in April at 2:00 a.m., and ends on the last Sunday in October at 2:00 a.m.

```
EST+5EDT,M4.1.0/2,M10.5.0/2
```

The schedule of daylight saving time in any particular jurisdiction has changed over the years. To be strictly correct, the conversion of dates and times in the past should be based on the schedule that was in effect then. However, this format has no facilities to let you specify how the schedule has changed from year to year. The most you can do is specify one particular schedule—usually the present day schedule—and this is used to convert any date, no matter when. For precise time zone specifications, it is best to use the time zone information database (see below).

The third format looks like this:

:characters

Each operating system interprets this format differently; in the GNU C Library, *characters* is the name of a file that describes the time zone.

If the `TZ` environment variable does not have a value, the operation chooses a time zone by default. In the GNU C Library, the default time zone is like the specification '`TZ=:/etc/localtime`' (or

‘TZ=:/usr/local/etc/localtime’, depending on how GNU C Library was configured.[5] Other C libraries use their own rule for choosing the default time zone, so there is little we can say about them.

If *characters* begins with a slash, it is an absolute file name; otherwise the library looks for the file ‘/share/lib/zoneinfo/*characters*’. The ‘zoneinfo’ directory contains data files describing local time zones in many different parts of the world. The names represent major cities, with subdirectories for geographical areas; for example, ‘America/New_York’, ‘Europe/London’, ‘Asia/Hong_Kong’. These data files are installed by the system administrator, who also sets ‘/etc/localtime’ to point to the data file for the local time zone. The GNU C Library comes with a large database of time zone information for most regions of the world, which is maintained by a community of volunteers and put in the public domain.

10.4.8 Functions and Variables for Time Zones

`char *` **tzname [2]** Variable

The array `tzname` contains two strings, which are the standard names of the pair of time zones (standard and daylight saving) that the user has selected. `tzname[0]` is the name of the standard time zone (for example, `"EST"`), and `tzname[1]` is the name for the time zone when daylight saving time is in use (for example, `"EDT"`). These correspond to the *std* and *dst* strings (respectively) from the `TZ` environment variable. If daylight saving time is never used, `tzname[1]` is the empty string.

The `tzname` array is initialized from the `TZ` environment variable whenever `tzset`, `ctime`, `strftime`, `mktime` or `localtime` is called. If multiple abbreviations have been used (e.g., `"EWT"` and `"EDT"` for US eastern war time and eastern daylight time), the array contains the most recent abbreviation.

The `tzname` array is required for POSIX.1 compatibility, but in GNU programs it is better to use the `tm_zone` member of the broken-down time structure, since `tm_zone` reports the correct abbreviation even when it is not the latest one.

Though the strings are declared as `char *`, the user must refrain from modifying these strings. Modifying the strings will almost certainly lead to trouble.

`void` **tzset** (`void`) Function

The `tzset` function initializes the `tzname` variable from the value of the `TZ` environment variable. It is not usually necessary for your program to call this function, because it is called automatically when you use the other time conversion functions that depend on the time zone.

The following variables are defined for compatibility with System V Unix. Like `tzname`, these variables are set by calling `tzset` or the other time conversion functions.

[5] Ibid., “Installing the GNU C Library”.

`long int` **timezone** Variable

This contains the difference between UTC and the latest local standard time, in seconds west of UTC. For example, in the US eastern time zone, the value is `5*60*60`. Unlike the `tm_gmtoff` member of the broken-down time structure, this value is not adjusted for daylight saving, and its sign is reversed. In GNU programs it is better to use `tm_gmtoff`, since it contains the correct offset even when it is not the latest one.

`int` **daylight** Variable

This variable has a nonzero value if daylight saving time rules apply. A nonzero value does not necessarily mean that daylight saving time is now in effect; it means only that daylight saving time is sometimes in effect.

10.4.9 Time Functions Example

Here is an example program showing the use of some of the calendar time functions:

```
#include <time.h>
#include <stdio.h>

#define SIZE 256

int
main (void)
{
  char buffer[SIZE];
  time_t curtime;
  struct tm *loctime;

  /* Get the current time. */
  curtime = time (NULL);

  /* Convert it to local time representation. */
  loctime = localtime (&curtime);

  /* Print out the date and time in the standard format. */
  fputs (asctime (loctime), stdout);

  /* Print it out in a nice format. */
  strftime (buffer, SIZE, "Today is %A, %B %d.\n", loctime);
  fputs (buffer, stdout);
  strftime (buffer, SIZE, "The time is %I:%M %p.\n", loctime);
  fputs (buffer, stdout);

  return 0;
```

```
}
```

It produces output like this:

```
Wed Jul 31 13:02:36 1991
Today is Wednesday, July 31.
The time is 01:02 PM.
```

10.5 Setting an Alarm

The `alarm` and `setitimer` functions provide a mechanism for a process to interrupt itself in the future. They do this by setting a timer; when the timer expires, the process receives a signal.

Each process has three independent interval timers available:

- A real-time timer that counts elapsed time. This timer sends a `SIGALRM` signal to the process when it expires.
- A virtual timer that counts processor time used by the process. This timer sends a `SIGVTALRM` signal to the process when it expires.
- A profiling timer that counts both processor time used by the process and processor time spent in system calls on behalf of the process. This timer sends a `SIGPROF` signal to the process when it expires.

  This timer is useful for profiling in interpreters. The interval timer mechanism does not have the fine granularity necessary for profiling native code.

You can only have one timer of each kind set at any given time. If you set a timer that has not yet expired, that timer is simply reset to the new value.

You should establish a handler for the appropriate alarm signal using `signal` or `sigaction` before issuing a call to `setitimer` or `alarm`. Otherwise, an unusual chain of events could cause the timer to expire before your program establishes the handler. In this case it would be terminated, since termination is the default action for the alarm signals.[6]

To be able to use the alarm function to interrupt a system call that might otherwise block indefinitely, it is important to *not* set the `SA_RESTART` flag when registering the signal handler using `sigaction`. When not using `sigaction`, things get even uglier—the `signal` function has to fix semantics with respect to restarts. The BSD semantics for this function is to set the flag. Therefore, if `sigaction` for whatever reason cannot be used, it is necessary to use `sysv_signal` and not `signal`.

The `setitimer` function is the primary means for setting an alarm. This facility is declared in the header file ‘`sys/time.h`’. The `alarm` function, declared in ‘`unistd.h`’, provides a somewhat simpler interface for setting the real-time timer.

---

[6] Ibid., “Signal Handling”.

struct itimerval Data Type

This structure is used to specify when a timer should expire. It contains the following members:

`struct timeval it_interval`
: This is the period between successive timer interrupts. If zero, the alarm will only be sent once.

`struct timeval it_value`
: This is the period between now and the first timer interrupt. If zero, the alarm is disabled.

The `struct timeval` data type is described in Section 10.2 [Elapsed Time], page 277.

`int` **setitimer** (`int` *which*, `struct itimerval` **new*, `struct itimerval` **old*) Function

The `setitimer` function sets the timer specified by *which* according to *new*. The *which* argument can have a value of `ITIMER_REAL`, `ITIMER_VIRTUAL`, or `ITIMER_PROF`.

If *old* is not a null pointer, `setitimer` returns information about any previous unexpired timer of the same kind in the structure it points to.

The return value is `0` on success and `-1` on failure. The following `errno` error conditions are defined for this function:

`EINVAL` The timer period is too large.

`int` **getitimer** (`int` *which*, `struct itimerval` **old*) Function

The `getitimer` function stores information about the timer specified by *which* in the structure pointed at by *old*.

The return value and error conditions are the same as for `setitimer`.

`ITIMER_REAL`
: This constant can be used as the *which* argument to the `setitimer` and `getitimer` functions to specify the real-time timer.

`ITIMER_VIRTUAL`
: This constant can be used as the *which* argument to the `setitimer` and `getitimer` functions to specify the virtual timer.

`ITIMER_PROF`
: This constant can be used as the *which* argument to the `setitimer` and `getitimer` functions to specify the profiling timer.

`unsigned int` **alarm** (`unsigned int` *seconds*) Function

The `alarm` function sets the real-time timer to expire in *seconds* seconds.[7] If you want to cancel any existing alarm, you can do this by calling `alarm` with a *seconds* argument of zero.

The return value indicates how many seconds remain before the previous alarm would have been sent. If there is no previous alarm, `alarm` returns zero.

The `alarm` function could be defined in terms of `setitimer` like this:

```
unsigned int
alarm (unsigned int seconds)
{
  struct itimerval old, new;
  new.it_interval.tv_usec = 0;
  new.it_interval.tv_sec = 0;
  new.it_value.tv_usec = 0;
  new.it_value.tv_sec = (long int) seconds;
  if (setitimer (ITIMER_REAL, &new, &old) < 0)
    return 0;
  else
    return old.it_value.tv_sec;
}
```

If you simply want your process to wait for a given number of seconds, you should use the `sleep` function (see Section 10.6 [Sleeping], page 312).

You shouldn't count on the signal arriving precisely when the timer expires. In a multiprocessing environment, there is typically some amount of delay involved.

Portability Note: The `setitimer` and `getitimer` functions are derived from BSD Unix, while the `alarm` function is specified by the POSIX.1 standard. `setitimer` is more powerful than `alarm`, but `alarm` is more widely used.

10.6 Sleeping

The function `sleep` gives a simple way to make the program wait for a short interval. If your program doesn't use signals (except to terminate), then you can expect `sleep` to wait reliably throughout the specified interval. Otherwise, `sleep` can return sooner if a signal arrives; if you want to wait for a given interval regardless of signals, use `select` and don't specify any descriptors to wait for.[8]

`unsigned int` **sleep** (`unsigned int` *seconds*) Function

The `sleep` function waits for *seconds* or until a signal is delivered, whichever happens first.

---

[7] See Loosemore et al., "Signal Handlers That Return", for an example showing the use of the `alarm` function.

[8] Ibid., "Waiting for Input or Output".

If the `sleep` function returns because the requested interval is over, it returns a value of zero. If it returns because of delivery of a signal, its return value is the remaining time in the sleep interval.

The `sleep` function is declared in '`unistd.h`'.

Resist the temptation to implement a sleep for a fixed amount of time by using the return value of `sleep`, when nonzero, to call `sleep` again. This will work with a certain amount of accuracy as long as signals arrive infrequently. But each signal can cause the eventual wake-up time to be off by an additional second or so. Suppose a few signals happen to arrive in rapid succession by bad luck—there is no limit on how much this could shorten or lengthen the wait.

Instead, compute the calendar time at which the program should stop waiting, and keep trying to wait until that calendar time. This won't be off by more than a second. With just a little more work, you can use `select` and make the waiting period quite accurate. Of course, heavy system load can cause additional unavoidable delays—unless the machine is dedicated to one application, there is no way you can avoid this.

On some systems, `sleep` can do strange things if your program uses `SIGALRM` explicitly. Even if `SIGALRM` signals are being ignored or blocked when `sleep` is called, `sleep` might return prematurely on delivery of a `SIGALRM` signal. If you have established a handler for `SIGALRM` signals and a `SIGALRM` signal is delivered while the process is sleeping, the action taken might be just to cause `sleep` to return instead of to invoke your handler. And, if `sleep` is interrupted by delivery of a signal whose handler requests an alarm or alters the handling of `SIGALRM`, this handler and `sleep` will interfere.

On the GNU system, it is safe to use `sleep` and `SIGALRM` in the same program, because `sleep` does not work by means of `SIGALRM`.

`int` **nanosleep** (`const struct timespec` **requested_time*, `struct timespec` **remaining*) Function

If resolution to seconds is not enough, the `nanosleep` function can be used. As the name suggests, the sleep interval can be specified in nanoseconds. The actual elapsed time of the sleep interval might be longer, since the system rounds the elapsed time you request up to the next integer multiple of the actual resolution the system can deliver.

`*requested_time` is the elapsed time of the interval you want to sleep.

The function returns as `*remaining` the elapsed time left in the interval for which you requested to sleep. If the interval completed without getting interrupted by a signal, this is zero.

`struct timespec` is described in Section 10.2 [Elapsed Time], page 277.

If the function returns because the interval is over, the return value is zero. If the function returns -1, the global variable *errno* is set to the following values:

`EINTR` The call was interrupted because a signal was delivered to the thread. If the *remaining* parameter is not the null pointer, the struc-

ture pointed to by *remaining* is updated to contain the remaining elapsed time.

`EINVAL` The nanosecond value in the *requested_time* parameter contains an illegal value. Either the value is negative or greater than or equal to 1000 million.

This function is a cancellation point in multithreaded programs. This is a problem if the thread allocates some resources (like memory, file descriptors, semaphores or whatever) at the time `nanosleep` is called. If the thread gets canceled, these resources stay allocated until the program ends. To avoid this, calls to `nanosleep` should be protected using cancellation handlers.

The `nanosleep` function is declared in '`time.h`'.

11 Message Translation

The program's interface with a user should be designed to make tasks easier for the user. One way it can do this is to use messages in whichever language the user prefers.

Printing messages in different languages can be implemented in different ways. You could add all the different languages in the source code and add among the variants every time a message has to be printed. This is certainly not a good solution, since extending the set of languages is difficult (the code must be changed), and the code itself can become really big with dozens of message sets.

A better solution is to keep the message sets for each language in separate files that are loaded at run time depending on the language selection of the user.

The GNU C Library provides two different sets of functions to support message translation. The problem is that neither of the interfaces is officially defined by the POSIX standard. The `catgets` family of functions is defined in the X/Open standard, but this is derived from industry decisions and therefore not necessarily based on reasonable decisions.

As mentioned above, the message catalog handling provides easy extendability by using external data files that contain the message translations. These files contain for each of the messages used in the program a translation for the appropriate language. So the tasks of the message handling functions are

- Locate the external data file with the appropriate translations.
- Load the data and make it possible to address the messages.
- Map a given key to the translated message.

The two approaches mainly differ in the implementation of this last step. The design decisions made for this step influence the rest.

11.1 X/Open Message Catalog Handling

The `catgets` functions are based on the simple scheme:

> Associate every message to translate in the source code with a unique identifier. To retrieve a message from a catalog file, just the identifier is used.

This means for the author of the program that he will have to make sure the meaning of the identifier in the program code and in the message catalogs are always the same.

Before a message can be translated, the catalog file must be located. The user of the program must be able to guide the responsible function to find whichever catalog she wants.

All the types, constants and functions for the `catgets` functions are defined/declared in the '`nl_types.h`' header file.

11.1.1 The `catgets` Function Family

`nl_catd` **catopen** (`const char *`*cat_name*`, int` *flag*) Function

The `catgets` function tries to locate the message data file name *cat_name* and loads it when found. The return value is of an opaque type and can be used in calls to the other functions to refer to this loaded catalog.

The return value is `(nl_catd) -1` in case the function failed and no catalog was loaded. The global variable *errno* contains a code for the error causing the failure. But even if the function call succeeded, this does not mean that all messages can be translated.

Locating the catalog file must happen in a way that lets the user of the program influence the decision. It is up to the user to decide about the language to use, and sometimes it is useful to use alternate catalog files. All this can be specified by the user by setting some environment variables.

The first problem is to find out where all the message catalogs are stored. Every program could have its own place to keep all the different files, but usually the catalog files are grouped by languages and the catalogs for all programs are kept in the same place.

To tell the `catopen` function where the catalog for the program can be found, the user can set the environment variable `NLSPATH` to a value that describes her choice. Since this value must be usable for different languages and locales, it cannot be a simple string. Instead, it is a format string (similar to `printf`'s). An example is

```
/usr/share/locale/%L/%N:/usr/share/locale/%L/LC_MESSAGES/%N
```

First, one can see that more than one directory can be specified (with the usual syntax of separating them by colons). The next things to observe are the format strings, `%L` and `%N` in this case. The `catopen` function knows about several of them, and the replacement for all of them is of course different.

`%N` This format element is substituted with the name of the catalog file. This is the value of the *cat_name* argument given to `catgets`.

`%L` This format element is substituted with the name of the currently selected locale for translating messages. How this is determined is explained below.

`%l` This is the lowercase ell. This format element is substituted with the language element of the locale name. The string describing the selected locale is expected to have the form *lang*`[_`*terr*`[.`*codeset*`]]`, and this format uses the first part *lang*.

`%t` This format element is substituted by the territory part *terr* of the name of the currently selected locale. See the explanation of the format above.

`%c` This format element is substituted by the codeset part *codeset* of the name of the currently selected locale. See the explanation of the format above.

`%%` Since `%` is used in a meta character, there must be a way to express the `%` character in the result itself. Using `%%` does this just like it works for `printf`.

Using `NLSPATH` allows arbitrary directories to be searched for message catalogs while still allowing different languages to be used. If the `NLSPATH` environment variable is not set, the default value is

prefix`/share/locale/%L/%N:`*prefix*`/share/locale/%L/LC_MESSAGES/%N`

where *prefix* is given to `configure` while installing the GNU C Library (this value is in many cases `/usr` or the empty string).

The remaining problem is to decide which must be used. The value decides about the substitution of the format elements mentioned above. First of all, the user can specify a path in the message catalog name (i.e., the name contains a slash character). In this situation the `NLSPATH` environment variable is not used. The catalog must exist as specified in the program, perhaps relative to the current working directory. This situation in not desirable and catalog names should never be written this way. Besides this, this behavior is not portable to all other platforms providing the `catgets` interface.

Otherwise, the values of environment variables from the standard environment are examined (see Section 14.4.2 [Standard Environment Variables], page 421). Which variables are examined is decided by the *flag* parameter of `catopen`. If the value is `NL_CAT_LOCALE` (which is defined in '`nl_types.h`'), then the `catopen` function uses the name of the locale currently selected for the `LC_MESSAGES` category.

If *flag* is zero, the `LANG` environment variable is examined. This is a leftover from the early days where the concept of the locales had not even reached the level of POSIX locales.

The environment variable and the locale name should have a value of the form *lang*`[_`*terr*`[.`*codeset*`]]` as explained above. If no environment variable is set, the `"C"` locale is used, which prevents any translation.

The return value of the function is in any case a valid string. Either it is a translation from a message catalog or it is the same as the *string* parameter. So a piece of code to decide whether a translation actually happened must look like this:

```
{
  char *trans = catgets (desc, set, msg, input_string);
  if (trans == input_string)
    {
      /* Something went wrong.  */
    }
}
```

When an error occurred the global variable *errno* is set to:

EBADF The catalog does not exist.

ENOMSG The set/message tuple does not name an existing element in the message catalog.

While it sometimes can be useful to test for errors, programs normally will avoid any test. If the translation is not available, it is not a big problem if the original, untranslated message is printed. Either the user understands this as well, or he will look for the reason why the messages are not translated.

The currently selected locale does not depend on a call to the `setlocale` function. It is not necessary for the locale data files for this locale to exist and for calling `setlocale` to succeed. The `catopen` function directly reads the values of the environment variables.

`char *` **catgets** (`nl_catd` *catalog_desc*, `int` *set*, `int` *message*, `const char *`*string*) Function

The function `catgets` has to be used to access the massage catalog previously opened using the `catopen` function. The *catalog_desc* parameter must be a value previously returned by `catopen`.

The next two parameters, *set* and *message*, reflect the internal organization of the message catalog files. This will be explained in detail below. For now it is interesting to know that a catalog can consist of several sets and the messages in each thread are individually numbered using numbers. Neither the set number nor the message number must be consecutive. They can be arbitrarily chosen. But each message (unless equal to another one) must have its own unique pair of set and message number.

Since it is not guaranteed that the message catalog for the language selected by the user exists, the last parameter *string* helps to handle this case gracefully. If no matching string can be found, *string* is returned. This means for the programmer that:

- The *string* parameters should contain reasonable text (this also helps to understand the program, since otherwise there would be no hint as to the string that is expected to be returned.
- All *string* arguments should be written in the same language.

It is somewhat uncomfortable to write a program using the `catgets` functions if no supporting functionality is available. Since each set/message number tuple must be unique, the programmer must keep lists of the messages at the same time the code is written. And the work between several people working on the same project must be coordinated. We will see how these problems can be relaxed a bit (see Section 11.1.4 [How to Use the `catgets` Interface], page 322).

`int` **catclose** (`nl_catd` *catalog_desc*) Function

The `catclose` function can be used to free the resources associated with a message catalog that previously was opened by a call to `catopen`. If the resources can be successfully freed, the function returns `0`. Otherwise, it returns `-1` and the global variable *errno* is set. Errors can occur if the catalog descriptor *catalog_desc* is not valid, in which case *errno* is set to `EBADF`.

11.1.2 Format of the Message Catalog Files

The only reasonable way to translate all the messages of a function and store the result in a message catalog file that can be read by the `catopen` function is to write all the message text to the translator and let her translate them all; we must have a file with entries that associate the set/message tuple with a specific translation. This file format is specified in the X/Open standard and is as follows:

- Lines containing only white-space characters or empty lines are ignored.
- Lines that contain as the first non-white-space character a `$` followed by a white-space character are comment and are also ignored.
- If a line contains as the first non-white-space characters the sequence `$set` followed by a white-space character, an additional argument is required to follow. This argument can either be
  - A number; in this case, the value of this number determines the set to which the following messages are added.
  - An identifier consisting of alphanumeric characters plus the underscore character; in this case, the set automatically gets a number assigned. This value is one added to the largest set number that has appeared so far.

    How to use the symbolic names is explained in Section 11.1.4 [How to Use the `catgets` Interface], page 322.

    It is an error if a symbol name appears more than once. All following messages are placed in a set with this number.
- If a line contains as the first non-white-space characters the sequence `$delset` followed by a white-space character, an additional argument is required to follow. This argument can either be
  - A number; in this case, the value of this number determines the set that will be deleted.
  - An identifier consisting of alphanumeric characters plus the underscore character; this symbolic identifier must match a name for a set that was previously defined. It is an error if the name is unknown.

  In both cases, all messages in the specified set will be removed. They will not appear in the output. But if this set is selected again later with a `$set` command, messages could be added again, and these messages will appear in the output.
- If a line contains after leading white spaces the sequence `$quote`, the quoting character used for this input file is changed to the first non-white-space character following the `$quote`. If no non-white-space character is present before the line ends, quoting is disabled.

  By default, no quoting character is used. In this mode, strings are terminated with the first unescaped line break. If there is a `$quote` sequence present, newline need not be escaped. Instead, a string is terminated with the first unescaped appearance of the quote character.

A common usage of this feature would be to set the quote character to '"'. Then any appearance of the '"' in the strings must be escaped using the backslash (i.e., '\"' must be written).

- Any other line must start with a number or an alphanumeric identifier (with the underscore character included). The following characters (starting after the first white-space character) will form the string that gets associated with the currently selected set and the message number represented by the number and identifier respectively.

  If the start of the line is a number, the message number is obvious. It is an error if the same message number already appeared for this set.

  If the leading token was an identifier, the message number gets automatically assigned. The value is the current maximum messages number for this set plus one. It is an error if the identifier was already used for a message in this set. It is OK to reuse the identifier for a message in another thread. How to use the symbolic identifiers will be explained below (see Section 11.1.4 [How to Use the `catgets` Interface], page 322). There is one limitation with the identifier: it must not be `Set`. The reason will be explained below.

  The text of the messages can contain escape characters. The usual bunch of characters known from the ISO C language are recognized ('\n', '\t', '\v', '\b', '\r', '\f', '\\', and '*nnn*', where *nnn* is the octal coding of a character code).

Important: The handling of identifiers instead of numbers for the set and messages is a GNU extension. Systems strictly following the X/Open specification do not have this feature. An example for a message catalog file is this:

```
$ This is a leading comment.
$quote "

$set SetOne
1 Message with ID 1.
two "   Message with ID \"two\", which gets the value 2 assigned"

$set SetTwo
$ Since the last set got the number 1 assigned this set has number 2.
4000 "The numbers can be arbitrary, they need not start at one."
```

This small example shows various aspects:

- Lines 1 and 9 are comments, since they start with `$` followed by a white space.
- The quoting character is set to '"'. Otherwise, the quotes in the message definition would have to be left away, and in this case, the message with the identifier `two` would lose its leading white space.
- Mixing numbered messages with messages having symbolic names is no problem, and the numbering happens automatically.

While this file format is pretty easy, it is not the best possible for use in a running program. The `catopen` function would have to parse the file and handle syn-

tactic errors gracefully. This is not so easy, and the whole process is pretty slow. Therefore, the `catgets` functions expect the data in another more compact and ready-to-use file format. There is a special program `gencat`, which is explained in detail in the next section.

Files in this other format are not human readable. To be easy to use for programs, it is a binary file. But the format is byte-order independent, so translation files can be shared by systems of arbitrary architecture (as long as they use the GNU C Library).

Details about the binary file format are not important to know, since these files are always created by the `gencat` program. The sources of the GNU C Library also provide the sources for the `gencat` program, so the interested reader can look through these source files to learn about the file format.

11.1.3 Generate Message Catalogs Files

The `gencat` program is specified in the X/Open standard, and the GNU implementation follows this specification, so it processes all correctly formed input files. Additionally, some extension are implemented that help to work in a more reasonable way with the `catgets` functions.

The `gencat` program can be invoked in two ways:

`'gencat [`*Option*`]... [`*Output-File* `[`*Input-File*`]...]'`

This is the interface defined in the X/Open standard. If no *Input-File* parameter is given, input will be read from standard input. Multiple input files will be read as if they are concatenated. If *Output-File* is also missing, the output will be written to standard output. To provide the interface one is used to from other programs, a second interface is provided.

`'gencat [`*Option*`]... -o` *Output-File* `[`*Input-File*`]...'`

The option '`-o`' is used to specify the output file, and all file arguments are used as input files.

Besides this, you can use '`-`' or '`/dev/stdin`' for *Input-File* to denote the standard input. Correspondingly, one can use '`-`' and '`/dev/stdout`' for *Output-File* to denote standard output. Using '`-`' as a file name is allowed in X/Open, while using the device names is a GNU extension.

The `gencat` program works by concatenating all input files and then merging the resulting collection of message sets with a possibly existing output file. This is done by removing all messages with set/message number tuples matching any of the generated messages from the output file, and then adding all the new messages. To regenerate a catalog file while ignoring the old contents therefore requires you to remove the output file if it exists. If the output is written to standard output, no merging takes place.

The following table shows the options understood by the `gencat` program; the X/Open standard does not specify any option for the program, so all of these are GNU extensions:

‘-V’
‘--version’
Print the version information and exit.

‘-h’
‘--help’ Print a usage message listing all available options, then exit successfully.

‘--new’ Never merge the new messages from the input files with the old content of the output files. The old content of the output files is discarded.

‘-H’
‘--header=name’
This option is used to emit the symbolic names given to sets and messages in the input files for use in the program. Details about how to use this are given in the next section. The *name* parameter to this option specifies the name of the output file. It will contain a number of C preprocessor `#defines` to associate a name with a number.

The generated file only contains the symbols from the input files. If the output is merged with the previous content of the output file, the possibly existing symbols from the file(s) that generated the old output files are not in the generated header file.

11.1.4 How to Use the `catgets` Interface

The `catgets` functions can be used in two different ways: by slavishly following the X/Open specs and not relying on the extensions or by using the GNU extensions. We will take a look at the former method first to understand the benefits of extensions.

11.1.4.1 Not Using Symbolic Names

Since the X/Open format of the message catalog files does not allow symbol names, we have to work with numbers all the time. When we start writing a program, we have to replace all appearances of translatable strings with something like:

```
catgets (catdesc, set, msg, "string")
```

catgets is retrieved from a call to `catopen`, which is normally done once at the program start. The `"string"` is the string we want to translate. The problems start with the set and message numbers.

In a bigger program, several programmers usually work at the same time on the program, so coordinating the number allocation is crucial. Though no two different strings must be indexed by the same tuple of numbers, it is highly desirable to reuse the numbers for equal strings with equal translations (there might be strings that are equal in one language but have different translations due to different contexts).

The allocation process can be relaxed a bit by different set numbers for different parts of the program. So the number of developers who have to coordinate the allo-

cation can be reduced. But still, lists must be made to keep track of the allocation, and errors can easily happen. These errors cannot be discovered by the compiler or the `catgets` functions. Only the user of the program might see wrong messages printed. In the worst cases, the messages are so irritating that they cannot be recognized as wrong. Think about the translations for `"true"` and `"false"` being exchanged. This could result in a disaster.

11.1.4.2 Using Symbolic Names

The problems mentioned in the last section derive from the fact that:

1. The numbers are allocated once, and due to the possibly frequent use of them, it is difficult to change a number later.
2. The numbers do not allow you to guess anything about the string, and therefore collisions can easily happen.

By constantly using symbolic names and by providing a method that maps the string content to a symbolic name (however this will happen), you can prevent both of the above problems. The cost of this is that the programmer has to write a complete message catalog file while she is writing the program itself.

This is necessary since the symbolic names must be mapped to numbers before the program sources can be compiled. In the last section, it was described how to generate a header containing the mapping of the names. For the example message file given in the last section, we could call the `gencat` program as follows (assume '`ex.msg`' contains the sources):

```
gencat -H ex.h -o ex.cat ex.msg
```

This generates a header file with the following content:

```
#define SetTwoSet 0x2   /* ex.msg:8 */


#define SetOneSet 0x1   /* ex.msg:4 */
#define SetOnetwo 0x2   /* ex.msg:6 */
```

As can be seen, the various symbols given in the source file are mangled to generate unique identifiers and these identifiers get numbers assigned. Reading the source file and knowing about the rules will allow you to predict the content of the header file (it is deterministic), but this is not necessary. The `gencat` program can take care of everything. All the programmer has to do is put the generated header file in the dependency list of the source files of his project and add a rule to regenerate the header of any of the input files change.

One word about the symbol mangling. Every symbol consists of two parts: the name of the message set plus the name of the message or the special string `Set`. So `SetOnetwo` means this macro can be used to access the translation with identifier `two` in the message set `SetOne`.

The other names denote the names of the message sets. The special string `Set` is used in the place of the message identifier.

If in the code the second string of the set `SetOne` is used, the C code should look like this:

```
catgets (catdesc, SetOneSet, SetOnetwo,
         "   Message with ID \"two\", which gets the value 2 assigned")
```

Writing the function this way will allow you to change the message number and even the set number without any change in the C source code (the text of the string is normally not the same; this is only for this example).

11.1.4.3 Using Symbolic Version Numbers

To illustrate the usual way to work with the symbolic version numbers, here is a little example. Assume we want to write the very complex and famous greeting program. We start by writing the code as usual:

```
#include <stdio.h>
int
main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

Now we want to internationalize the message and therefore replace the message with whatever the user wants:

```
#include <nl_types.h>
#include <stdio.h>
#include "msgnrs.h"
int
main (void)
{
  nl_catd catdesc = catopen ("hello.cat", NL_CAT_LOCALE);
  printf (catgets (catdesc, SetMainSet, SetMainHello,
                   "Hello, world!\n"));
  catclose (catdesc);
  return 0;
}
```

We see how the catalog object is opened and the returned descriptor used in the other function calls. It is not really necessary to check for failure of any of the functions, since even in these situations, the functions will behave reasonably. They simply will be return a translation.

What remains unspecified here are the constants `SetMainSet` and `SetMainHello`. These are the symbolic names describing the message. To get the actual definitions that match the information in the catalog file, we have to create the message catalog source file and process it using the `gencat` program:

```
$ Messages for the famous greeting program.
$quote "

$set Main
```

```
Hello "Hallo, Welt!\n"
```

Now we can start building the program (assume the message catalog source file is named ‘hello.msg’ and the program source file ‘hello.c’):

```
% gencat -H msgnrs.h -o hello.cat hello.msg
% cat msgnrs.h
#define MainSet 0x1     /* hello.msg:4 */
#define MainHello 0x1   /* hello.msg:5 */
% gcc -o hello hello.c -I.
% cp hello.cat /usr/share/locale/de/LC_MESSAGES
% echo $LC_ALL
de
% ./hello
Hallo, Welt!
%
```

The call of the gencat program creates the missing header file ‘msgnrs.h’ as well as the message catalog binary. The former is used in the compilation of ‘hello.c’ while the latter is placed in a directory where the catopen function will try to locate it. Please check the LC_ALL environment variable and the default path for catopen presented in the description above.

11.2 The Uniforum Approach to Message Translation

Sun Microsystems tried to standardize a different approach to message translation in the Uniforum group. There never was a real standard defined, but the interface was used in Sun’s operation systems. Since this approach fits better in the development process of free software, it is also used throughout the GNU project and the GNU ‘gettext’ package provides support for this outside the GNU C Library.

The code of the ‘libintl’ from GNU ‘gettext’ is the same as the code in the GNU C Library. So the documentation in the GNU ‘gettext’ manual is also valid for the functionality here. The following text will describe the library functions in detail. But the numerous helper programs are not described in this manual. Instead, people should read the GNU ‘gettext’ manual.[1] We will only give a short overview.

Though the catgets functions are available by default on more systems, the gettext interface is at least as portable as the former. The GNU ‘gettext’ package can be used wherever the functions are not available.

[1] The GNU Project, *GNU gettext Utilities* (Free Software Foundation, May 6, 2003), http://www.gnu.org/software/gettext/manual/gettext.

11.2.1 The `gettext` Family of Functions

The paradigm underlying the `gettext` approach to message translation is different from that of the `catgets` functions, though the basic functionally is equivalent.

11.2.1.1 What Has to Be Done to Translate a Message?

The `gettext` functions have a very simple interface. The most basic function just takes the string that will be translated as the argument and returns the translation. This is fundamentally different from the `catgets` approach, where an extra key is necessary and the original string is only used for the error case.

If the string that has to be translated is the only argument, this of course means the string itself is the key—the translation will be selected based on the original string. The message catalogs must therefore contain the original strings plus one translation for any such string. The task of the `gettext` function is it to compare the argument string with the available strings in the catalog and return the appropriate translation. Of course this process is optimized so that it is not more expensive than an access using an atomic key like in `catgets`.

The `gettext` approach has some advantages but also some disadvantages. Please see the GNU '`gettext`' manual for a detailed discussion of the pros and cons.[2]

All the definitions and declarations for `gettext` can be found in the '`libintl.h`' header file. On systems where these functions are not part of the C library, they can be found in a separate library named '`libintl.a`' (or accordingly different for shared libraries).

`char *` **gettext** (`const char *`*msgid*) Function

The `gettext` function searches the currently selected message catalogs for a string that is equal to *msgid*. If there is such a string available, it is returned. Otherwise, the argument string *msgid* is returned.

Although the return value is `char *`, the returned string must not be changed. This broken type results from the history of the function and does not reflect the way the function should be used.

Above, we referred to "message catalogs" (plural). This is a specialty of the GNU implementation of these functions, and we will say more about this when we talk about the ways message catalogs are selected (see Section 11.2.1.2 [How to Determine Which Catalog to Use], page 328).

The `gettext` function does not modify the value of the global *errno* variable. This is necessary to make it possible to write something like:

```
printf (gettext ("Operation failed: %m\n"));
```

[2] Ibid.

Here the *errno* value is used in the `printf` function while processing the `%m` format element, and if the `gettext` function would change this value (it is called before `printf` is called), we would get a wrong message.

So there is no easy way to detect a missing message catalog besides comparing the argument string with the result. But it is normally the task of the user to react on missing catalogs. The program cannot guess when a message catalog is really necessary, since no translation is necessary for a user who speaks the language the program was developed in.

The remaining two functions to access the message catalog add some functionality to select a message catalog that is not the default one. This is important if parts of the program are developed independently. Every part can have its own message catalog and all of them can be used at the same time. The C library itself is an example; internally it uses the `gettext` functions, but since it must not depend on a currently selected default message catalog, it must specify all ambiguous information.

`char *` **dgettext** (`const char *`*domainname*`, const char *`*msgid*) Function

The `dgettext` function acts just like the `gettext` function. It only takes an additional first argument *domainname*, which guides the selection of the message catalogs that are searched for the translation. If the *domainname* parameter is the null pointer, the `dgettext` function is exactly equivalent to `gettext`, since the default value for the domain name is used.

As for `gettext`, the return value type is `char *`, which is an anachronism. The returned string must never be modified.

`char *` **dcgettext** (`const char *`*domainname*`, const char *`*msgid*`, int` *category*) Function

The `dcgettext` adds another argument to those that `dgettext` takes. This argument *category* specifies the last piece of information needed to localize the message catalog—the domain name and the locale category specify exactly which message catalog has to be used (relative to a given directory, see below).

The `dgettext` function can be expressed in terms of `dcgettext` by using:

```
dcgettext (domain, string, LC_MESSAGES)
```

instead of:

```
dgettext (domain, string)
```

This also shows which values are expected for the third parameter. You have to use the available selectors for the categories available in '`locale.h`'. Normally the available values are `LC_CTYPE`, `LC_COLLATE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC` and `LC_TIME`. `LC_ALL` must not be used, and even though the names might suggest this, there is no relation to the environment variables with the same names.

The `dcgettext` function is only implemented for compatibility with other systems that have `gettext` functions. There is not really any situation where

it is necessary (or useful) to use a value other than `LC_MESSAGES` for the *category* parameter. We are dealing with messages here, and any other choice can only be irritating.

As for `gettext`, the return value type is `char *`, which is an anachronism. The returned string must never be modified.

When using the three functions above in a program, it is a frequent case that the *msgid* argument is a constant string, so it is worthwhile to optimize this case. As long as no new message catalog is loaded, the translation of a message will not change. This optimization is actually implemented by the `gettext`, `dgettext` and `dcgettext` functions.

11.2.1.2 How to Determine Which Catalog to Use

The functions to retrieve the translations for a given message have a remarkably simple interface. But to provide the user of the program with the opportunity to select exactly the translation she wants and also to provide the programmer the ability to influence the way to locate the search for catalog files, there is a quite complicated underlying control mechanism. The code is complicated, the use is easy.

Basically, we have two different tasks to perform that can also be performed by the `catgets` functions:

1. Locate the set of message catalogs. There are a number of files for different languages and that all belong to the package. Usually they are all stored in the file system below a certain directory.

   There can be an arbitrary number of packages installed, and they can follow different guidelines for the placement of their files.
2. Relative to the location specified by the package, the actual translation files must be searched, based on the wishes of the user—for each language the user selects, the program should be able to locate the appropriate file.

This is the functionality required by the specifications for `gettext`, and this is also what the `catgets` functions are able to do. But there are some unresolved problems:

- The language to be used can be specified in several different ways. There is no generally accepted standard for this, and the user always expects the program to understand what he means. For example, to select the German translation, one could write `de`, `german` or `deutsch`, and the program should always react the same.
- Sometimes the user's specification is too detailed. If he for example, specifies `de_DE.ISO-8859-1`, which means German, spoken in Germany, coded using the ISO 8859-1 character set, there is the possibility that an exact message catalog match is not available. But there could be a catalog matching `de`, and if the character set used on the machine is always ISO 8859-1, there is no reason why this latter message catalog should not be used (we call this *message inheritance*).

- If a catalog for a desired language is not available, it is not always the second-best choice to fall back on the language of the developer and simply not translate any message. Instead, a user might be better able to read the messages in another language, and so the user of the program should be able to define a precedence order of languages.

We can divide the configuration actions into two parts: one is performed by the programmer and the other by the user. We will start with the functions the programmer can use since the user configuration will be based on this.

As the description of the functions in the previous sections already mentioned, separate sets of messages can be selected by a *domain name*. This is a simple string that should be unique for each program part that uses a separate domain. It is possible to use in one program an arbitrary number of domains at the same time. For example, the GNU C Library itself uses a domain named `libc` while the program using the C Library could use a domain named `foo`. The important point is that at any given time, exactly one domain is active. This is controlled with the following function:

`char *` **textdomain** (`const char *`*domainname*) Function

The `textdomain` function sets the default domain, which is used in all future `gettext` calls, to *domainname*. `dgettext` and `dcgettext` calls are not influenced if the *domainname* parameter of these functions is not the null pointer.

Before the first call to `textdomain`, the default domain is `messages`. This is the name specified in the specification of the `gettext` API. This name is as good as any other name. No program should ever really use a domain with this name, since this can only lead to problems.

The function returns the value that is from now on taken as the default domain. If the system ran out of memory, the returned value is `NULL` and the global variable *errno* is set to `ENOMEM`. Despite the return value type being `char *`, the return string must not be changed. It is allocated internally by the `textdomain` function.

If the *domainname* parameter is the null pointer, no new default domain is set. Instead, the currently selected default domain is returned.

If the *domainname* parameter is an empty string, the default domain is reset to its initial value—the domain with the name `messages`. Note, though, that the domain `messages` really never should be used.

`char *` **bindtextdomain** (`const char *`*domainname*`, const char *`*dirname*) Function

The `bindtextdomain` function can be used to specify the directory that contains the message catalogs for domain *domainname* for the different languages. This is the directory where the hierarchy of directories is expected. Details are explained below.

For the programmer, it is important to note that the translations that come with the program have be placed in a directory hierarchy starting at, say,

'/foo/bar'. Then the program should make a bindtextdomain call to bind the domain for the current program to this directory. So it is ensured that the catalogs will be found. A correctly running program does not depend on the user setting an environment variable.

The bindtextdomain function can be used several times and if the *domainname* argument is different, the previously bound domains will not be overwritten.

If a program that wishes to use bindtextdomain at some point uses the chdir function to change the current working directory, it is important that the *dirname* strings be an absolute pathname. Otherwise, the addressed directory might vary with the time.

If the *dirname* parameter is the null pointer, bindtextdomain returns the currently selected directory for the domain with the name *domainname*.

The bindtextdomain function returns a pointer to a string containing the name of the selected directory name. The string is allocated internally in the function and must not be changed by the user. If the system runs out of memory during the execution of bindtextdomain, the return value is NULL, and the global variable *errno* is set accordingly.

11.2.1.3 Additional Functions for More Complicated Situations

The functions of the gettext family described so far (and all the catgets functions as well) have one problem in the real world that has been neglected completely in all existing approaches—the handling of plural forms.

Looking through Unix source code written before the time anybody thought about internationalization (and, sadly, even afterwards), you can often find code similar to the following:

```
printf ("%d file%s deleted", n, n == 1 ? "" : "s");
```

After the first complaints from people internationalizing the code, people either completely avoided formulations like this, or used strings like "file(s)". Both look unnatural and should be avoided. First attempts to solve the problem correctly looked like this:

```
if (n == 1)
  printf ("%d file deleted", n);
else
  printf ("%d files deleted", n);
```

But this does not solve the problem. It helps languages where the plural form of a noun is not simply constructed by adding an 's', but that is all. Once again, people fell into the trap of believing the rules their language is using are universal. But the handling of plural forms differs widely between the language families (and even inside language families). There are two things we can differentiate between:

- The form for how plural forms are built differs. This is a problem with language that have many irregularities. German, for instance, is a drastic case. Though English and German are part of the same language family (Germanic),

the almost regular forming of plural noun forms (appending an 's') is hardly found in German.

- The number of plural forms differ. This is somewhat surprising for those who only have experiences with Romanic and Germanic languages since here the number is the same (there are two).

  Other language families have only one form or many forms.

The consequence of this is that application writers should not try to solve the problem in their code. This would be localization, since it is only usable for certain, hard-coded language environments. Instead, the extended `gettext` interface should be used.

These extra functions are taking two strings and a numerical argument instead of the one key string. The idea behind this is that using the numerical argument and the first string as a key, the implementation can select, using rules specified by the translator, the right plural form. The two string arguments then will be used to provide a return value in case no message catalog is found (similar to the normal `gettext` behavior). In this case, the rules for Germanic language are used, and it is assumed that the first string argument is the singular form, the second the plural form.

This has the consequence that programs without language catalogs can display the correct strings only if the program itself is written using a Germanic language. This is a limitation, but since the GNU C Library (as well as the GNU `gettext` package) are written as part of the GNU package and the coding standards for the GNU project require programs to be written in English, this solution nevertheless fulfills its purpose.

`char *` **ngettext** (`const char *`*msgid1*`, const char *`*msgid2*`, unsigned long int` *n*) Function

The `ngettext` function is similar to the `gettext` function, since it finds the message catalogs in the same way. But it takes two extra arguments. The *msgid1* parameter must contain the singular form of the string to be converted. It is also used as the key for the search in the catalog. The *msgid2* parameter is the plural form. The parameter *n* is used to determine the plural form. If no message catalog is found, *msgid1* is returned if `n == 1`, otherwise `msgid2` is returned.

An example for the us of this function is

```
printf (ngettext ("%d file removed", "%d files removed", n), n);
```

Please note that the numeric value *n* has to be passed to the `printf` function as well. It is not sufficient to pass it only to `ngettext`.

`char *` **dngettext** (`const char *`*domain*`, const char *`*msgid1*`, const char *`*msgid2*`, unsigned long int` *n*) Function

The `dngettext` is similar to the `dgettext` function in the way the message catalog is selected. The difference is that it takes two extra parameters to provide

the correct plural form. These two parameters are handled in the same way `ngettext` handles them.

`char *` **dcngettext** (`const char *`*domain*`, const char *`*msgid1*`, const char *`*msgid2*`, unsigned long int` *n*`, int` *category*) Function

The `dcngettext` is similar to the `dcgettext` function in the way the message catalog is selected. The difference is that it takes two extra parameter to provide the correct plural form. These two parameters are handled in the same way `ngettext` handles them.

The Problem of Plural Forms

A description of the problem can be found at the beginning of the last section. Now there is the question how to solve it. Without the input of linguists (which was not available), it was not possible to determine whether there are only a few different ways in which plurals are formed or whether the number can increase with every new supported language.

Therefore, the solution implemented is to allow the translator to specify the rules for how to select the plural form. Since the formula varies with every language, this is the only viable solution except for hard-coding the information in the code (which still would require the possibility of extensions to not prevent the use of new languages). The details are explained in the GNU `gettext` manual.[3] Here, only a a bit of information is provided.

The information about the plural form selection has to be stored in the header entry (the one with the empty `msgid` string). It looks like this:

```
Plural-Forms: nplurals=2; plural=n == 1 ? 0 : 1;
```

The `nplurals` value must be a decimal number that specifies how many different plural forms exist for this language. The string following `plural` is an expression that is using the C language syntax. Exceptions are that nonnegative numbers are allowed, numbers must be decimal and the only variable allowed is `n`. This expression will be evaluated whenever one of the functions `ngettext`, `dngettext` or `dcngettext` is called. The numeric value passed to these functions is then substituted for all uses of the variable `n` in the expression. The resulting value then must be greater than or equal to zero and smaller than the value given as the value of `nplurals`.

The following rules are known at this point. The languages with their families are listed. But this does not necessarily mean that the information can be generalized for the whole family (as can be easily seen in the table below).[4]

- *Only one form:*

[3] The GNU Project, *GNU gettext Utilities* (Free Software Foundation, May 6, 2003), http://www.gnu.org/software/gettext/manual/gettext.

[4] Additions are welcome. Send appropriate information to bug-glibc-manual@gnu.org.

  - Some languages require only one single form. There is no distinction between the singular and plural form. An appropriate header entry would look like this:

    ```
    Plural-Forms: nplurals=1; plural=0;
    ```

  - Languages with this property include:
    - Finno-Ugric family
      - Hungarian
    - Asian family
      - Japanese
      - Korean
    - Turkic/Altaic family
      - Turkish

- *Two forms, singular used for one only:*
  - This is the form used in most existing programs since it is what English is using. A header entry would look like this:

    ```
    Plural-Forms: nplurals=2; plural=n != 1;
    ```

    (Note: this uses the feature of C expressions that Boolean expressions have to value zero or one.)
  - Languages with this property include:
    - Germanic family
      - Danish
      - Dutch
      - English
      - German
      - Norwegian
      - Swedish
    - Finno-Ugric family
      - Estonian
      - Finnish
    - Latin/Greek family
      - Greek
    - Semitic family
      - Hebrew
    - Romance family
      - Italian
      - Portuguese
      - Spanish
    - Artificial

  - Esperanto
- *Two forms, singular used for zero and one:*
  - This is an exceptional case in the language family. The header entry would be

```
Plural-Forms: nplurals=2; plural=n>1;
```

  - Languages with this property include:
    - Romanic family
      - French
      - Brazilian
      - Portuguese
- *Three forms, special case for zero:*
  - The header entry would be

```
Plural-Forms: nplurals=3; \
plural=n%10==1 && n%100!=11 ? 0 : n != 0 ? 1 : 2;
```

  - Languages with this property include:
    - Baltic family
      - Latvian
- *Three forms, special cases for one and two:*
  - The header entry would be

```
Plural-Forms: nplurals=3; plural=n==1 ? 0 : n==2 ? 1 : 2;
```

  - Languages with this property include:
    - Celtic
      - Gaeilge (Irish)
- *Three forms, special case for numbers ending in 1[2-9]:*
  - The header entry would look like this:

```
Plural-Forms: nplurals=3; \
    plural=n%10==1 && n%100!=11 ? 0 : \
           n%10>=2 && (n%100<10 || n%100>=20) ? 1 : 2;
```

  - Languages with this property include:
    - Baltic family
      - Lithuanian
- *Three forms, special cases for numbers ending in 1 and 2, 3, 4, except those ending in 1[1-4]:*
  - The header entry would look like this:

```
Plural-Forms: nplurals=3; \
    plural=n%100/10==1 ? 2 : n%10==1 ? 0 : (n+9)%10>3 ? 2 : 1;
```

  - Languages with this property include:
    - Slavic family

  - Croatian
  - Czech
  - Russian
  - Ukrainian
- *Three forms, special cases for 1 and 2, 3, 4:*
  - The header entry would look like this:

```
Plural-Forms: nplurals=3; \
    plural=(n==1) ? 1 : (n>=2 && n<=4) ? 2 : 0;
```

  - Languages with this property include:
    - Slavic family
      - Slovak
- *Three forms, special case for one and some numbers ending in 2, 3, or 4:*
  - The header entry would look like this:

```
Plural-Forms: nplurals=3; \
  plural= n ==1 ? 0 : \
  n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2;
```

  - Languages with this property include:
    - Slavic family
      - Polish
- *Four forms, special case for one and all numbers ending in 02, 03, or 04:*
  - The header entry would look like this:

```
Plural-Forms: nplurals=4; \
  plural= n%100 == 1 ? 0 : n%100==2 ? 1 : n%100 == 3
        || n%100 == 4 ? 2 : 3;
```

  - Languages with this property include:
    - Slavic family
      - Slovenian

11.2.1.4 How to Specify the Output Character Set That `gettext` Uses

`gettext` not only looks up a translation in a message catalog, it also converts the translation on the fly to the desired output character set. This is useful if the user is working in a different character set than the translator who created the message catalog, because it avoids distributing variants of message catalogs that differ only in the character set.

The output character set is, by default, the value of `nl_langinfo (CODESET)`, which depends on the `LC_CTYPE` part of the current locale. But programs that store strings in a locale-independent way (e.g. UTF-8) can request that `gettext` and related functions return the translations in that encoding, by use of the `bind_textdomain_codeset` function.

The *msgid* argument to `gettext` is not subject to character-set conversion. Also, when `gettext` does not find a translation for *msgid*, it returns *msgid* unchanged—independently of the current output character set. It is therefore recommended that all *msgid*s be US-ASCII strings.

`char *` **bind_textdomain_codeset** (`const char *`*domainname*`, const char *`*codeset*) Function

The `bind_textdomain_codeset` function can be used to specify the output character set for message catalogs for domain *domainname*. The *codeset* argument must be a valid codeset name that can be used for the `iconv_open` function, or a null pointer.

If the *codeset* parameter is the null pointer, `bind_textdomain_codeset` returns the currently selected codeset for the domain with the name *domainname*. It returns `NULL` if no codeset has yet been selected.

The `bind_textdomain_codeset` function can be used several times. If used multiple times with the same *domainname* argument, the later call overrides the settings made by the earlier one.

The `bind_textdomain_codeset` function returns a pointer to a string containing the name of the selected codeset. The string is allocated internally in the function and must not be changed by the user. If the system runs out of memory during the execution of `bind_textdomain_codeset`, the return value is `NULL` and the global variable *errno* is set accordingly.

11.2.1.5 How to Use `gettext` in GUI Programs

One place where the `gettext` functions, if used normally, have big problems is within programs with graphical user interfaces (GUIs). The problem is that many of the strings that have to be translated are very short. They have to appear in pull-down menus, which restricts their length. But strings that do not contain entire sentences or at least large fragments of a sentence may appear in more than one situation in the program with different translations. This is especially true for the one-word strings that are frequently used in GUI programs.

As a consequence, many people say that the `gettext` approach is wrong and that instead `catgets`, which indeed does not have this problem, should be used. But there is a very simple and powerful method to handle this problem with the `gettext` functions.

As as example, consider the following fictional situation. A GUI program has a menu bar with the following entries:

```
+------------+------------+--------------------------------------+
| File       | Printer    |                                      |
+------------+------------+--------------------------------------+
| Open     | | Select   |
| New      | | Open     |
+----------+ | Connect  |
             +----------+
```

To have the strings `File`, `Printer`, `Open`, `New`, `Select` and `Connect` translated, there has to be at some point in the code a call to a function of the `gettext` family. But in two places the string passed into the function would be `Open`. The translations might not be the same and therefore we are in the dilemma described above.

One solution to this problem is to artificially lengthen the strings to make them unambiguous. But what would the program do if no translation were available? The lengthened string is not what should be printed. So we should use a modified version of the functions.

To lengthen the strings, a uniform method should be used. In the example above, the strings could be chosen as:

```
Menu|File
Menu|Printer
Menu|File|Open
Menu|File|New
Menu|Printer|Select
Menu|Printer|Open
Menu|Printer|Connect
```

Now all the strings are different, and if now instead of `gettext` the following little wrapper function is used, everything works just fine:

```
char *
sgettext (const char *msgid)
{
  char *msgval = gettext (msgid);
  if (msgval == msgid)
    msgval = strrchr (msgid, '|') + 1;
  return msgval;
}
```

What this little function does is to recognize the case when no translation is available. This can be done very efficiently by a pointer comparison since the return value is the input value. If there is no translation, we know that the input string is in the format we used for the Menu entries and therefore contains a '`|`' character. We simply search for the last occurrence of this character and return a pointer to the character following it. That's it!

If one now consistently uses the lengthened-string form and replaces the `gettext` calls with calls to `sgettext` (this is normally limited to very few places in the GUI implementation), then it is possible to produce a program that can be internationalized.

With advanced compilers (such as GNU C), one can write the `sgettext` functions as an in-line function or as a macro like this:

```
#define sgettext(msgid) \
  ({ const char *__msgid = (msgid);            \
     char *__msgstr = gettext (__msgid);       \
```

```
if (__msgval == __msgid)                    \
  __msgval = strrchr (__msgid, '|') + 1;    \
__msgval; })
```

The other `gettext` functions (`dgettext`, `dcgettext` and the `ngettext` equivalents) can and should have corresponding functions as well that look almost identical, except for the parameters and the call to the underlying function.

Now, why do such functions not exist in the GNU C Library? There are two parts to the answer for this question:

- They are easy to write and therefore can be provided by the project they are used in. This is not an answer by itself and must be seen together with the second part, which is
- There is no way the C library can contain a version that can work everywhere. The problem is the selection of the character to separate the prefix from the actual string in the lenghtened string. The examples above used '`|`', which is a good choice because it resembles a notation frequently used in this context and it also is a character not often used in message strings. But, what if the character *is* used in message strings? Or what if the chosen character is not available in the character set on the machine one compiles on (e.g., '`|`' is not required to exist for ISO C; this is why the '`iso646.h`' file exists in ISO C programming environments)? So, even this method is not perfect.

There is only one more comment left to make. The wrapper function above requires that the translation strings themselves not be lengthened. This is only logical. There is no need to disambiguate the strings (since they are never used as keys for a search), and one also saves some memory and disk space by doing this.

11.2.1.6 User Influence on `gettext`

The last sections described what the programmer can do to internationalize the messages of the program. But it is up to the user in the end to select the message she wants to see. She must understand them.

The POSIX locale model uses the environment variables `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `NUMERIC` and `LC_TIME` to select the locale that is to be used. This way the user can influence lots of functions. As we mentioned above, the `gettext` functions also take advantage of this.

To understand how this happens, it is necessary to take a look at the various components of the file name that get computed to locate a message catalog. It is composed as follows:

dir_name`/`*locale*`/LC_`*category*`/`*domain_name*`.mo`

The default value for *dir_name* is system specific. It is computed from the value given as the prefix while configuring the C library. This value normally is '`/usr`' or '`/`'. For the former, the complete *dir_name* is

```
/usr/share/locale
```

We can use '`/usr/share`' since the '`.mo`' files containing the message catalogs are system independent, so all systems can use the same files. If the program

executed the `bindtextdomain` function for the message domain that is currently handled, the `dir_name` component is exactly the value that was given to the function as the second parameter. `bindtextdomain` allows overwriting the only system-dependent and fixed value to make it possible to address files anywhere in the file system.

The *category* is the name of the locale category that was selected in the program code. For `gettext` and `dgettext`, this is always `LC_MESSAGES`; for `dcgettext`, this is selected by the value of the third parameter. As said above, using a category other than `LC_MESSAGES` should be avoided.

The *locale* component is computed based on the category used. Just like for the `setlocale` function, user selection comes into play. Some environment variables are examined in a fixed order, and the first environment variable set determines the return value of the lookup process. In detail, for the category `LC_xxx` the following variables are examined in this order:

- `LANGUAGE`
- `LC_ALL`
- `LC_xxx`
- `LANG`

This looks very familiar. With the exception of the `LANGUAGE` environment variable, this is exactly the lookup order the `setlocale` function uses. But why introduce the `LANGUAGE` variable?

The reason is that the syntax of the values these variables can have is different from what is expected by the `setlocale` function. If we would set `LC_ALL` to a value following the extended syntax, that would mean the `setlocale` function would never be able to use the value of this variable. An additional variable removes this problem, plus we can select the language independently of the locale setting, which is sometimes useful.

While for the `LC_xxx` variables, the value should consist of exactly one specification of a locale, the `LANGUAGE` variable's value can consist of a colon-separated list of locale names. This is the way we manage to implement one of our additional demands above, to be able to specify an ordered list of languages.

Back to the constructed file name. We have only one component missing. The *domain_name* part is the name that was either registered using the `textdomain` function or that was given to `dgettext` or `dcgettext` as the first parameter. Now it becomes obvious that a good choice for the domain name in the program code is a string that is closely related to the program or package name. For example, the domain name for the GNU C Library is `libc`.

A limited piece of example code should show how the programmer is supposed to work:

```
{
  setlocale (LC_ALL, "");
  textdomain ("test-package");
  bindtextdomain ("test-package", "/usr/local/share/locale");
```

```
  puts (gettext ("Hello, world!"));
}
```

At the program start, the default domain is `messages`, and the default locale is '`C`'. The `setlocale` call sets the locale according to the user's environment variables; remember that correct functioning of `gettext` relies on the correct setting of the `LC_MESSAGES` locale (for looking up the message catalog) and of the `LC_CTYPE` locale (for the character-set conversion). The `textdomain` call changes the default domain to `test-package`. The `bindtextdomain` call specifies that the message catalogs for the domain `test-package` can be found below the directory '`/usr/local/share/locale`'.

If now the user set in her environment the variable `LANGUAGE` to `de`, the `gettext` function will try to use the translations from the file:

```
/usr/local/share/locale/de/LC_MESSAGES/test-package.mo
```

From the above descriptions, it should be clear which component of this file name is determined by which source.

In the above example, we assumed that the `LANGUAGE` environment variable was set to `de`. This might be an appropriate selection, but what happens if the user wants to use `LC_ALL` because of the wider usability, and here the required value is `de_DE.ISO-8859-1`? We already mentioned above that a situation like this is not infrequent. For example, a person might prefer reading a dialect and, if this is not available, falling back on the standard language.

The `gettext` functions know about situations like this and can handle them gracefully. The functions recognize the format of the value of the environment variable. It can split the value into different pieces and, by leaving out one or the other part, it can construct new values. This happens of course in a predictable way. To understand this, one must know the format of the environment variable value. There is one more or less standardized form, originally from the X/Open specification:

```
language[_territory[.codeset]][@modifier]
```

Less specific locale names will be stripped off in the order of the following list:

1. `codeset`
2. `normalized codeset`
3. `territory`
4. `modifier`

The `language` field will never be dropped for obvious reasons.

The only new thing is the `normalized codeset` entry. This is another goodie, which is introduced to help reduce the chaos that derives from the inability of people to standardize the names of character sets. Instead of ISO-8859-1, one can often see 8859-1, 88591, iso8859-1 or iso_8859-1. The `normalized codeset` value is generated from the user-provided character-set name by applying the following rules:

1. Remove all characters besides numbers and letters.
2. Fold letters to lowercase.

3. If the name only contains digits, prepend the string ‘iso’.

So all of the above names will be normalized to iso88591. This allows the program user to more freely choose the locale name.

Even this extended functionality still does not help solve the problem that completely different names can be used to denote the same locale (e.g., de and german). To be of help in this situation, the locale implementation and the gettext functions know about aliases.

The file ‘/usr/share/locale/locale.alias’ (replace ‘/usr’ with whatever prefix you used for configuring the C library) contains a mapping of alternative names to more regular names. The system manager is free to add new entries to fill his own needs. The selected locale from the environment is compared with the entries in the first column of this file, ignoring the case. If they match, the value of the second column is used instead for further handling.

In the description of the format of the environment variables we already mentioned the character set as a factor in the selection of the message catalog. In fact, only catalogs that contain text written using the character set of the system or program can be used (directly; a solution for this will come some day). The user will always have to be careful about this. Also, if in the collection of message catalogs there are files for the same language that are coded using different character sets, the user has to be careful.

11.2.2 Programs to Handle Message Catalogs for `gettext`

The GNU C Library does not contain the source code for the programs to handle message catalogs for the gettext functions. As part of the GNU project, the GNU gettext package contains everything the developer needs. The functionality provided by the tools in this package by far exceeds the abilities of the gencat program described above for the catgets functions.

There is a program msgfmt, which is equivalent to the gencat program. It generates from the human-readable and human-editable form of the message catalog a binary file that can be used by the gettext functions. But there are several more programs available.

The xgettext program can be used to automatically extract the translatable messages from a source file, so the programmer need not be careful of the translations and the list of messages that have to be translated. She will simply wrap the translatable string in calls to gettext and the others, and the rest will be done by xgettext. This program has a lot of options that help customize the output or help understand the input.

Other programs help to manage the development cycle when new messages appear in the source files or when a new translation of the messages appear. Here it should only be noted that using all the tools in GNU gettext, it is possible to *completely* automatize the handling of message catalogs. Besides marking the translatable string in the source code and generating the translations, the developers themselves do not have to do anything.

12 Searching and Sorting

This chapter describes functions for searching and sorting arrays of arbitrary objects. You pass the appropriate comparison function to be applied as an argument, along with the size of the objects in the array and the total number of elements.

12.1 Defining the Comparison Function

In order to use the sorted array library functions, you have to describe how to compare the elements of the array.

To do this, you supply a comparison function to compare two elements of the array. The library will call this function, passing as arguments pointers to two array elements to be compared. Your comparison function should return a value the way `strcmp` (see Section 5.5 [String/Array Comparison], page 105) does: negative if the first argument is "less" than the second, zero if they are "equal" and positive if the first argument is "greater".

Here is an example of a comparison function that works with an array of numbers of type `double`:

```
int
compare_doubles (const void *a, const void *b)
{
  const double *da = (const double *) a;
  const double *db = (const double *) b;

  return (*da > *db) - (*da < *db);
}
```

The header file '`stdlib.h`' defines a name for the data type of comparison functions. This type is a GNU extension.

```
int comparison_fn_t (const void *, const void *);
```

12.2 Array Search Function

Generally, searching for a specific element in an array means that potentially all elements must be checked. The GNU C Library contains functions to perform linear searches. The prototypes for the following two functions can be found in '`search.h`'.

`void *` **lfind** (`const void *`*key*`, void *`*base*`, size_t *`*nmemb*`, size_t` *size*`, comparison_fn_t` *compar*) Function

The `lfind` function searches in the array with `*`*nmemb* elements of *size* bytes pointed to by *base* for an element that matches the one pointed to by *key*. The function pointed to by *compar* is used to decide whether two elements match.

The return value is a pointer to the matching element in the array starting at *base* if it is found. If no matching element is available, `NULL` is returned.

The mean run time of this function is **nmemb*/2. This function should only be used if elements often get added to or deleted from the array, in which case it might not be useful to sort the array before searching.

`void *` **lsearch** (`const void *`*key*`, void *`*base*`, size_t *`*nmemb*`, size_t` *size*`, comparison_fn_t` *compar*) Function

The `lsearch` function is similar to the `lfind` function. It searches the given array for an element and returns it if found. The difference is that if no matching element is found, the `lsearch` function adds the object pointed to by *key* (with a size of *size* bytes) at the end of the array and it increments the value of **nmemb* to reflect this addition.

This means that if the caller is not sure that the array contains the element you are searching for, the memory allocated for the array starting at *base* must have room for at least *size* more bytes. If you are sure the element is in the array, it is better to use `lfind`, so having more room in the array is always necessary when calling `lsearch`.

To search a sorted array for an element matching the key, use the `bsearch` function. The prototype for this function is in the header file '`stdlib.h`'.

`void *` **bsearch** (`const void *`*key*`, const void *`*array*`, size_t` *count*`, size_t` *size*`, comparison_fn_t` *compare*) Function

The `bsearch` function searches the sorted array *array* for an object that is equivalent to *key*. The array contains *count* elements, each of which is of size *size* bytes.

The *compare* function is used to perform the comparison. This function is called with two pointer arguments and should return an integer less than, equal to or greater than zero corresponding to whether its first argument is considered less than, equal to or greater than its second argument. The elements of the *array* must already be sorted in ascending order according to this comparison function.

The return value is a pointer to the matching array element, or a null pointer if no match is found. If the array contains more than one element that matches, the one that is returned is unspecified.

This function derives its name from the fact that it is implemented using the binary search algorithm.

12.3 Array Sort Function

To sort an array using an arbitrary comparison function, use the `qsort` function. The prototype for this function is in '`stdlib.h`'.

`void` **qsort** (`void *`*array*`, size_t` *count*`, size_t` *size*`, comparison_fn_t` *compare*) Function

The *qsort* function sorts the array *array*. The array contains *count* elements, each of which is of size *size*.

The *compare* function is used to perform the comparison on the array elements. This function is called with two pointer arguments and should return an integer less than, equal to or greater than zero corresponding to whether its first argument is considered less than, equal to or greater than its second argument.

Warning: If two objects compare as equal, their order after sorting is unpredictable. That is to say, the sorting is not stable. This can make a difference when the comparison considers only part of the elements. Two elements with the same sort key may differ in other respects.

If you want the effect of a stable sort, you can get this result by writing the comparison function so that, lacking other reasons to distinguish between two elements, it compares them by their addresses. Doing this may make the sorting algorithm less efficient, so do it only if necessary.

Here is a simple example of sorting an array of doubles in numerical order, using the comparison function defined above (see Section 12.1 [Defining the Comparison Function], page 343):

```
{
  double *array;
  int size;
  ...
  qsort (array, size, sizeof (double), compare_doubles);
}
```

The `qsort` function derives its name from the fact that it was originally implemented using the "quick sort" algorithm.

The implementation of `qsort` in this library might not be an in-place sort and might thereby use an extra amount of memory to store the array.

12.4 Searching and Sorting Example

Here is an example showing the use of `qsort` and `bsearch` with an array of structures. The objects in the array are sorted by comparing their `name` fields with the `strcmp` function. Then, we can look up individual objects based on their names.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Define an array of critters to sort. */

struct critter
  {
    const char *name;
    const char *species;
  };
```

```
struct critter muppets[] =
  {
    {"Kermit", "frog"},
    {"Piggy", "pig"},
    {"Gonzo", "whatever"},
    {"Fozzie", "bear"},
    {"Sam", "eagle"},
    {"Robin", "frog"},
    {"Animal", "animal"},
    {"Camilla", "chicken"},
    {"Sweetums", "monster"},
    {"Dr. Strangepork", "pig"},
    {"Link Hogthrob", "pig"},
    {"Zoot", "human"},
    {"Dr. Bunsen Honeydew", "human"},
    {"Beaker", "human"},
    {"Swedish Chef", "human"}
  };

int count = sizeof (muppets) / sizeof (struct critter);



/* This is the comparison function used for sorting and searching. */

int
critter_cmp (const struct critter *c1, const struct critter *c2)
{
  return strcmp (c1->name, c2->name);
}


/* Print information about a critter. */

void
print_critter (const struct critter *c)
{
  printf ("%s, the %s\n", c->name, c->species);
}



/* Do the lookup into the sorted array. */
```

```
void
find_critter (const char *name)
{
  struct critter target, *result;
  target.name = name;
  result = bsearch (&target, muppets, count, sizeof (struct critter),
                    critter_cmp);
  if (result)
    print_critter (result);
  else
    printf ("Couldn't find %s.\n", name);
}


/* Main program. */

int
main (void)
{
  int i;

  for (i = 0; i < count; i++)
    print_critter (&muppets[i]);
  printf ("\n");

  qsort (muppets, count, sizeof (struct critter), critter_cmp);

  for (i = 0; i < count; i++)
    print_critter (&muppets[i]);
  printf ("\n");

  find_critter ("Kermit");
  find_critter ("Gonzo");
  find_critter ("Janice");

  return 0;
}
```

The output from this program looks like:

```
Kermit, the frog
Piggy, the pig
Gonzo, the whatever
Fozzie, the bear
Sam, the eagle
```

```
Robin, the frog
Animal, the animal
Camilla, the chicken
Sweetums, the monster
Dr. Strangepork, the pig
Link Hogthrob, the pig
Zoot, the human
Dr. Bunsen Honeydew, the human
Beaker, the human
Swedish Chef, the human

Animal, the animal
Beaker, the human
Camilla, the chicken
Dr. Bunsen Honeydew, the human
Dr. Strangepork, the pig
Fozzie, the bear
Gonzo, the whatever
Kermit, the frog
Link Hogthrob, the pig
Piggy, the pig
Robin, the frog
Sam, the eagle
Swedish Chef, the human
Sweetums, the monster
Zoot, the human

Kermit, the frog
Gonzo, the whatever
Couldn't find Janice.
```

12.5 The `hsearch` Function

The functions mentioned so far in this chapter are for searching in a sorted or an unsorted array. There are functions available that can organize information to make searching easier. The costs of insert, delete and search differ. One possible implementation is using hashing tables. The following functions are declared in the the header file '`search.h`'.

`int` **hcreate** (`size_t` *nel*) Function

The `hcreate` function creates a hashing table that can contain at least *nel* elements. There is no way to grow this table, so it is necessary to choose the value for *nel* wisely. The methods used to implement this function might make it necessary to make the number of elements in the hashing table larger than the

expected maximum number of elements. Hashing tables usually work inefficiently if they are filled 80% or more. The constant access time guaranteed by hashing can only be achieved if few collisions exist.[1]

The weakest aspect of this function is that there can be at most one hashing table used through the whole program. The table is allocated in local memory out of control of the programmer. As an extension, the GNU C Library provides an additional set of functions with a reentrant interface that provide a similar interface but that allow you to keep an arbitrary number of hashing tables.

It is possible to use more than one hashing table in the program run if the former table is first destroyed by a call to `hdestroy`.

The function returns a nonzero value if successful. If it returns zero, something went wrong. This could either mean there was already a hashing table in use, or the program has run out of memory.

`void` **hdestroy** (`void`) Function

The `hdestroy` function can be used to free all the resources allocated in a previous call of `hcreate`. After a call to this function, it is again possible to call `hcreate` and allocate a new table with a potentially different size.

It is important to remember that the elements contained in the hashing table at the time `hdestroy` is called are *not* freed by this function. It is the responsibility of the program code to free those strings (if necessary at all). Freeing all the element memory is not possible without extra, separately kept information, since there is no function to iterate through all available elements in the hashing table. If it is really necessary to free a table and all elements, the programmer has to keep a list of all table elements, and before calling `hdestroy` he has to free all element data using this list. This is a very unpleasant mechanism and it also shows that this kind of hashing table is mainly meant for tables that are created once and used until the end of the program run.

Entries of the hashing table and keys for the search are defined using this type:

struct ENTRY Data type

Both elements of this structure are pointers to zero-terminated strings. This is a limiting restriction of the functionality of the `hsearch` functions. They can only be used for data sets that use the NUL character always and solely to terminate the records. It is not possible to handle general binary data.

`char *key`

This is a pointer to a zero-terminated string of characters describing the key for the search or the element in the hashing table.

`char *data`

This is a pointer to a zero-terminated string of characters describing the data. If the functions will be called only for searching an

[1] See Donald E. Knuth, *The Art of Computer Programming,* vol. 3, *Sorting and Searching*, 2nd ed. (Reading, MA: Addison-Wesley, April 24, 1998).

existing entry, this element might stay undefined, since it is not used.

`ENTRY *` **hsearch** (`ENTRY` *item*, `ACTION` *action*) Function

To search in a hashing table created using `hcreate`, the `hsearch` function must be used. This function can perform a simple search for an element (if *action* has the `FIND`) or alternatively it can insert the key element into the hashing table. Entries are never replaced.

The key is denoted by a pointer to an object of type `ENTRY`. For locating the corresponding position in the hashing table, only the `key` element of the structure is used.

If an entry with matching key is found, the *action* parameter is irrelevant. The found entry is returned. If no matching entry is found and the *action* parameter has the value `FIND`, the function returns a `NULL` pointer. If no entry is found and the *action* parameter has the value `ENTER`, a new entry is added to the hashing table that is initialized with the parameter *item*. A pointer to the newly added entry is returned.

As mentioned before, the hashing table used by the functions described so far is global and there can be at any time at most one hashing table in the program. A solution is to use the following functions, which are a GNU extension. All have in common that they operate on a hashing table that is described by the content of an object of the type `struct hsearch_data`. This type should be treated as opaque; none of its members should be changed directly.

`int` **hcreate_r** (`size_t` *nel*, `struct hsearch_data *`*htab*) Function

The `hcreate_r` function initializes the object pointed to by *htab* to contain a hashing table with at least *nel* elements. This function is equivalent to the `hcreate` function except that the initialized data structure is controlled by the user.

This allows having more than one hashing table at one time. The memory necessary for the `struct hsearch_data` object can be allocated dynamically. It must be initialized with zero before calling this function.

The return value is nonzero if the operation was successful. if the return value is zero, something went wrong, which probably means the program runs out of memory.

`void` **hdestroy_r** (`struct hsearch_data *`*htab*) Function

The `hdestroy_r` function frees all resources allocated by the `hcreate_r` function for this very same object *htab*. As for `hdestroy`, it is the program's responsibility to free the strings for the elements of the table.

`int` **hsearch_r** (`ENTRY` *item*, `ACTION` *action*, `ENTRY` `**`*retval*, `struct hsearch_data *`*htab*) Function

The `hsearch_r` function is equivalent to `hsearch`. The meaning of the first two arguments is identical. But instead of operating on a single global hashing

table the function works on the table described by the object pointed to by *htab* (which is initialized by a call to `hcreate_r`).

Another difference with `hcreate` is that the pointer to the found entry in the table is not the return value of the functions. It is returned by storing it in a pointer variable pointed to by the *retval* parameter. The return value of the function is an integer value indicating success if it is nonzero and failure if it is zero. In the latter case, the global variable *errno* signals the reason for the failure.

`ENOMEM` The table is filled, `hsearch_r` was called with a so far unknown key and *action* was set to `ENTER`.

`ESRCH` The *action* parameter is `FIND`, and no corresponding element is found in the table.

12.6 The `tsearch` Function

Another common way to organize data for efficient search is to use trees. The `tsearch` function family provides a nice interface to functions to organize possibly large amounts of data by providing a mean access time proportional to the logarithm of the number of elements. The GNU C Library implementation guarantees that this bound is never exceeded even for input data that cause problems for simple binary tree implementations.

The functions described in this chapter are all described in the System V and X/Open specifications and are therefore quite portable.

In contrast to the `hsearch` functions, the `tsearch` functions can be used with arbitrary data and not just zero-terminated strings.

The `tsearch` functions have the advantage that no function to initialize data structures is necessary. A simple pointer of type `void *` initialized to `NULL` is a valid tree and can be extended or searched. The prototypes for these functions can be found in the header file '`search.h`'.

`void *` **tsearch** (`const void *`*key*`, void **`*rootp*`,` `comparison_fn_t` *compar*) Function

The `tsearch` function searches in the tree pointed to by **rootp* for an element matching *key*. The function pointed to by *compar* is used to determine whether two elements match. See Section 12.1 [Defining the Comparison Function], page 343, for a specification of the functions that can be used for the *compar* parameter.

If the tree does not contain a matching entry, the *key* value will be added to the tree. `tsearch` does not make a copy of the object pointed to by *key* (how could it, since the size is unknown). Instead, it adds a reference to this object, which means the object must be available as long as the tree data structure is used.

The tree is represented by a pointer to a pointer, since it is sometimes necessary to change the root node of the tree. It must not be assumed that the variable

pointed to by *rootp* has the same value after the call. This also shows that it is not safe to call the `tsearch` function more than once at the same time using the same tree. It is no problem to run it more than once at a time on different trees.

The return value is a pointer to the matching element in the tree. If a new element was created, the pointer points to the new data (which is in fact *key*). If an entry had to be created and the program ran out of space, `NULL` is returned.

`void *` **tfind** (`const void *`*key*`, void *const *`*rootp*`, comparison_fn_t` *compar*) Function

The `tfind` function is similar to the `tsearch` function. It locates an element matching the one pointed to by *key* and returns a pointer to this element. But if no matching element is available, no new element is entered (note that the *rootp* parameter points to a constant pointer). Instead the function returns `NULL`.

Another advantage of the `tsearch` function in contrast to the `hsearch` functions is that there is an easy way to remove elements.

`void *` **tdelete** (`const void *`*key*`, void **`*rootp*`, comparison_fn_t` *compar*) Function

To remove a specific element matching *key* from the tree, `tdelete` can be used. It locates the matching element using the same method as `tfind`. The corresponding element is then removed and a pointer to the parent of the deleted node is returned by the function. If there is no matching entry in the tree, nothing can be deleted and the function returns `NULL`. If the root of the tree is deleted, `tdelete` returns some unspecified value not equal to `NULL`.

`void` **tdestroy** (`void *`*vroot*`, __free_fn_t` *freefct*) Function

If the complete search tree has to be removed, one can use `tdestroy`. It frees all resources allocated by the `tsearch` function to generate the tree pointed to by *vroot*.

For the data in each tree node, the function *freefct* is called. The pointer to the data is passed as the argument to the function. If no such work is necessary, *freefct* must point to a function doing nothing. It is called in any case.

This function is a GNU extension and is not covered by the System V or X/Open specifications.

In addition to the function to create and destroy the tree data structure, there is another function that allows you to apply a function to all elements of the tree. The function must have this type:

```
void __action_fn_t (const void *nodep, VISIT value, int level);
```

The *nodep* is the data value of the current node (once given as the *key* argument to `tsearch`). *level* is a numeric value that corresponds to the depth of the current node in the tree. The root node has the depth 0, its children have a depth of 1, and so on. The `VISIT` type is an enumeration type.

VISIT Data Type

The `VISIT` value indicates the status of the current node in the tree and how the function is called. The status of a node is either 'leaf' or 'internal node'. For each leaf node the function is called exactly once; for each internal node it is called three times: before the first child is processed, after the first child is processed and after both children are processed. This makes it possible to handle all three methods of tree traversal (or even a combination of them).

`preorder`
: The current node is an internal node and the function is called before the first child was processed.

`postorder`
: The current node is an internal node and the function is called after the first child was processed.

`endorder`
: The current node is an internal node and the function is called after the second child was processed.

`leaf`
: The current node is a leaf.

`void` **twalk** (`const void *`*root*, `__action_fn_t` *action*) Function

For each node in the tree with a node pointed to by *root*, the `twalk` function calls the function provided by the parameter *action*. For leaf nodes, the function is called exactly once with *value* set to `leaf`. For internal nodes, the function is called three times, setting the *value* parameter or *action* to the appropriate value. The *level* argument for the *action* function is computed while descending the tree with increasing the value by one for the descend to a child, starting with the value 0 for the root node.

Since the functions used for the *action* parameter to `twalk` must not modify the tree data, it is safe to run `twalk` in more than one thread at the same time, working on the same tree. It is also safe to call `tfind` in parallel. Functions that modify the tree must not be used, otherwise the behavior is undefined.

13 Pattern Matching

The GNU C Library provides pattern-matching facilities for two kinds of patterns: regular expressions and file-name wildcards. The library also provides a facility for expanding variable and command references and parsing text into words in the way the shell does.

13.1 Wildcard Matching

This section describes how to match a wildcard pattern against a particular string. The result is a yes or no answer—does the string fit the pattern or not. The symbols described here are all declared in '`fnmatch.h`'.

`int` **fnmatch** (`const char *`*pattern*`, const char *`*string*`, int` *flags*) Function

This function tests whether the string *string* matches the pattern *pattern*. It returns `0` if they do match; otherwise, it returns the nonzero value `FNM_NOMATCH`. The arguments *pattern* and *string* are both strings.

The argument *flags* is a combination of flag bits that alter the details of matching. See below for a list of the defined flags.

In the GNU C Library, `fnmatch` cannot experience an "error"—it always returns an answer for whether the match succeeds. However, other implementations of `fnmatch` might sometimes report "errors". They would do so by returning nonzero values that are not equal to `FNM_NOMATCH`.

These are the available flags for the *flags* argument:

`FNM_FILE_NAME`
: Treat the '`/`' character specially, for matching file names. If this flag is set, wildcard constructs in *pattern* cannot match '`/`' in *string*. Thus, the only way to match '`/`' is with an explicit '`/`' in *pattern*.

`FNM_PATHNAME`
: This is an alias for `FNM_FILE_NAME`; it comes from POSIX.2. We don't recommend this name because we don't use the term "pathname" for file names.

`FNM_PERIOD`
: Treat the '`.`' character specially if it appears at the beginning of *string*. If this flag is set, wildcard constructs in *pattern* cannot match '`.`' as the first character of *string*.

  If you set both `FNM_PERIOD` and `FNM_FILE_NAME`, then the special treatment applies to '`.`' following '`/`' as well as to '`.`' at the beginning of *string*. (The shell uses the `FNM_PERIOD` and `FNM_FILE_NAME` flags together for matching file names.)

`FNM_NOESCAPE`
: Don't treat the '`\`' character specially in patterns. Normally, '`\`' quotes the following character, turning off its special meaning (if any) so that it matches only itself. When quoting is enabled, the pattern '`\?`' matches only the string '`?`', because the question mark in the pattern acts like an ordinary character.

  If you use `FNM_NOESCAPE`, then '`\`' is an ordinary character.

`FNM_LEADING_DIR`
: Ignore a trailing sequence of characters starting with a '`/`' in *string*; that is to say, test whether *string* starts with a directory name that *pattern* matches.

  If this flag is set, either '`foo*`' or '`foobar`' as a pattern would match the string '`foobar/frobozz`'.

`FNM_CASEFOLD`
: Ignore case in comparing *string* to *pattern*.

`FNM_EXTMATCH`
: Recognize besides the normal patterns also the extended patterns introduced in '`ksh`'. The patterns are written in the form explained in the following table where *pattern-list* is a '`|`'-separated list of patterns.

  `?(`*pattern-list*`)`
  : The pattern matches if zero or one occurrences of any of the patterns in the *pattern-list* allow matching the input string.

  `*(`*pattern-list*`)`
  : The pattern matches if zero or more occurrences of any of the patterns in the *pattern-list* allow matching the input string.

  `+(`*pattern-list*`)`
  : The pattern matches if one or more occurrences of any of the patterns in the *pattern-list* allow matching the input string.

  `@(`*pattern-list*`)`
  : The pattern matches if exactly one occurrence of any of the patterns in the *pattern-list* allows matching the input string.

  `!(`*pattern-list*`)`
  : The pattern matches if the input string cannot be matched with any of the patterns in the *pattern-list*.

13.2 Globbing

The archetypal use of wildcards is for matching against the files in a directory, and making a list of all the matches. This is called *globbing*.

You could do this using `fnmatch`, by reading the directory entries one by one and testing each one with `fnmatch`. But that would be slow (and complex, since you would have to handle subdirectories by hand).

The library provides a function `glob` to make this particular use of wildcards convenient. `glob` and the other symbols in this section are declared in '`glob.h`'.

13.2.1 Calling `glob`

The result of globbing is a vector of file names (strings). To return this vector, `glob` uses a special data type, `glob_t`, which is a structure. You pass `glob` the address of the structure, and it fills in the structure's fields to tell you about the results.

glob_t Data Type

This data type holds a pointer to a word vector. More precisely, it records both the address of the word vector and its size. The GNU implementation contains some more fields that are nonstandard extensions.

`gl_pathc`
: The number of elements in the vector, excluding the initial null entries if the GLOB_DOOFFS flag is used (see `gl_offs` below).

`gl_pathv`
: The address of the vector; this field has type `char **`.

`gl_offs`
: The offset of the first real element of the vector, from its nominal address in the `gl_pathv` field; unlike the other fields, this is always an input to `glob`, rather than an output from it.

  If you use a nonzero offset, then that many elements at the beginning of the vector are left empty (the `glob` function fills them with null pointers).

  The `gl_offs` field is meaningful only if you use the `GLOB_DOOFFS` flag. Otherwise, the offset is always zero regardless of what is in this field, and the first real element comes at the beginning of the vector.

`gl_closedir`
: The address of an alternative implementation of the `closedir` function; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `void (*) (void *)`.

  This is a GNU extension.

`gl_readdir`
: The address of an alternative implementation of the `readdir` function used to read the contents of a directory; it is used if the

GLOB_ALTDIRFUNC bit is set in the flag parameter. The type of this field is `struct dirent *(*) (void *)`.

This is a GNU extension.

`gl_opendir`
: The address of an alternative implementation of the `opendir` function; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `void *(*) (const char *)`.

  This is a GNU extension.

`gl_stat`
: The address of an alternative implementation of the `stat` function to get information about an object in the file system; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `int (*) (const char *, struct stat *)`.

  This is a GNU extension.

`gl_lstat`
: The address of an alternative implementation of the `lstat` function to get information about an object in the file systems, not following symbolic links; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `int (*) (const char *, struct stat *)`.

  This is a GNU extension.

For use in the `glob64` function, '`glob.h`' contains another definition for a very similar type. `glob64_t` differs from `glob_t` only in the types of the members `gl_readdir`, `gl_stat` and `gl_lstat`.

glob64_t Data Type

This data type holds a pointer to a word vector. More precisely, it records both the address of the word vector and its size. The GNU implementation contains some more fields that are nonstandard extensions.

`gl_pathc`
: The number of elements in the vector, excluding the initial null entries if the GLOB_DOOFFS flag is used (see `gl_offs` below).

`gl_pathv`
: The address of the vector; this field has type `char **`.

`gl_offs`
: The offset of the first real element of the vector, from its nominal address in the `gl_pathv` field; unlike the other fields, this is always an input to `glob`, rather than an output from it.

  If you use a nonzero offset, then that many elements at the beginning of the vector are left empty. (The `glob` function fills them with null pointers.)

The `gl_offs` field is meaningful only if you use the `GLOB_DOOFFS` flag. Otherwise, the offset is always zero regardless of what is in this field, and the first real element comes at the beginning of the vector.

`gl_closedir`
: The address of an alternative implementation of the `closedir` function; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `void (*) (void *)`.

  This is a GNU extension.

`gl_readdir`
: The address of an alternative implementation of the `readdir64` function used to read the contents of a directory; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `struct dirent64 *(*) (void *)`.

  This is a GNU extension.

`gl_opendir`
: The address of an alternative implementation of the `opendir` function; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `void *(*) (const char *)`.

  This is a GNU extension.

`gl_stat`
: The address of an alternative implementation of the `stat64` function to get information about an object in the file system; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `int (*) (const char *, struct stat64 *)`.

  This is a GNU extension.

`gl_lstat`
: The address of an alternative implementation of the `lstat64` function to get information about an object in the file systems, not following symbolic links; it is used if the `GLOB_ALTDIRFUNC` bit is set in the flag parameter. The type of this field is `int (*) (const char *, struct stat64 *)`.

  This is a GNU extension.

`int` **glob** (`const char *`*pattern*`, int` *flags*`, int (*`*errfunc*`) (const char *`*filename*`, int` *error-code*`), glob_t *`*vector-ptr*) Function

The function `glob` does globbing using the pattern *pattern* in the current directory. It puts the result in a newly allocated vector, and stores the size and address of this vector into `*`*vector-ptr*. The argument *flags* is a combination of bit flags (see Section 13.2.2 [Flags for Globbing], page 361).

The result of globbing is a sequence of file names. The function `glob` allocates a string for each resulting word, then allocates a vector of type `char **` to store

the addresses of these strings. The last element of the vector is a null pointer. This vector is called the *word vector*.

To return this vector, `glob` stores both its address and its length (number of elements, not counting the terminating null pointer) into **vector-ptr*.

Normally, `glob` sorts the file names alphabetically before returning them. You can turn this off with the flag `GLOB_NOSORT` if you want to get the information as fast as possible. Usually it's a good idea to let `glob` sort them—if you process the files in alphabetical order, the users will have a feel for the rate of progress that your application is making.

If `glob` succeeds, it returns 0. Otherwise, it returns one of these error codes:

`GLOB_ABORTED`
: There was an error opening a directory, and you used the flag `GLOB_ERR` or your specified *errfunc* returned a nonzero value. See below for an explanation of the `GLOB_ERR` flag and *errfunc*.

`GLOB_NOMATCH`
: The pattern didn't match any existing files. If you use the `GLOB_NOCHECK` flag, then you never get this error code, because that flag tells `glob` to *pretend* that the pattern matched at least one file.

`GLOB_NOSPACE`
: It was impossible to allocate memory to hold the result.

In the event of an error, `glob` stores information in **vector-ptr* about all the matches it has found so far.

It is important to notice that the `glob` function will not fail if it encounters directories or files that cannot be handled without the LFS interfaces. The implementation of `glob` is supposed to use these functions internally. This at least is the assumption made by the Unix standard. The GNU extension of allowing the user to provide her own directory handling and `stat` functions complicates things a bit. If these callback functions are used and a large file or directory is encountered, `glob` *can* fail.

`int` **glob64** (`const char` **pattern*, `int` *flags*, `int` (**errfunc*) (`const char` **filename*, `int` *error-code*), `glob64_t` **vector-ptr*) — Function

The `glob64` function was added as part of the Large File Summit extensions but is not part of the original LFS proposal. The reason for this is simple—it is not necessary. The necessity for a `glob64` function is added by the extensions of the GNU `glob` implementation, which allows the user to provide his own directory handling and `stat` functions. The `readdir` and `stat` functions do depend on the choice of `_FILE_OFFSET_BITS`, since the definition of the types `struct dirent` and `struct stat` will change depending on the choice.

Besides this difference, the `glob64` works just like `glob` in all aspects.

This function is a GNU extension.

13.2.2 Flags for Globbing

This section describes the flags that you can specify in the *flags* argument to `glob`. Choose the flags you want, and combine them with the C bit-wise OR operator '`|`'.

`GLOB_APPEND`
: Append the words from this expansion to the vector of words produced by previous calls to `glob`. This way you can effectively expand several words as if they were concatenated with spaces between them.

  In order for appending to work, you must not modify the contents of the word vector structure between calls to `glob`. And, if you set `GLOB_DOOFFS` in the first call to `glob`, you must also set it when you append to the results.

  Note that the pointer stored in `gl_pathv` may no longer be valid after you call `glob` the second time, because `glob` might have relocated the vector. So always fetch `gl_pathv` from the `glob_t` structure after each `glob` call; **never** save the pointer across calls.

`GLOB_DOOFFS`
: Leave blank slots at the beginning of the vector of words. The `gl_offs` field says how many slots to leave. The blank slots contain null pointers.

`GLOB_ERR`
: Give up right away and report an error if there is any difficulty reading the directories that must be read in order to expand *pattern* fully. Such difficulties might include a directory in which you don't have the requisite access. Normally, `glob` tries its best to keep on going despite any errors, reading whatever directories it can.

  You can exercise even more control than this by specifying an error-handler function *errfunc* when you call `glob`. If *errfunc* is not a null pointer, then `glob` doesn't give up right away when it can't read a directory; instead, it calls *errfunc* with two arguments, like this:

  ```
  (*errfunc) (filename, error-code)
  ```

  The argument *filename* is the name of the directory that `glob` couldn't open or couldn't read, and *error-code* is the `errno` value that was reported to `glob`.

  If the error-handler function returns nonzero, then `glob` gives up right away. Otherwise, it continues.

`GLOB_MARK`
: If the pattern matches the name of a directory, append '`/`' to the directory's name when returning it.

`GLOB_NOCHECK`
: If the pattern doesn't match any file names, return the pattern itself as if it were a file name that had been matched. (Normally, when the pattern doesn't match anything, `glob` returns that there were no matches.)

`GLOB_NOSORT`
: Don't sort the file names; return them in no particular order. (In practice, the order will depend on the order of the entries in the directory.) The only reason *not* to sort is to save time.

`GLOB_NOESCAPE`
: Don't treat the '`\`' character specially in patterns. Normally, '`\`' quotes the following character, turning off its special meaning (if any) so that it matches only itself. When quoting is enabled, the pattern '`\?`' matches only the string '`?`', because the question mark in the pattern acts like an ordinary character.

  If you use `GLOB_NOESCAPE`, then '`\`' is an ordinary character.

  `glob` does its work by calling the function `fnmatch` repeatedly. It handles the flag `GLOB_NOESCAPE` by turning on the `FNM_NOESCAPE` flag in calls to `fnmatch`.

13.2.3 More Flags for Globbing

Besides the flags described in the last section, the GNU implementation of `glob` allows a few more flags, which are also defined in the '`glob.h`' file. Some of the extensions implement functionality that is available in modern shell implementations.

`GLOB_PERIOD`
: The `.` character (period) is treated special. It cannot be matched by wildcards (see Section 13.1 [Wildcard Matching], page 355, `FNM_PERIOD`).

`GLOB_MAGCHAR`
: The `GLOB_MAGCHAR` value is not to be given to `glob` in the *flags* parameter. Instead, `glob` sets this bit in the *gl_flags* element of the *glob_t* structure provided as the result if the pattern used for matching contains any wildcard character.

`GLOB_ALTDIRFUNC`
: Instead of using the normal functions for accessing the file system, the `glob` implementation uses the user-supplied functions specified in the structure pointed to by the *pglob* parameter.[1]

[1] For more information about the functions, see Loosemore et al., "Accessing Directories" and "Reading the Attributes of a File" (see chap. 1, n. 1).

GLOB_BRACE

If this flag is given, the handling of braces in the pattern is changed. It is now required that braces appear correctly grouped—for each opening brace there must be a closing one. Braces can be used recursively. So it is possible to define one brace expression in another one. It is important to note that the range of each brace expression is completely contained in the outer brace expression (if there is one).

The string between the matching braces is separated into single expressions by splitting at ‘,’ (comma) characters. The commas themselves are discarded. Please note what we said above about recursive brace expressions. The commas used to separate the subexpressions must be at the same level. Commas in brace subexpressions are not matched. They are used during expansion of the brace expression of the deeper level. The example below shows this:

```
glob ("{foo/{,bar,biz},baz}", GLOB_BRACE, NULL, &result)
```

is equivalent to the sequence:

```
glob ("foo/", GLOB_BRACE, NULL, &result)
glob ("foo/bar", GLOB_BRACE|GLOB_APPEND, NULL, &result)
glob ("foo/biz", GLOB_BRACE|GLOB_APPEND, NULL, &result)
glob ("baz", GLOB_BRACE|GLOB_APPEND, NULL, &result)
```

if we leave aside error handling.

GLOB_NOMAGIC

If the pattern contains no wildcard constructs (it is a literal file name), return it as the sole “matching” word, even if no file exists by that name.

GLOB_TILDE

If this flag is used, the character ‘~’ (tilde) is handled specially if it appears at the beginning of the pattern. Instead of being taken verbatim, it is used to represent the home directory of a known user.

If ‘~’ is the only character in the pattern or it is followed by a ‘/’ (slash), the home directory of the process owner is substituted. Using `getlogin` and `getpwnam`, the information is read from the system databases. As an example, take user `bart` with his home directory at ‘`/home/bart`’. For him a call like:

```
glob ("~/bin/*", GLOB_TILDE, NULL, &result)
```

would return the contents of the directory ‘`/home/bart/bin`’. Instead of referring to your own home directory, it is also possible to name the home directory of other users. To do so one has to append the user name after the tilde character. So the contents of user `homer`’s ‘`bin`’ directory can be retrieved by:

```
glob ("~homer/bin/*", GLOB_TILDE, NULL, &result)
```

If the user name is not valid or the home directory cannot be determined for some reason, the pattern is left untouched and is itself used

as the result. For example, if in the last example `home` is not available, the tilde expansion yields to `"~homer/bin/*"` and `glob` is not looking for a directory named `~homer`.

This functionality is equivalent to what is available in C shells if the `nonomatch` flag is set.

`GLOB_TILDE_CHECK`
: If this flag is used, `glob` behaves as if `GLOB_TILDE` is given. The only difference is that if the user name is not available or the home directory cannot be determined for other reasons, this leads to an error. `glob` will return `GLOB_NOMATCH` instead of using the pattern itself as the name.

  This functionality is equivalent to what is available in C shells if `nonomatch` flag is not set.

`GLOB_ONLYDIR`
: If this flag is used, the globbing function takes this as a *hint* that the caller is only interested in directories matching the pattern. If the information about the type of the file is easily available, nondirectories will be rejected, but no extra work will be done to determine the information for each file—the caller must still be able to filter directories out.

  This functionality is only available with the GNU `glob` implementation. It is mainly used internally to increase performance but might be useful for a user as well.

Calling `glob` will in most cases allocate resources, which are used to represent the result of the function call. If the same object of type `glob_t` is used in multiple calls to `glob`, the resources are freed or reused so that no leaks appear. But this does not include the time when all `glob` calls are done.

`void` **globfree** (`glob_t *`*pglob*) Function

The `globfree` function frees all resources allocated by previous calls to `glob` associated with the object pointed to by *pglob*. This function should be called whenever the currently used `glob_t` typed object isn't used anymore.

`void` **globfree64** (`glob64_t *`*pglob*) Function

This function is equivalent to `globfree`, but it frees records of type `glob64_t` that were allocated by `glob64`.

13.3 Regular Expression Matching

The GNU C Library supports two interfaces for matching regular expressions. One is the standard POSIX.2 interface, and the other is what the GNU system has had for many years.

Both interfaces are declared in the header file ‘regex.h’. If you define _POSIX_C_SOURCE, then only the POSIX.2 functions, structures and constants are declared.

13.3.1 POSIX Regular Expression Compilation

Before you can actually match a regular expression, you must *compile* it. This is not true compilation—it produces a special data structure, not machine instructions. But it is like ordinary compilation in that its purpose is to enable you to “execute” the pattern quickly (see Section 13.3.3 [Matching a Compiled POSIX Regular Expression], page 367, for how to use the compiled regular expression for matching).

There is a special data type for compiled regular expressions:

regex_t Data Type

This type of object holds a compiled regular expression. It is actually a structure. It has just one field that your programs should look at:

`re_nsub` This field holds the number of parenthetical subexpressions in the regular expression that was compiled.

There are several other fields, but we don’t describe them here, because only the functions in the library should use them.

After you create a `regex_t` object, you can compile a regular expression into it by calling `regcomp`.

`int` **regcomp** (`regex_t *`*compiled*`, const char *`*pattern*`, int` *cflags*) Function

The function `regcomp` “compiles” a regular expression into a data structure that you can use with `regexec` to match against a string. The compiled regular expression format is designed for efficient matching. `regcomp` stores it into `*`*compiled*.

It’s up to you to allocate an object of type `regex_t` and pass its address to `regcomp`.

The argument *cflags* lets you specify various options that control the syntax and semantics of regular expressions (see Section 13.3.2 [Flags for POSIX Regular Expressions], page 367).

If you use the flag `REG_NOSUB`, then `regcomp` omits from the compiled regular expression the information necessary to record how subexpressions actually match. In this case, you might as well pass `0` for the *matchptr* and *nmatch* arguments when you call `regexec`.

If you don’t use `REG_NOSUB`, then the compiled regular expression does have the capacity to record how subexpressions match. Also, `regcomp` tells you how many subexpressions *pattern* has, by storing the number in *compiled*`->re_nsub`. You can use that value to decide how long of an array to allocate to hold information about subexpression matches.

`regcomp` returns `0` if it succeeds in compiling the regular expression; otherwise, it returns a nonzero error code (see the table below). You can use `regerror` to produce an error message string describing the reason for a nonzero value (see Section 13.3.6 [POSIX Regexp Matching Clean-Up], page 369).

Here are the possible nonzero values that `regcomp` can return:

`REG_BADBR`
: There was an invalid '`\{...\}`' construct in the regular expression. A valid '`\{...\}`' construct must contain either a single number, or two numbers in increasing order separated by a comma.

`REG_BADPAT`
: There was a syntax error in the regular expression.

`REG_BADRPT`
: A repetition operator such as '`?`' or '`*`' appeared in a bad position (with no preceding subexpression to act on).

`REG_ECOLLATE`
: The regular expression referred to an invalid collating element (one not defined in the current locale for string collation) (see Section 7.3 [Categories of Activities That Locales Affect], page 182).

`REG_ECTYPE`
: The regular expression referred to an invalid character class name.

`REG_EESCAPE`
: The regular expression ended with '`\`'.

`REG_ESUBREG`
: There was an invalid number in the '`\`*digit*' construct.

`REG_EBRACK`
: There were unbalanced square brackets in the regular expression.

`REG_EPAREN`
: An extended regular expression had unbalanced parentheses, or a basic regular expression had unbalanced '`\(`' and '`\)`'.

`REG_EBRACE`
: The regular expression had unbalanced '`\{`' and '`\}`'.

`REG_ERANGE`
: One of the endpoints in a range expression was invalid.

`REG_ESPACE`
: `regcomp` ran out of memory.

13.3.2 Flags for POSIX Regular Expressions

These are the bit flags that you can use in the *cflags* operand when compiling a regular expression with `regcomp`.

`REG_EXTENDED`
: Treat the pattern as an extended regular expression, rather than as a basic regular expression.

`REG_ICASE`
: Ignore case when matching letters.

`REG_NOSUB`
: Don't bother storing the contents of the *matches-ptr* array.

`REG_NEWLINE`
: Treat a newline in *string* as dividing *string* into multiple lines, so that '`$`' can match before the newline and '`^`' can match after. Also, don't permit '`.`' to match a newline, and don't permit '`[^...]`' to match a newline.

  Otherwise, newline acts like any other ordinary character.

13.3.3 Matching a Compiled POSIX Regular Expression

Once you have compiled a regular expression, as described in Section 13.3.1 [POSIX Regular Expression Compilation], page 365, you can match it against strings using `regexec`. A match anywhere inside the string counts as success, unless the regular expression contains anchor characters ('`^`' or '`$`').

`int` **regexec** (`regex_t` **compiled*, `char` **string*, `size_t` *nmatch*, `regmatch_t` *matchptr* [], `int` *eflags*) — Function

This function tries to match the compiled regular expression **compiled* against *string*.

`regexec` returns `0` if the regular expression matches; otherwise, it returns a nonzero value. See the table below for what nonzero values mean. You can use `regerror` to produce an error message string describing the reason for a nonzero value (see Section 13.3.6 [POSIX Regexp Matching Clean-Up], page 369).

The argument *eflags* is a word of bit flags that enable various options.

If you want to get information about what part of *string* actually matched the regular expression or its subexpressions, use the arguments *matchptr* and *nmatch*. Otherwise, pass `0` for *nmatch*, and `NULL` for *matchptr* (see Section 13.3.4 [Match Results with Subexpressions], page 368).

You must match the regular expression with the same set of current locales that were in effect when you compiled the regular expression.

The function `regexec` accepts the following flags in the *eflags* argument:

`REG_NOTBOL`
: Do not regard the beginning of the specified string as the beginning of a line; more generally, don't make any assumptions about what text might precede it.

`REG_NOTEOL`
: Do not regard the end of the specified string as the end of a line; more generally, don't make any assumptions about what text might follow it.

Here are the possible nonzero values that `regexec` can return:

`REG_NOMATCH`
: The pattern didn't match the string. This isn't really an error.

`REG_ESPACE`
: `regexec` ran out of memory.

13.3.4 Match Results with Subexpressions

When `regexec` matches parenthetical subexpressions of *pattern*, it records which parts of *string* they match. It returns that information by storing the offsets into an array whose elements are structures of type `regmatch_t`. The first element of the array (index `0`) records the part of the string that matched the entire regular expression. Each other element of the array records the beginning and end of the part that matched a single parenthetical subexpression.

regmatch_t Data Type

This is the data type of the *matcharray* array that you pass to `regexec`. It contains two structure fields, as follows:

`rm_so`
: The offset in *string* of the beginning of a substring. Add this value to *string* to get the address of that part.

`rm_eo`
: The offset in *string* of the end of the substring.

regoff_t Data Type

`regoff_t` is an alias for another signed integer type. The fields of `regmatch_t` have type `regoff_t`.

The `regmatch_t` elements correspond to subexpressions positionally; the first element (index `1`) records where the first subexpression matched, the second element records the second subexpression, and so on. The order of the subexpressions is the order in which they begin.

When you call `regexec`, you specify how long the *matchptr* array is, with the *nmatch* argument. This tells `regexec` how many elements to store. If the actual regular expression has more than *nmatch* subexpressions, then you won't get offset information about the rest of them. But this doesn't alter whether the pattern matches a particular string or not.

If you don't want `regexec` to return any information about where the subexpressions matched, you can either supply `0` for *nmatch*, or use the flag `REG_NOSUB` when you compile the pattern with `regcomp`.

13.3.5 Complications in Subexpression Matching

Sometimes, a subexpression matches a substring of no characters. This happens when '`f\(o*\)`' matches the string '`fum`' (it really matches just the '`f`'). In this case, both of the offsets identify the point in the string where the null substring was found. In this example, the offsets are both `1`.

Sometimes, the entire regular expression can match without using some of its subexpressions at all—for example, when '`ba\(na\)*`' matches the string '`ba`', the parenthetical subexpression is not used. When this happens, `regexec` stores `-1` in both fields of the element for that subexpression.

Sometimes matching the entire regular expression can match a particular subexpression more than once—for example, when '`ba\(na\)*`' matches the string '`bananana`', the parenthetical subexpression matches three times. When this happens, `regexec` usually stores the offsets of the last part of the string that matched the subexpression. In the case of '`bananana`', these offsets are `6` and `8`.

But the last match is not always the one that is chosen. It's more accurate to say that the last *opportunity* to match is the one that takes precedence. What this means is that when one subexpression appears within another, then the results reported for the inner subexpression reflect whatever happened on the last match of the outer subexpression. For an example, consider '`\(ba\(na\)*s \)*`' matching the string '`bananas bas `'. The last time the inner expression actually matches is near the end of the first word. But it is *considered* again in the second word, and fails to match there. `regexec` reports nonuse of the '`na`' subexpression.

Another place where this rule applies is when the regular expression:

```
\(ba\(na\)*s \|nefer\(ti\)* \)*
```

matches '`bananas nefertiti`'. The '`na`' subexpression does match in the first word, but it doesn't match in the second word because the other alternative is used there. Once again, the second repetition of the outer subexpression overrides the first, and within that second repetition, the '`na`' subexpression is not used. So `regexec` reports nonuse of the '`na`' subexpression.

13.3.6 POSIX Regexp Matching Clean-Up

When you are finished using a compiled regular expression, you can free the storage it uses by calling `regfree`.

`void` **regfree** (`regex_t *`*compiled*) Function

Calling `regfree` frees all the storage that **compiled* points to. This includes various internal fields of the `regex_t` structure that aren't documented in this manual.

`regfree` does not free the object **compiled* itself.

You should always free the space in a `regex_t` structure with `regfree` before using the structure to compile another regular expression.

When `regcomp` or `regexec` reports an error, you can use the function `regerror` to turn it into an error message string.

`size_t` **regerror** (`int` *errcode*, `regex_t *`*compiled*, `char *`*buffer*, `size_t` *length*) Function

This function produces an error message string for the error code *errcode*, and stores the string in *length* bytes of memory starting at *buffer*. For the *compiled* argument, supply the same compiled regular expression structure that `regcomp` or `regexec` was working with when it got the error. Alternatively, you can supply `NULL` for *compiled*; you will still get a meaningful error message, but it might not be as detailed.

If the error message can't fit in *length* bytes (including a terminating null character), then `regerror` truncates it. The string that `regerror` stores is always null-terminated even if it has been truncated.

The return value of `regerror` is the minimum length needed to store the entire error message. If this is less than *length*, then the error message was not truncated, and you can use it. Otherwise, you should call `regerror` again with a larger buffer.

Here is a function that uses `regerror`, but always dynamically allocates a buffer for the error message:

```
char *get_regerror (int errcode, regex_t *compiled)
{
  size_t length = regerror (errcode, compiled, NULL, 0);
  char *buffer = xmalloc (length);
  (void) regerror (errcode, compiled, buffer, length);
  return buffer;
}
```

13.4 Shell-Style Word Expansion

Word expansion means the process of splitting a string into *words* and substituting for variables, commands, and wildcards just as the shell does.

For example, when you write '`ls -l foo.c`', this string is split into three separate words—'`ls`', '`-l`' and '`foo.c`'. This is the most basic function of word expansion.

When you write '`ls *.c`', this can become many words, because the word '`*.c`' can be replaced with any number of file names. This is called *wildcard expansion*, and it is also a part of word expansion.

When you use '`echo $PATH`' to print your path, you are taking advantage of *variable substitution*, which is also part of word expansion.

Ordinary programs can perform word expansion just like the shell by calling the library function `wordexp`.

13.4.1 The Stages of Word Expansion

When word expansion is applied to a sequence of words, it performs the following transformations in the order shown here:

1. *Tilde expansion* '`~foo`' us replaced with the name of the home directory of '`foo`'.
2. Next, three different transformations are applied in the same step, from left to right:
   - *Variable substitution* Environment variables are substituted for references such as '`$foo`'.
   - *Command substitution* Constructs such as '`` `cat foo` ``' and the equivalent '`$(cat foo)`' are replaced with the output from the inner command.
   - *Arithmetic expansion* Constructs such as '`$((x-1))`' are replaced with the result of the arithmetic computation.
3. *Field splitting* The text is subdivided into *words*.
4. *Wildcard expansion* A construct such as '`*.c`' is replaced with a list of '`.c`' file names. Wildcard expansion applies to an entire word at a time, and replaces that word with 0 or more file names that are themselves words.
5. *Quote removal* String-quotes are deleted, now that they have done their job by inhibiting the above transformations when appropriate.

For the details of these transformations, and how to write the constructs that use them, see *The GNU Bash Reference Manual*.[2]

13.4.2 Calling `wordexp`

All the functions, constants and data types for word expansion are declared in the header file '`wordexp.h`'.

Word expansion produces a vector of words (strings). To return this vector, `wordexp` uses a special data type, `wordexp_t`, which is a structure. You pass `wordexp` the address of the structure, and it fills in the structure's fields to tell you about the results.

wordexp_t Data Type

This data type holds a pointer to a word vector. More precisely, it records both the address of the word vector and its size.

`we_wordc`
: This is the number of elements in the vector.

`we_wordv`
: This is the address of the vector. This field has type `char **`.

[2] Chet Ramey and Brian Fox, *The GNU Bash Reference Manual* (Bristol, UK: Network Theory Ltd., January 2003), http://www.gnu.org/software/bash/manual/bashref.html.

`we_offs` This is the offset of the first real element of the vector, from its nominal address in the `we_wordv` field. Unlike the other fields, this is always an input to `wordexp`, rather than an output from it.

If you use a nonzero offset, then that many elements at the beginning of the vector are left empty (the `wordexp` function fills them with null pointers).

The `we_offs` field is meaningful only if you use the `WRDE_DOOFFS` flag. Otherwise, the offset is always zero regardless of what is in this field, and the first real element comes at the beginning of the vector.

`int` **wordexp** (`const char` ⋆*words*`, wordexp_t` ⋆*word-vector-ptr*`, int` *flags*) Function

Perform word expansion on the string *words*, putting the result in a newly allocated vector, and store the size and address of this vector into ⋆*word-vector-ptr*. The argument *flags* is a combination of bit flags (see Section 13.4.3 [Flags for Word Expansion], page 373).

You shouldn't use any of the characters '`|&;<>`' in the string *words* unless they are quoted; likewise for newline. If you use these characters unquoted, you will get the `WRDE_BADCHAR` error code. Don't use parentheses or braces unless they are quoted or part of a word expansion construct. If you use quotation characters '`'"\`', they should come in pairs that balance.

The results of word expansion are a sequence of words. The function `wordexp` allocates a string for each resulting word, then allocates a vector of type `char **` to store the addresses of these strings. The last element of the vector is a null pointer. This vector is called the *word vector*.

To return this vector, `wordexp` stores both its address and its length (number of elements, not counting the terminating null pointer) into ⋆*word-vector-ptr*.

If `wordexp` succeeds, it returns 0. Otherwise, it returns one of these error codes:

`WRDE_BADCHAR`
: The input string *words* contains an unquoted invalid character such as '`|`'.

`WRDE_BADVAL`
: The input string refers to an undefined shell variable, and you used the flag `WRDE_UNDEF` to forbid such references.

`WRDE_CMDSUB`
: The input string uses command substitution, and you used the flag `WRDE_NOCMD` to forbid command substitution.

`WRDE_NOSPACE`
: It was impossible to allocate memory to hold the result. In this case, `wordexp` can store part of the results—as much as it could allocate room for.

`WRDE_SYNTAX`
: There was a syntax error in the input string. For example, an unmatched quoting character is a syntax error.

`void` **wordfree** (`wordexp_t` ⋆*word-vector-ptr*) Function

Free the storage used for the word-strings and vector that ⋆*word-vector-ptr* points to. This does not free the structure ⋆*word-vector-ptr* itself—only the other data it points to.

13.4.3 Flags for Word Expansion

This section describes the flags that you can specify in the *flags* argument to `wordexp`. Choose the flags you want, and combine them with the C operator '`|`'.

`WRDE_APPEND`
: Append the words from this expansion to the vector of words produced by previous calls to `wordexp`. This way you can effectively expand several words as if they were concatenated with spaces between them.

  In order for appending to work, you must not modify the contents of the word vector structure between calls to `wordexp`. And, if you set `WRDE_DOOFFS` in the first call to `wordexp`, you must also set it when you append to the results.

`WRDE_DOOFFS`
: Leave blank slots at the beginning of the vector of words. The `we_offs` field says how many slots to leave. The blank slots contain null pointers.

`WRDE_NOCMD`
: Don't do command substitution; if the input requests command substitution, report an error.

`WRDE_REUSE`
: Reuse a word vector made by a previous call to `wordexp`. Instead of allocating a new vector of words, this call to `wordexp` will use the vector that already exists (making it larger if necessary).

  The vector may move, so it is not safe to save an old pointer and use it again after calling `wordexp`. You must fetch `we_pathv` anew after each call.

`WRDE_SHOWERR`
: Do show any error messages printed by commands run by command substitution. More precisely, allow these commands to inherit the standard error output stream of the current process. By default, `wordexp` gives these commands a standard error stream that discards all output.

`WRDE_UNDEF`
: If the input refers to a shell variable that is not defined, report an error.

13.4.4 `wordexp` Example

Here is an example of using `wordexp` to expand several strings and use the results to run a shell command. It also shows the use of `WRDE_APPEND` to concatenate the expansions and of `wordfree` to free the space allocated by `wordexp`.

```
int
expand_and_execute (const char *program, const char **options)
{
  wordexp_t result;
  pid_t pid
  int status, i;

  /* Expand the string for the program to run.  */
  switch (wordexp (program, &result, 0))
    {
    case 0:  /* Successful.  */
      break;
    case WRDE_NOSPACE:
      /* If the error was WRDE_NOSPACE,
         then perhaps part of the result was allocated.  */
      wordfree (&result);
    default:                    /* Some other error.  */
      return -1;
    }

  /* Expand the strings specified for the arguments.  */
  for (i = 0; options[i] != NULL; i++)
    {
      if (wordexp (options[i], &result, WRDE_APPEND))
        {
          wordfree (&result);
          return -1;
        }
    }

  pid = fork ();
  if (pid == 0)
    {
      /* This is the child process.  Execute the command.  */
      execv (result.we_wordv[0], result.we_wordv);
      exit (EXIT_FAILURE);
```

```
    }
  else if (pid < 0)
    /* The fork failed.  Report failure.  */
    status = -1;
  else
    /* This is the parent process.  Wait for the child to complete.  */
    if (waitpid (pid, &status, 0) != pid)
      status = -1;

  wordfree (&result);
  return status;
}
```

13.4.5 Details of Tilde Expansion

It's a standard part of shell syntax that you can use '`~`' at the beginning of a file name to stand for your own home directory. You can use '`~`*user*' to stand for *user*'s home directory.

Tilde expansion is the process of converting these abbreviations to the directory names that they stand for.

Tilde expansion applies to the '`~`' plus all following characters up to white space or a slash. It takes place only at the beginning of a word, and only if none of the characters to be transformed is quoted in any way.

Plain '`~`' uses the value of the environment variable `HOME` as the proper home directory name. '`~`' followed by a user name uses `getpwname` to look up that user in the user database, and uses whatever directory is recorded there. Thus, '`~`' followed by your own name can give different results from plain '`~`', if the value of `HOME` is not really your home directory.

13.4.6 Details of Variable Substitution

Part of ordinary shell syntax is the use of '`$`*variable*' to substitute the value of a shell variable into a command. This is called *variable substitution*, and it is one part of doing word expansion.

There are two basic ways you can write a variable reference for substitution:

`${`*variable*`}`
: If you write braces around the variable name, then it is completely unambiguous where the variable name ends. You can concatenate additional letters onto the end of the variable value by writing them immediately after the close brace. For example, '`${foo}s`' expands into '`tractors`'.

`$`*variable*
: If you do not put braces around the variable name, then the variable name consists of all the alphanumeric characters and underscores that follow the '`$`'. The next punctuation character ends the variable

name. Thus, '`$foo-bar`' refers to the variable `foo` and expands into '`tractor-bar`'.

When you use braces, you can also use various constructs to modify the value that is substituted, or test it in various ways:

`${`*variable*`:-`*default*`}`
Substitute the value of *variable*, but if that is empty or undefined, use *default* instead.

`${`*variable*`:=`*default*`}`
Substitute the value of *variable*, but if that is empty or undefined, use *default* instead and set the variable to *default*.

`${`*variable*`:?`*message*`}`
If *variable* is defined and not empty, substitute its value.

Otherwise, print *message* as an error message on the standard error stream, and consider word expansion a failure.

`${`*variable*`:+`*replacement*`}`
Substitute *replacement*, but only if *variable* is defined and nonempty. Otherwise, substitute nothing for this construct.

`${#`*variable*`}`
Substitute a numeral that expresses in base-10 the number of characters in the value of *variable*. '`${#foo}`' stands for '`7`', because '`tractor`' is seven characters.

These variants of variable substitution let you remove part of the variable's value before substituting it. The *prefix* and *suffix* are not mere strings; they are wildcard patterns, just like the patterns that you use to match multiple file names. But in this context, they match against parts of the variable value rather than against file names.

`${`*variable*`%%`*suffix*`}`
Substitute the value of *variable*, but first discard from that variable any portion at the end that matches the pattern *suffix*.

If there is more than one alternative for how to match against *suffix*, this construct uses the longest possible match.

Thus, '`${foo%%r*}`' substitutes '`t`', because the largest match for '`r*`' at the end of '`tractor`' is '`ractor`'.

`${`*variable*`%`*suffix*`}`
Substitute the value of *variable*, but first discard from that variable any portion at the end that matches the pattern *suffix*.

If there is more than one alternative for how to match against *suffix*, this construct uses the shortest possible alternative.

Thus, '`${foo%r*}`' substitutes '`tracto`', because the shortest match for '`r*`' at the end of '`tractor`' is just '`r`'.

`${`*variable*`##`*prefix*`}`

Substitute the value of *variable*, but first discard from that variable any portion at the beginning that matches the pattern *prefix*.

If there is more than one alternative for how to match against *prefix*, this construct uses the longest possible match.

Thus, '`${foo##*t}`' substitutes '`or`', because the largest match for '`*t`' at the beginning of '`tractor`' is '`tract`'.

`${`*variable*`#`*prefix*`}`

Substitute the value of *variable*, but first discard from that variable any portion at the beginning that matches the pattern *prefix*.

If there is more than one alternative for how to match against *prefix*, this construct uses the shortest possible alternative.

Thus, '`${foo#*t}`' substitutes '`ractor`', because the shortest match for '`*t`' at the beginning of '`tractor`' is just '`t`'.

14 The Basic Program/System Interface

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies. Though it may have multiple threads of control within the same program and a program may be composed of multiple logically separate modules, a process always executes exactly one program.

We are using a specific definition of *program* for the purposes of this manual, which corresponds to a common definition in the context of Unix system. In popular usage, *program* enjoys a much broader definition; it can refer for example to a system's kernel, an editor macro, a complex package of software, or a discrete section of code executing within a process.

Writing the program is what this manual is all about. This chapter explains the most basic interface between your program and the system that runs, or calls, it. This includes passing of parameters (arguments and environment) from the system, requesting basic services from the system, and telling the system the program is done.

A program starts another program with the `exec` family of system calls. This chapter looks at program start-up from the execee's point of view.[1]

14.1 Program Arguments

The system starts a C program by calling the function `main`. It is up to you to write a function named `main`—otherwise, you won't even be able to link your program without errors.

In ISO C you can define `main` either to take no arguments, or to take two arguments that represent the command-line arguments to the program, like this:

```
int main (int argc, char *argv[])
```

The command-line arguments are the white-space-separated tokens given in the shell command used to invoke the program; thus, in '`cat foo bar`', the arguments are '`foo`' and '`bar`'. The only way a program can look at its command-line arguments is via the arguments of `main`. If `main` doesn't take arguments, then you cannot get at the command line.

The value of the *argc* argument is the number of command-line arguments. The *argv* argument is a vector of C strings; its elements are the individual command-line argument strings. The file name of the program being run is also included in the vector as the first element; the value of *argc* counts this element. A null pointer always follows the last element: *argv*`[`*argc*`]` is this null pointer.

For the command '`cat foo bar`', *argc* is 3 and *argv* has three elements, '`cat`', '`foo`' and '`bar`'.

[1] See Loosemore et al., "Executing a File" (see chap. 1, n. 1), to see the event from the execor's point of view.

In Unix systems you can define `main` a third way, using three arguments:

```
int main (int argc, char *argv[], char *envp[])
```

The first two arguments are just the same. The third argument *envp* gives the program's environment; it is the same as the value of `environ` (see Section 14.4 [Environment Variables], page 418). POSIX.1 does not allow this three-argument form, so to be portable it is best to write `main` to take two arguments, and use the value of `environ`.

14.1.1 Program Argument Syntax Conventions

POSIX recommends these conventions for command-line arguments. `getopt` (see Section 14.2 [Parsing Program Options Using `getopt`], page 381) and `argp_parse` (see Section 14.3 [Parsing Program Options with Argp], page 389) make it easy to implement them.

- Arguments are options if they begin with a hyphen delimiter ('`-`').
- Multiple options may follow a hyphen delimiter in a single token if the options do not take arguments. Thus, '`-abc`' is equivalent to '`-a -b -c`'.
- Option names are single alphanumeric characters (as for `isalnum`; see Section 4.1 [Classification of Characters], page 79).
- Certain options require an argument. For example, the '`-o`' command of the `ld` command requires an argument—an output file name.
- An option and its argument may or may not appear as separate tokens. (In other words, the white space separating them is optional.) Thus, '`-o foo`' and '`-ofoo`' are equivalent.
- Options typically precede other non-option arguments.

  The implementations of `getopt` and `argp_parse` in the GNU C Library normally make it appear as if all the option arguments were specified before all the non-option arguments for the purposes of parsing, even if the user of your program intermixed option and non-option arguments. They do this by reordering the elements of the *argv* array. This behavior is nonstandard; if you want to suppress it, define the `_POSIX_OPTION_ORDER` environment variable (see Section 14.4.2 [Standard Environment Variables], page 421).
- The argument '`--`' terminates all options; any following arguments are treated as non-option arguments, even if they begin with a hyphen.
- A token consisting of a single hyphen character is interpreted as an ordinary non-option argument. By convention, it is used to specify input from or output to the standard input and output streams.
- Options may be supplied in any order, or appear multiple times. The interpretation is left up to the particular application program.

GNU adds *long options* to these conventions. Long options consist of '`--`' followed by a name made of alphanumeric characters and dashes. Option names are typically one to three words long, with hyphens to separate words. Users can abbreviate the option names as long as the abbreviations are unique.

To specify an argument for a long option, write '`--`*name*`=`*value*'. This syntax enables a long option to accept an argument that is itself optional.

Eventually, the GNU system will provide completion for long option names in the shell.

14.1.2 Parsing Program Arguments

If the syntax for the command-line arguments to your program is simple enough, you can pick the arguments off from *argv* by hand. But unless your program takes a fixed number of arguments, or all of the arguments are interpreted in the same way (as file names, for example), you are usually better off using `getopt` (see Section 14.2 [Parsing Program Options Using `getopt`], page 381) or `argp_parse` (see Section 14.3 [Parsing Program Options with Argp], page 389) to do the parsing.

`getopt` is more standard (the short-option-only version of it is a part of the POSIX standard), but using `argp_parse` is often easier, both for very simple and very complex option structures, because it does more of the dirty work for you.

14.2 Parsing Program Options Using `getopt`

The `getopt` and `getopt_long` functions automate some of the chore involved in parsing typical Unix command-line options.

14.2.1 Using the `getopt` Function

Here are the details about how to call the `getopt` function. To use this facility, your program must include the header file '`unistd.h`'.

`int` **opterr** Variable

If the value of this variable is nonzero, then `getopt` prints an error message to the standard error stream if it encounters an unknown option character or an option with a missing required argument. This is the default behavior. If you set this variable to zero, `getopt` does not print any messages, but it still returns the character '`?`' to indicate an error.

`int` **optopt** Variable

When `getopt` encounters an unknown option character or an option with a missing required argument, it stores that option character in this variable. You can use this for providing your own diagnostic messages.

`int` **optind** Variable

This variable is set by `getopt` to the index of the next element of the *argv* array to be processed. Once `getopt` has found all of the option arguments, you can use this variable to determine where the remaining non-option arguments begin. The initial value of this variable is `1`.

`char *` **optarg** Variable

This variable is set by `getopt` to point at the value of the option argument, for those options that accept arguments.

`int` **getopt** (`int` *argc*, `char **`*argv*, `const char *`*options*) Function

The `getopt` function gets the next option argument from the argument list specified by the *argv* and *argc* arguments. Normally these values come directly from the arguments received by `main`.

The *options* argument is a string that specifies the option characters that are valid for this program. An option character in this string can be followed by a colon (‘`:`’) to indicate that it takes a required argument. If an option character is followed by two colons (‘`::`’), its argument is optional; this is a GNU extension.

`getopt` has three ways to deal with options that follow non-options *argv* elements. The special argument ‘`--`’ forces in all cases the end of option scanning.

- The default is to permute the contents of *argv* while scanning it so that eventually all the non-options are at the end. This allows options to be given in any order, even with programs that were not written to expect this.
- If the *options* argument string begins with a hyphen (‘`-`’), this is treated specially. It permits arguments that are not options to be returned as if they were associated with option character ‘`\1`’.
- POSIX demands the following behavior: The first non-option stops option processing. This mode is selected by either setting the environment variable `POSIXLY_CORRECT` or beginning the *options* argument string with a plus sign (‘`+`’).

The `getopt` function returns the option character for the next command-line option. When no more option arguments are available, it returns `-1`. There may still be more non-option arguments; you must compare the external variable `optind` against the *argc* parameter to check this.

If the option has an argument, `getopt` returns the argument by storing it in the variable *optarg*. You don’t ordinarily need to copy the `optarg` string, since it is a pointer into the original *argv* array, not into a static area that might be overwritten.

If `getopt` finds an option character in *argv* that was not included in *options*, or a missing option argument, it returns ‘`?`’ and sets the external variable `optopt` to the actual option character. If the first character of *options* is a colon (‘`:`’), then `getopt` returns ‘`:`’ instead of ‘`?`’ to indicate a missing option argument. In addition, if the external variable `opterr` is nonzero (which is the default), `getopt` prints an error message.

14.2.2 Example of Parsing Arguments with `getopt`

Here is an example showing how `getopt` is typically used. The key points to notice are

- Normally, `getopt` is called in a loop. When `getopt` returns `-1`, indicating no more options are present, the loop terminates.
- A `switch` statement is used to dispatch on the return value from `getopt`. In typical use, each case just sets a variable that is used later in the program.
- A second loop is used to process the remaining non-option arguments.

```
#include <unistd.h>
#include <stdio.h>

int
main (int argc, char **argv)
{
  int aflag = 0;
  int bflag = 0;
  char *cvalue = NULL;
  int index;
  int c;

  opterr = 0;



  while ((c = getopt (argc, argv, "abc:")) != -1)
    switch (c)
      {
      case 'a':
        aflag = 1;
        break;
      case 'b':
        bflag = 1;
        break;
      case 'c':
        cvalue = optarg;
        break;
      case '?':
        if (isprint (optopt))
          fprintf (stderr, "Unknown option `-%c'.\n", optopt);
        else
          fprintf (stderr,
                   "Unknown option character `\\x%x'.\n",
                   optopt);
        return 1;
      default:
        abort ();
```

```
        }


      printf ("aflag = %d, bflag = %d, cvalue = %s\n",
              aflag, bflag, cvalue);

      for (index = optind; index < argc; index++)
        printf ("Non-option argument %s\n", argv[index]);
      return 0;
    }
```

Here are some examples showing what this program prints with different combinations of arguments:

```
% testopt
aflag = 0, bflag = 0, cvalue = (null)

% testopt -a -b
aflag = 1, bflag = 1, cvalue = (null)

% testopt -ab
aflag = 1, bflag = 1, cvalue = (null)

% testopt -c foo
aflag = 0, bflag = 0, cvalue = foo

% testopt -cfoo
aflag = 0, bflag = 0, cvalue = foo

% testopt arg1
aflag = 0, bflag = 0, cvalue = (null)
Non-option argument arg1

% testopt -a arg1
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument arg1

% testopt -c foo arg1
aflag = 0, bflag = 0, cvalue = foo
Non-option argument arg1

% testopt -a -- -b
aflag = 1, bflag = 0, cvalue = (null)
```

```
Non-option argument -b

% testopt -a -
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument -
```

14.2.3 Parsing Long Options with `getopt_long`

To accept GNU-style long options as well as single-character options, use `getopt_long` instead of `getopt`. This function is declared in '`getopt.h`', not '`unistd.h`'. You should make every program accept long options if it uses any options, because this takes little extra work and helps beginners remember how to use the program.

struct option Data Type

This structure describes a single long option name for the sake of `getopt_long`. The argument *longopts* must be an array of these structures, one for each long option. Terminate the array with an element containing all zeros.

The `struct option` structure has these fields:

`const char *name`
: This field is the name of the option. It is a string.

`int has_arg`
: This field says whether the option takes an argument. It is an integer, and there are three legitimate values: `no_argument`, `required_argument` and `optional_argument`.

`int *flag`
`int val`
: These fields control how to report or act on the option when it occurs.

  If `flag` is a null pointer, then the `val` is a value that identifies this option. Often these values are chosen to uniquely identify particular long options.

  If `flag` is not a null pointer, it should be the address of an `int` variable that is the flag for this option. The value in `val` is the value to store in the flag to indicate that the option was seen.

`int` **getopt_long** (`int` *argc*, `char *const *`*argv*, `const char *`*shortopts*, `const struct option *`*longopts*, `int *`*indexptr*) Function

Decode options from the vector *argv* (whose length is *argc*). The argument *shortopts* describes the short options to accept, just as it does in `getopt`. The argument *longopts* describes the long options to accept (see above).

When `getopt_long` encounters a short option, it does the same thing that `getopt` would do—it returns the character code for the option, and stores the options argument (if it has one) in `optarg`.

When `getopt_long` encounters a long option, it takes actions based on the `flag` and `val` fields of the definition of that option.

If `flag` is a null pointer, then `getopt_long` returns the contents of `val` to indicate which option it found. You should arrange distinct values in the `val` field for options with different meanings, so you can decode these values after `getopt_long` returns. If the long option is equivalent to a short option, you can use the short option's character code in `val`.

If `flag` is not a null pointer, that means this option should just set a flag in the program. The flag is a variable of type `int` that you define. Put the address of the flag in the `flag` field. Put in the `val` field the value you would like this option to store in the flag. In this case, `getopt_long` returns `0`.

For any long option, `getopt_long` tells you the index in the array *longopts* of the options definition, by storing it into **indexptr*. You can get the name of the option with *longopts*`[`**indexptr*`].name`. So you can distinguish among long options either by the values in their `val` fields or by their indices. You can also distinguish in this way among long options that set flags.

When a long option has an argument, `getopt_long` puts the argument value in the variable `optarg` before returning. When the option has no argument, the value in `optarg` is a null pointer. This is how you can tell whether an optional argument was supplied.

When `getopt_long` has no more options to handle, it returns `-1` and leaves, in the variable `optind`, the index in *argv* of the next remaining argument.

Since long option names were used before before the `getopt_long` options were invented, there are program interfaces that require programs to recognize options like '`-option value`' instead of '`--option value`'. To enable these programs to use the GNU getopt functionality, there is one more function available.

`int` **getopt_long_only** (`int` *argc*, `char *const *`*argv*, `const char *`*shortopts*, `const struct option *`*longopts*, `int *`*indexptr*) — Function

The `getopt_long_only` function is equivalent to the `getopt_long` function, but it allows the user of the application to specify passing long options with only '`-`' instead of '`--`'. The '`--`' prefix is still recognized, but instead of looking through the short options if a '`-`' is seen, it first determines whether this parameter names a long option. If not, it is parsed as a short option.

Assuming `getopt_long_only` is used starting an application with:

```
app -foo
```

the `getopt_long_only` will first look for a long option named '`foo`'. If this is not found, the short options '`f`', '`o`', and again '`o`' are recognized.

14.2.4 Example of Parsing Long Options with `getopt_long`

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <getopt.h>

/* Flag set by '--verbose'. */
static int verbose_flag;

int
main (argc, argv)
     int argc;
     char **argv;
{
  int c;

  while (1)
    {
      static struct option long_options[] =
        {
          /* These options set a flag. */
          {"verbose", no_argument,       &verbose_flag, 1},
          {"brief",   no_argument,       &verbose_flag, 0},
          /* These options don't set a flag.
     We distinguish them by their indices. */
          {"add",     no_argument,       0, 'a'},
          {"append",  no_argument,       0, 'b'},
          {"delete",  required_argument, 0, 'd'},
          {"create",  required_argument, 0, 'c'},
          {"file",    required_argument, 0, 'f'},
          {0, 0, 0, 0}
        };
      /* getopt_long stores the option index here. */
      int option_index = 0;

      c = getopt_long (argc, argv, "abc:d:f:",
                       long_options, &option_index);

      /* Detect the end of the options. */
      if (c == -1)
        break;

      switch (c)
        {
        case 0:
          /* If this option set a flag, do nothing else now. */
          if (long_options[option_index].flag != 0)
            break;
```

```
          printf ("option %s", long_options[option_index].name);
          if (optarg)
            printf (" with arg %s", optarg);
          printf ("\n");
          break;

        case 'a':
          puts ("option -a\n");
          break;

        case 'b':
          puts ("option -b\n");
          break;

        case 'c':
          printf ("option -c with value `%s'\n", optarg);
          break;

        case 'd':
          printf ("option -d with value `%s'\n", optarg);
          break;

        case 'f':
          printf ("option -f with value `%s'\n", optarg);
          break;

        case '?':
          /* getopt_long already printed an error message. */
          break;

        default:
          abort ();
        }
    }

/* Instead of reporting `--verbose'
and `--brief' as they are encountered,
we report the final status resulting from them. */
if (verbose_flag)
  puts ("verbose flag is set");

/* Print any remaining command-line arguments (not options). */
if (optind < argc)
  {
```

```
          printf ("non-option ARGV-elements: ");
          while (optind < argc)
            printf ("%s ", argv[optind++]);
          putchar ('\n');
        }

      exit (0);
    }
```

14.3 Parsing Program Options with Argp

Argp is an interface for parsing Unix-style argument vectors (see Section 14.1 [Program Arguments], page 379).

Argp provides features unavailable in the more commonly used `getopt` interface. These features include automatically producing output in response to the '`--help`' and '`--version`' options, as described in the GNU coding standards. Using argp makes it less likely that programmers will neglect to implement these additional options or keep them up to date.

Argp also provides the ability to merge several independently defined option parsers into one, mediating conflicts between them and making the result appear seamless. A library can export an argp option parser that user programs might employ in conjunction with their own option parsers, resulting in less work for the user programs. Some programs may use only argument parsers exported by libraries, thereby achieving consistent and efficient option-parsing for abstractions implemented by the libraries.

The header file '`<argp.h>`' should be included to use argp.

14.3.1 The `argp_parse` Function

The main interface to argp is the `argp_parse` function. In many cases, calling `argp_parse` is the only argument-parsing code needed in `main` (see Section 14.1 [Program Arguments], page 379).

`error_t` **argp_parse** (`const struct argp` **argp*`, int` *argc*`, char` ***argv*`, unsigned` *flags*`, int` **arg_index*`, void` **input*) Function

The `argp_parse` function parses the arguments in *argv*, of length *argc*, using the argp parser *argp* (see Section 14.3.3 [Specifying Argp Parsers], page 391).

A value of zero is the same as a `struct argp` containing all zeros. *flags* is a set of flag bits that modify the parsing behavior (see Section 14.3.7 [Flags for `argp_parse`], page 401). *input* is passed through to the argp parser *argp*, and has meaning defined by *argp*. A typical usage is to pass a pointer to a structure that is used for specifying parameters to the parser and passing back the results.

Unless the ARGP_NO_EXIT or ARGP_NO_HELP flags are included in *flags*, calling argp_parse may result in the program exiting. This behavior is true if an error is detected, or when an unknown option is encountered (see Section 14.6 [Program Termination], page 425).

If *arg_index* is nonnull, the index of the first unparsed option in *argv* is returned as a value.

The return value is zero for successful parsing, or an error code (see Section 2.2 [Error Codes], page 18) if an error is detected. Different argp parsers may return arbitrary error codes, but the standard error codes are: ENOMEM if a memory allocation error occurred, or EINVAL if an unknown option or option argument is encountered.

14.3.2 Argp Global Variables

These variables make it easy for user programs to implement the '--version' option and provide a bug-reporting address in the '--help' output. These are implemented in argp by default.

const char * **argp_program_version** Variable

If defined or set by the user program to a nonzero value, then a '--version' option is added when parsing with argp_parse, which will print the '--version' string followed by a newline and exit. The exception to this is if the ARGP_NO_EXIT flag is used.

const char * **argp_program_bug_address** Variable

If defined or set by the user program to a nonzero value, argp_program_bug_address should point to a string that will be printed at the end of the standard output for the '--help' option, embedded in a sentence that says 'Report bugs to *address*.'.

argp_program_version_hook Variable

If defined or set by the user program to a nonzero value, a '--version' option is added when parsing with arg_parse, which prints the program version and exits with a status of zero. This is not the case if the ARGP_NO_HELP flag is used. If the ARGP_NO_EXIT flag is set, the exit behavior of the program is suppressed or modified, as when the argp parser is going to be used by other programs.

It should point to a function with this type of signature:

```
void print-version (FILE *stream, struct argp_state *state)
```

See Section 14.3.5.3 [Argp Parsing State], page 399, for an explanation of *state*.

This variable takes precedence over argp_program_version, and is useful if a program has version information not easily expressed in a simple string.

`error_t` **argp_err_exit_status** Variable

This is the exit status used when argp exits due to a parsing error. If not defined or set by the user program, this defaults to `EX_USAGE` from '`<sysexits.h>`'.

14.3.3 Specifying Argp Parsers

The first argument to the `argp_parse` function is a pointer to a `struct argp`, which is known as an *argp parser*:

struct argp Data Type

This structure specifies how to parse a given set of options and arguments, perhaps in conjunction with other argp parsers. It has the following fields:

`const struct argp_option *options`

This is a pointer to a vector of `argp_option` structures specifying which options this argp parser understands; it may be zero if there are no options at all (see Section 14.3.4 [Specifying Options in an Argp Parser], page 392).

`argp_parser_t parser`

This is a pointer to a function that defines actions for this parser; it is called for each option parsed, and at other well-defined points in the parsing process. A value of zero is the same as a pointer to a function that always returns `ARGP_ERR_UNKNOWN` (see Section 14.3.5 [Argp Parser Functions], page 394).

`const char *args_doc`

If nonzero, this is a string describing what non-option arguments are called by this parser. This is only used to print the '`Usage:`' message. If it contains newlines, the strings separated by them are considered alternative usage patterns and printed on separate lines. Lines after the first are prefixed by '`or:`' instead of '`Usage:`'.

`const char *doc`

If nonzero, this is a string containing extra text to be printed before and after the options in a long help message, with the two sections separated by a vertical tab (`'\v'`, `'\013'`) character. By convention, the documentation before the options is just a short string explaining what the program does. Documentation printed after the options describe behavior in more detail.

`const struct argp_child *children`

This is a pointer to a vector of `argp_children` structures. This pointer specifies which additional argp parsers should be combined with this one (see Section 14.3.6 [Combining Multiple Argp Parsers], page 400).

`char *(*help_filter)(int key, const char *text, void *input)`
: If nonzero, this is a pointer to a function that filters the output of help messages (see Section 14.3.8 [Customizing Argp Help Output], page 402).

`const char *argp_domain`
: If nonzero, the strings used in the argp library are translated using the domain described by this string. If zero, the current default domain is used.

Of the above group, `options`, `parser`, `args_doc` and the `doc` fields are usually all that are needed. If an argp parser is defined as an initialized C variable, only the fields used need be specified in the initializer. The rest will default to zero due to the way C structure initialization works. This design is exploited in most argp structures; the most-used fields are grouped near the beginning and the unused fields are left unspecified.

14.3.4 Specifying Options in an Argp Parser

The `options` field in a `struct argp` points to a vector of `struct argp_option` structures, each of which specifies an option that the argp parser supports. Multiple entries may be used for a single option provided it has multiple names. This should be terminated by an entry with zero in all fields. Note that when using an initialized C array for options, writing `{ 0 }` is enough to achieve this.

struct argp_option Data Type

This structure specifies a single option that an argp parser understands, as well as how to parse and document that option. It has the following fields:

`const char *name`
: The long name for this option, corresponding to the long option '*--name*'; this field may be zero if this option *only* has a short name. To specify multiple names for an option, additional entries may follow this one, with the `OPTION_ALIAS` flag set. See Section 14.3.4.1 [Flags for Argp Options], page 393.

`int key`
: The integer key provided by the current option to the option parser. If *key* has a value that is a printable ASCII character (i.e., `isascii (key)` is true), it *also* specifies a short option '*-char*', where *char* is the ASCII character with the code *key*.

`const char *arg`
: If non-zero, this is the name of an argument associated with this option, which must be provided (e.g., with the '*--name=value*' or '*-char value*' syntaxes), unless the `OPTION_ARG_OPTIONAL` flag (see Section 14.3.4.1 [Flags for Argp Options], page 393) is set, in which case it *may* be provided.

`int flags`
: Flags associated with this option, some of which are referred to above. See Section 14.3.4.1 [Flags for Argp Options], page 393.

`const char *doc`
: A documentation string for this option, for printing in help messages.

  If both the `name` and `key` fields are zero, this string will be printed tabbed left from the normal option column, making it useful as a group header. This will be the first thing printed in its group. In this usage, it's conventional to end the string with a '`:`' character.

`int group`
: Group identity for this option.

  In a long help message, options are sorted alphabetically within each group, and the groups presented in the order 0, 1, 2, . . . , *n*, −*m*, . . . , −2, −1.

  Every entry in an options array with this field 0 will inherit the group number of the previous entry, or zero if it's the first one. If it's a group header with `name` and `key` fields both zero, the previous entry + 1 is the default. Automagic options such as '`--help`' are put into group −1.

  Note that because of C structure initialization rules, this field often need not be specified, because 0 is the correct value.

14.3.4.1 Flags for Argp Options

The following flags may be ORed together in the `flags` field of a `struct argp_option`. These flags control various aspects of how that option is parsed or displayed in help messages:

`OPTION_ARG_OPTIONAL`
: The argument associated with this option is optional.

`OPTION_HIDDEN`
: This option isn't displayed in any help messages.

`OPTION_ALIAS`
: This option is an alias for the closest previous non-alias option. This means that it will be displayed in the same help entry and will inherit fields other than `name` and `key` from the option being aliased.

`OPTION_DOC`
: This option isn't actually an option and should be ignored by the actual option parser. It is an arbitrary section of documentation that should be displayed in much the same manner as the options. This is known as a *documentation option*.

If this flag is set, then the option `name` field is displayed unmodified (e.g., no '`--`' prefix is added) at the left margin where a *short* option would normally be displayed, and this documentation string is left in its usual place. For purposes of sorting, any leading white space and punctuation is ignored, unless the first non-white-space character is '`-`'. This entry is displayed after all options, after `OPTION_DOC` entries with a leading '`-`', in the same group.

`OPTION_NO_USAGE`
: This option shouldn't be included in long usage messages, but should still be included in other help messages. This is intended for options that are completely documented in an argp's `args_doc` field (see Section 14.3.3 [Specifying Argp Parsers], page 391). Including this option in the generic usage list would be redundant, and should be avoided.

  For instance, if `args_doc` is '`FOO BAR\n-x BLAH`', and the '`-x`' option's purpose is to distinguish these two cases, '`-x`' should probably be marked `OPTION_NO_USAGE`.

14.3.5 Argp Parser Functions

The function pointed to by the `parser` field in a `struct argp` (see Section 14.3.3 [Specifying Argp Parsers], page 391) defines what actions take place in response to each option or argument parsed. It is also used as a hook, allowing a parser to perform tasks at certain other points during parsing.

Argp parser functions have the following type signature:

```
error_t parser (int key, char *arg, struct argp_state *state)
```

where the arguments are as follows:

key
: For each option that is parsed, *parser* is called with a value of *key* from that option's `key` field in the option vector (see Section 14.3.4 [Specifying Options in an Argp Parser], page 392). *parser* is also called at other times with special reserved keys, such as `ARGP_KEY_ARG` for non-option arguments (see Section 14.3.5.1 [Special Keys for Argp Parser Functions], page 395).

arg
: If *key* is an option, *arg* is its given value. This defaults to zero if no value is specified. Only options that have a nonzero `arg` field can ever have a value. These must *always* have a value unless the `OPTION_ARG_OPTIONAL` flag is specified. If the input being parsed specifies a value for an option that doesn't allow one, an error results before *parser* ever gets called.

  If *key* is `ARGP_KEY_ARG`, *arg* is a non-option argument. Other special keys always have a zero *arg*.

state *state* points to a `struct argp_state`, containing useful information about the current parsing state for use by *parser* (see Section 14.3.5.3 [Argp Parsing State], page 399).

When *parser* is called, it should perform whatever action is appropriate for *key*, and return `0` for success, `ARGP_ERR_UNKNOWN` if the value of *key* is not handled by this parser function, or a Unix error code if a real error occurred (see Section 2.2 [Error Codes], page 18).

`int` **ARGP_ERR_UNKNOWN** Macro

Argp parser functions should return `ARGP_ERR_UNKNOWN` for any *key* value they do not recognize, or for non-option arguments (*key* == `ARGP_KEY_ARG`) that they are not equipped to handle.

A typical parser function uses a switch statement on *key*:

```
error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  switch (key)
    {
    case option_key:
      action
      break;
    ...
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}
```

14.3.5.1 Special Keys for Argp Parser Functions

In addition to key values corresponding to user options, the *key* argument to argp parser functions may have a number of other special values. In the following example, *arg* and *state* refer to parser function arguments (see Section 14.3.5 [Argp Parser Functions], page 394).

`ARGP_KEY_ARG`

This is not an option at all, but rather a command-line argument, whose value is pointed to by *arg*.

When there are multiple parser functions in play due to argp parsers being combined, it's impossible to know which one will handle a specific argument. Each is called until one returns 0 or an error other than `ARGP_ERR_UNKNOWN`; if an argument is not handled, `argp_parse` immediately returns success, without parsing any more arguments.

Once a parser function returns success for this key, that fact is recorded, and the ARGP_KEY_NO_ARGS case won't be used. *However*, if while processing the argument a parser function decrements the next field of its *state* argument, the option won't be considered processed; this is to allow you to actually modify the argument, perhaps into an option, and have it processed again.

ARGP_KEY_ARGS
: If a parser function returns ARGP_ERR_UNKNOWN for ARGP_KEY_ARG, it is immediately called again with the key ARGP_KEY_ARGS, which has a similar meaning, but is slightly more convenient for consuming all remaining arguments. *arg* is 0, and the tail of the argument vector may be found at *state*->argv + *state*->next. If success is returned for this key, and *state*->next is unchanged, all remaining arguments are considered to have been consumed. Otherwise, the amount by which *state*->next has been adjusted indicates how many were used. Here is an example that uses both, for different args:

```
...
case ARGP_KEY_ARG:
  if (state->arg_num == 0)
    /* First argument */
    first_arg = arg;
  else
    /* Let the next case parse it.  */
    return ARGP_KEY_UNKNOWN;
  break;
case ARGP_KEY_ARGS:
  remaining_args = state->argv + state->next;
  num_remaining_args = state->argc - state->next;
  break;
```

ARGP_KEY_END
: This indicates that there are no more command-line arguments. Parser functions are called in a different order, children first. This allows each parser to clean up its state for the parent.

ARGP_KEY_NO_ARGS
: Because it's common to do some special processing if there aren't any non-option args, parser functions are called with this key if they didn't successfully process any non-option arguments. This is called just before ARGP_KEY_END, where more general validity checks on previously parsed arguments take place.

ARGP_KEY_INIT
: This is passed in before any parsing is done. Afterwards, the values of each element of the child_input field of *state*, if any, are copied

to each child's state to be the initial value of the `input` when *their* parsers are called.

`ARGP_KEY_SUCCESS`
: This is passed in when parsing has successfully been completed, even if arguments remain.

`ARGP_KEY_ERROR`
: This is passed in if an error has occurred and parsing is terminated. In this case, a call with a key of `ARGP_KEY_SUCCESS` is never made.

`ARGP_KEY_FINI`
: This is the final key ever seen by any parser, even after `ARGP_KEY_SUCCESS` and `ARGP_KEY_ERROR`. Any resources allocated by `ARGP_KEY_INIT` may be freed here. At times, certain resources allocated are to be returned to the caller after a successful parse. In that case, those particular resources can be freed in the `ARGP_KEY_ERROR` case.

In all cases, `ARGP_KEY_INIT` is the first key seen by parser functions, and `ARGP_KEY_FINI` the last, unless an error was returned by the parser for `ARGP_KEY_INIT`. Other keys can occur in one the following orders. *opt* refers to an arbitrary option key:

opt... `ARGP_KEY_NO_ARGS` `ARGP_KEY_END` `ARGP_KEY_SUCCESS`
: The arguments being parsed did not contain any non-option arguments.

(*opt* | `ARGP_KEY_ARG`)... `ARGP_KEY_END` `ARGP_KEY_SUCCESS`
: All non-option arguments were successfully handled by a parser function. There may be multiple parser functions if multiple argp parsers were combined.

(*opt* | `ARGP_KEY_ARG`)... `ARGP_KEY_SUCCESS`
: Some non-option argument went unrecognized.

  This occurs when every parser function returns `ARGP_KEY_UNKNOWN` for an argument, in which case parsing stops at that argument if *arg_index* is a null pointer. Otherwise an error occurs.

In all cases, if a nonnull value for *arg_index* gets passed to `argp_parse`, the index of the first unparsed command-line argument is passed back in that value.

If an error occurs and is either detected by argp or because a parser function returned an error value, each parser is called with `ARGP_KEY_ERROR`. No further calls are made, except the final call with `ARGP_KEY_FINI`.

14.3.5.2 Functions for Use in Argp Parsers

Argp provides a number of functions available to the user of argp (see Section 14.3.5 [Argp Parser Functions], page 394), mostly for producing error mes-

sages. These take as their first argument the *state* argument to the parser function (see Section 14.3.5.3 [Argp Parsing State], page 399).

`void` **argp_usage** (`const struct argp_state *`*state*) Function

This outputs the standard usage message for the argp parser referred to by *state* to *state*`->err_stream` and terminates the program with `exit (argp_err_exit_status)` (see Section 14.3.2 [Argp Global Variables], page 390).

`void` **argp_error** (`const struct argp_state *`*state*`, const char *`*fmt*`, ...`) Function

This prints the printf format string *fmt* and following args, preceded by the program name and '`:`', and followed by a '`Try ... --help`' message, and terminates the program with an exit status of `argp_err_exit_status` (see Section 14.3.2 [Argp Global Variables], page 390).

`void` **argp_failure** (`const struct argp_state *`*state*`, int` *status*`, int` *errnum*`, const char *`*fmt*`, ...`) Function

Similar to the standard GNU error-reporting function `error`, this prints the program name and '`:`', the printf format string *fmt*, and the appropriate following args. If it is nonzero, the standard Unix error text for *errnum* is printed. If *status* is nonzero, it terminates the program with that value as its exit status.

The difference between `argp_failure` and `argp_error` is that `argp_error` is for *parsing errors*, whereas `argp_failure` is for other problems that occur during parsing but don't reflect a syntactic problem with the input, such as illegal values for options, bad phase of the moon, etc.

`void` **argp_state_help** (`const struct argp_state *`*state*`, FILE *`*stream*`, unsigned` *flags*) Function

This outputs a help message to *stream* for the argp parser referred to by *state*. The *flags* argument determines what sort of help message is produced (see Section 14.3.10 [Flags for the `argp_help` Function], page 404).

Error output is sent to *state*`->err_stream`, and the program name printed is *state*`->name`.

The output or program termination behavior of these functions may be suppressed if the `ARGP_NO_EXIT` or `ARGP_NO_ERRS` flags are passed to `argp_parse` (see Section 14.3.7 [Flags for `argp_parse`], page 401).

This behavior is useful if an argp parser is exported for use by other programs (e.g., by a library), and may be used in a context where it is not desirable to terminate the program in response to parsing errors. In argp parsers intended for such general use, and for the case where the program *doesn't* terminate, calls to any of these functions should be followed by code that returns the appropriate error code:

```
if (bad argument syntax)
  {
     argp_usage (state);
     return EINVAL;
  }
```

If a parser function will *only* be used when `ARGP_NO_EXIT` is not set, the return may be omitted.

14.3.5.3 Argp Parsing State

The third argument to argp parser functions (see Section 14.3.5 [Argp Parser Functions], page 394) is a pointer to a `struct argp_state`, which contains information about the state of the option parsing.

struct argp_state Data Type

This structure has the following fields, which may be modified as noted:

`const struct argp *const root_argp`
: This is the top-level argp parser being parsed. This is often *not* the same `struct argp` passed into `argp_parse` by the invoking program (see Section 14.3 [Parsing Program Options with Argp], page 389). It is an internal argp parser that contains options implemented by `argp_parse` itself, such as '`--help`'.

`int argc`
`char **argv`
: This is the argument vector being parsed. This may be modified.

`int next`
: This is the index in `argv` of the next argument to be parsed. This may be modified.

  One way to consume all remaining arguments in the input is to set *state*`->next` = *state*`->argc`, perhaps after recording the value of the `next` field to find the consumed arguments. The current option can be reparsed immediately by decrementing this field, then modifying *state*`->argv[`*state*`->next]` to reflect the option that should be reexamined.

`unsigned flags`
: The flags supplied to `argp_parse`. These may be modified, although some flags may only take effect when `argp_parse` is first invoked (see Section 14.3.7 [Flags for `argp_parse`], page 401).

`unsigned arg_num`
: While calling a parsing function with the *key* argument `ARGP_KEY_ARG`, this represents the number of the current arg, starting at 0. It is incremented after each `ARGP_KEY_ARG` call returns. At all other times, this is the number of `ARGP_KEY_ARG` arguments that have been processed.

`int quoted`
: If nonzero, this is the index in `argv` of the first argument following a special '`--`' argument. This prevents anything that follows from being interpreted as an option. It is only set after argument parsing has proceeded past this point.

`void *input`
: This is an arbitrary pointer passed in from the caller of `argp_parse`, in the *input* argument.

`void **child_inputs`
: These are values that will be passed to child parsers. This vector will be the same length as the number of children in the current parser. Each child parser will be given the value of *state*`->child_inputs[`*i*`]` as *its state*`->input` field, where *i* is the index of the child in the this parser's `children` field (see Section 14.3.6 [Combining Multiple Argp Parsers], page 400).

`void *hook`
: This is for the parser function's use. It is initialized to 0, but otherwise ignored by argp.

`char *name`
: This is the name used when printing messages. This is initialized to `argv[0]`, or `program_invocation_name` if `argv[0]` is unavailable.

`FILE *err_stream`
`FILE *out_stream`
: These are the stdio streams used when argp prints. Error messages are printed to `err_stream` and all other output, such as '`--help`', is printed to `out_stream`. These are initialized to `stderr` and `stdout`, respectively (see Section 17.2 [Standard Streams], page 439).

`void *pstate`
: This is private, for use by the argp implementation.

14.3.6 Combining Multiple Argp Parsers

The `children` field in a `struct argp` enables other argp parsers to be combined with the referencing one for the parsing of a single set of arguments. This field should point to a vector of `struct argp_child`, which is terminated by an entry having a value of zero in the `argp` field.

Where conflicts between combined parsers arise, as when two specify an option with the same name, the parser conflicts are resolved in favor of the parent argp parser(s), or the earlier of the argp parsers in the list of children.

struct argp_child Data Type

An entry in the list of subsidiary argp parsers pointed to by the `children` field in a `struct argp`. The fields are as follows:

`const struct argp *argp`
: This is the child argp parser, or zero to end of the list.

`int flags`
: These are flags for this child.

`const char *header`
: If nonzero, this is an optional header to be printed within help output before the child options. As a side effect, a nonzero value forces the child options to be grouped together. To achieve this effect without actually printing a header string, use a value of `""`. As with header strings specified in an option entry, the conventional value of the last character is '`:`' (see Section 14.3.4 [Specifying Options in an Argp Parser], page 392).

`int group`
: This is where the child options are grouped relative to the other 'consolidated' options in the parent argp parser. The values are the same as the `group` field in `struct argp_option` (see Section 14.3.4 [Specifying Options in an Argp Parser], page 392). All child-groupings follow parent options at a particular group level. If both this field and `header` are zero, then the child's options aren't grouped together; they are merged with parent options at the parent option group level.

14.3.7 Flags for `argp_parse`

The default behavior of `argp_parse` is designed to be convenient for the most common case of parsing program command-line arguments. To modify these defaults, the following flags may be or'd together in the *flags* argument to `argp_parse`:

`ARGP_PARSE_ARGV0`
: Do not ignore the first element of the *argv* argument to `argp_parse`. Unless `ARGP_NO_ERRS` is set, the first element of the argument vector is skipped for option-parsing purposes, as it corresponds to the program name in a command line.

`ARGP_NO_ERRS`
: Do not print error messages for unknown options to `stderr`; unless this flag is set, `ARGP_PARSE_ARGV0` is ignored, as `argv[0]` is used as the program name in the error messages. This flag implies `ARGP_NO_EXIT`. This is based on the assumption that silent exiting upon errors is bad behavior.

`ARGP_NO_ARGS`
: Do not parse any non-option args. Normally, these are parsed by calling the parse functions with a key of `ARGP_KEY_ARG`, the actual argument being the value. This flag needn't normally be set, as the default behavior is to stop parsing as soon as an argument fails to be parsed (see Section 14.3.5 [Argp Parser Functions], page 394).

`ARGP_IN_ORDER`
: Parse options and arguments in the same order they occur on the command line. Normally, they are rearranged so that all options come first.

`ARGP_NO_HELP`
: Do not provide the standard long option '`--help`', which ordinarily causes usage and option help information to be output to `stdout` and `exit (0)`.

`ARGP_NO_EXIT`
: Do not exit on errors, although they may still result in error messages.

`ARGP_LONG_ONLY`
: Use the GNU getopt 'long-only' rules for parsing arguments. This allows long options to be recognized with only a single '`-`' (i.e. '`-help`'). This results in a less useful interface, and its use is discouraged as it conflicts with the way most GNU programs work as well as the GNU coding standards.

`ARGP_SILENT`
: Turns off any message-printing/exiting options, specifically `ARGP_NO_EXIT`, `ARGP_NO_ERRS`, and `ARGP_NO_HELP`.

14.3.8 Customizing Argp Help Output

The `help_filter` field in a `struct argp` is a pointer to a function that filters the text of help messages before displaying them. They have a function signature like:

```
char *help-filter (int key, const char *text, void *input)
```

Where *key* is either a key from an option, in which case *text* is that option's help text (see Section 14.3.4 [Specifying Options in an Argp Parser], page 392). Alternately, one of the special keys with names beginning with '`ARGP_KEY_HELP_`' might be used, describing which other help text *text* will contain (see Section 14.3.8.1 [Special Keys for Argp Help Filter Functions], page 403).

The function should return either *text* if it remains as-is, or a replacement string allocated using `malloc`. This will either be freed by argp or be zero, which prints nothing. The value of *text* is supplied *after* any translation has been done, so if any of the replacement text needs translation, it will be done by the filter function. *input* is either the input supplied to `argp_parse` or it is zero, if `argp_help` was called directly by the user.

14.3.8.1 Special Keys for Argp Help Filter Functions

The following special values may be passed to an argp help filter function as the first argument in addition to key values for user options. They specify which help text the *text* argument contains:

`ARGP_KEY_HELP_PRE_DOC`
: This is the help text preceding options.

`ARGP_KEY_HELP_POST_DOC`
: This is the help text following options.

`ARGP_KEY_HELP_HEADER`
: This is the option header string.

`ARGP_KEY_HELP_EXTRA`
: This is used after all other documentation; *text* is zero for this key.

`ARGP_KEY_HELP_DUP_ARGS_NOTE`
: This is the explanatory note printed when duplicate option arguments have been suppressed.

`ARGP_KEY_HELP_ARGS_DOC`
: This is the argument doc string; formally the `args_doc` field from the argp parser (see Section 14.3.3 [Specifying Argp Parsers], page 391).

14.3.9 The `argp_help` Function

Normally programs using argp need not be written with particular printing argument-usage-type help messages in mind, as the standard '`--help`' option is handled automatically by argp. Typical error cases can be handled using `argp_usage` and `argp_error` (see Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397). However, if it's desirable to print a help message in some context other than parsing the program options, argp offers the `argp_help` interface.

`void` **argp_help** (`const struct argp *`*argp*`, FILE *`*stream*`, unsigned` *flags*`, char *`*name*) Function

This outputs a help message for the argp parser *argp* to *stream*. The type of messages printed will be determined by *flags*.

Any options such as '`--help`' that are implemented automatically by argp itself will *not* be present in the help output; for this reason it is best to use `argp_state_help` if calling from within an argp parser function (see Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397).

14.3.10 Flags for the `argp_help` Function

When calling `argp_help` (see Section 14.3.9 [The `argp_help` Function], page 403) or `argp_state_help` (see Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397) the exact output is determined by the *flags* argument. This should consist of any of the following flags, or'd together:

`ARGP_HELP_USAGE`
: This is a Unix '`Usage:`' message that explicitly lists all options.

`ARGP_HELP_SHORT_USAGE`
: This is a Unix '`Usage:`' message that displays an appropriate placeholder to indicate where the options go; useful for showing the non-option argument syntax.

`ARGP_HELP_SEE`
: This is a '`Try ... for more help`' message; '`...`' contains the program name and '`--help`'.

`ARGP_HELP_LONG`
: This is a verbose option help message that gives each option available along with its documentation string.

`ARGP_HELP_PRE_DOC`
: This is the part of the argp parser doc string preceding the verbose option help.

`ARGP_HELP_POST_DOC`
: This is the part of the argp parser doc string following the verbose option help.

`ARGP_HELP_DOC`
: `(ARGP_HELP_PRE_DOC | ARGP_HELP_POST_DOC)`

`ARGP_HELP_BUG_ADDR`
: This is a message that prints where to report bugs for this program, if the `argp_program_bug_address` variable contains this information.

`ARGP_HELP_LONG_ONLY`
: This will modify any output to reflect the `ARGP_LONG_ONLY` mode.

The following flags are only understood when used with `argp_state_help`. They control whether the function returns after printing its output, or terminates the program:

`ARGP_HELP_EXIT_ERR`
: This will terminate the program with `exit (argp_err_exit_status)`.

`ARGP_HELP_EXIT_OK`
: This will terminate the program with `exit (0)`.

The following flags are combinations of the basic flags for printing standard messages:

`ARGP_HELP_STD_ERR`
: Assuming that an error message for a parsing error has printed, this prints a message on how to get help, and terminates the program with an error.

`ARGP_HELP_STD_USAGE`
: This prints a standard usage message and terminates the program with an error. This is used when no other specific error messages are appropriate or available.

`ARGP_HELP_STD_HELP`
: This prints the standard response for a '`--help`' option, and terminates the program successfully.

14.3.11 Argp Examples

These example programs demonstrate the basic usage of argp.

14.3.11.1 A Minimal Program Using Argp

This is perhaps the smallest program possible that uses argp. It won't do much except give an error messages and exit when there are any arguments, and prints a rather pointless message for '`--help`'.

```
/* Argp Example 1—a minimal program using argp */

/* This is (probably) the smallest possible program that
  uses argp. It won't do much except give an error
  message and exit when there are any arguments, and print
  a (rather pointless) message for –help. */

#include <argp.h>

int main (int argc, char **argv)
{
  argp_parse (0, argc, argv, 0, 0, 0);
  exit (0);
}
```

14.3.11.2 A Program Using Argp with Only Default Options

This program doesn't use any options or arguments; it uses argp to be compliant with the GNU standard command-line format.

In addition to giving no arguments and implementing a '`--help`' option, this example has a '`--version`' option, which will put the given documentation string and bug address in the '`--help`' output, as per GNU standards.

The variable `argp` contains the argument parser specification. Adding fields to this structure is the way most parameters are passed to `argp_parse`. The first three fields are normally used, but they are not in this small program. There are also two global variables that argp can use defined here, `argp_program_version` and `argp_program_bug_address`. They are considered global variables because they will almost always be constant for a given program, even if they use different argument parsers for various tasks.

```
/* Argp Example 2—a pretty minimal program using argp */

/* This program doesn't use any options or arguments, but uses
   argp to be compliant with the GNU standard command-line
   format.

   In addition to making sure no arguments are given, and
   implementing a –help option, this example will have a
   –version option, and will put the given documentation string
   and bug address in the –help output, as per GNU standards.

   The variable ARGP contains the argument parser specification;
   adding fields to this structure is the way most parameters are
   passed to argp_parse (the first three fields are usually used,
   but not in this small program).  There are also two global
   variables that argp knows about defined here,
   ARGP_PROGRAM_VERSION and ARGP_PROGRAM_BUG_ADDRESS (they are
   global variables because they will almost always be constant
   for a given program, even if it uses different argument
   parsers for various tasks). */

#include <argp.h>

const char *argp_program_version =
  "argp-ex2 1.0";
const char *argp_program_bug_address =
  "<bug-gnu-utils@gnu.org>";

/* Program documentation */
static char doc[] =
```

```
  "Argp Example 2---a pretty minimal program using argp";

/* This is our argument parser.  The options, parser, and
   args_doc fields are zero because we have neither options nor
   arguments; doc and argp_program_bug_address will be
   used in the output for '--help', and the '--version'
   option will print out argp_program_version. */
static struct argp argp = { 0, 0, 0, doc };

int main (int argc, char **argv)
{
  argp_parse (&argp, argc, argv, 0, 0, 0);
  exit (0);
}
```

14.3.11.3 A Program Using Argp with User Options

This program uses the same features as Example 2, adding user options and arguments.

We now use the first four fields in `argp` (see Section 14.3.3 [Specifying Argp Parsers], page 391) and specify `parse_opt` as the parser function (see Section 14.3.5 [Argp Parser Functions], page 394).

In this example, `main` uses a structure to communicate with the `parse_opt` function, a pointer to which it passes in the `input` argument to `argp_parse` (see Section 14.3 [Parsing Program Options with Argp], page 389). It is retrieved by `parse_opt` through the `input` field in its `state` argument (see Section 14.3.5.3 [Argp Parsing State], page 399). Of course, it's also possible to use global variables instead, but using a structure like this is somewhat more flexible and clean.

```
/* Argp Example 3—a program with options and arguments using argp */

/* This program uses the same features as Example 2, and uses options and
   arguments.

   We now use the first four fields in ARGP, so here's a description of them:
     OPTIONS: A pointer to a vector of struct argp_option (see below)
     PARSER: A function to parse a single option, called by argp
     ARGS_DOC: A string describing how the non-option arguments should look
     DOC: A descriptive string about this program; if it contains a
          vertical tab character (\v), the part after it will be
          printed *following* the options.

   The function PARSER takes the following arguments:
     KEY: An integer specifying which option this is (taken
```

```
        from the KEY field in each struct argp_option), or
        a special key specifying something else; the only
        special keys we use here are ARGP_KEY_ARG, meaning
        a non-option argument, and ARGP_KEY_END, meaning
        that all arguments have been parsed
      ARG: For an option KEY, the string value of its
        argument, or NULL if it has none
      STATE: A pointer to a struct argp_state, containing
          various useful information about the parsing state; used here
          are the INPUT field, which reflects the INPUT argument to
          argp_parse, and the ARG_NUM field, which is the number of the
          current non-option argument being parsed
    It should return either 0, meaning success, ARGP_ERR_UNKNOWN, meaning the
    given KEY wasn't recognized, or an errno value indicating some other
    error.

    The OPTIONS field contains a pointer to a vector of struct argp_option's;
    that structure has the following fields (if you assign your option
    structures using array initialization like this example, unspecified
    fields will be defaulted to 0, and need not be specified):
      NAME: The name of this option's long option (may be zero)
      KEY: The KEY to pass to the PARSER function when parsing this option,
        *and* the name of this option's short option, if it is a
        printable ASCII character
      ARG: The name of this option's argument, if any
      FLAGS: Flags describing this option; some of them are:
            OPTION_ARG_OPTIONAL: The argument to this option is optional
            OPTION_ALIAS: This option is an alias for the
                  previous option
            OPTION_HIDDEN: Don't show this option in –help output
      DOC: A documentation string for this option, shown in –help output

    An options vector should be terminated by an option with all fields zero. */

#include <argp.h>

const char *argp_program_version =
  "argp-ex3 1.0";
const char *argp_program_bug_address =
  "<bug-gnu-utils@gnu.org>";

/* Program documentation */
static char doc[] =
  "Argp Example 3---a program with options and arguments using argp";
```

```
/* A description of the arguments we accept */
static char args_doc[] = "ARG1 ARG2";

/* The options we understand */
static struct argp_option options[] = {
  {"verbose",  'v', 0,      0,  "Produce verbose output" },
  {"quiet",    'q', 0,      0,  "Don't produce any output" },
  {"silent",   's', 0,      OPTION_ALIAS },
  {"output",   'o', "FILE", 0,
   "Output to FILE instead of standard output" },
  { 0 }
};

/* Used by main to communicate with parse_opt */
struct arguments
{
  char *args[2];                /* arg1 & arg2 */
  int silent, verbose;
  char *output_file;
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  /* Get the input argument from argp_parse, which we
  know is a pointer to our arguments structure. */
  struct arguments *arguments = state->input;

  switch (key)
    {
    case 'q': case 's':
      arguments->silent = 1;
      break;
    case 'v':
      arguments->verbose = 1;
      break;
    case 'o':
      arguments->output_file = arg;
      break;

    case ARGP_KEY_ARG:
      if (state->arg_num >= 2)
```

```
        /* Too many arguments */
        argp_usage (state);

      arguments->args[state->arg_num] = arg;

      break;

    case ARGP_KEY_END:
      if (state->arg_num < 2)
        /* Not enough arguments */
        argp_usage (state);
      break;

    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

/* Our argp parser */
static struct argp argp = { options, parse_opt, args_doc, doc };

int main (int argc, char **argv)
{
  struct arguments arguments;

  /* Default values */
  arguments.silent = 0;
  arguments.verbose = 0;
  arguments.output_file = "-";

  /* Parse our arguments; every option seen by parse_opt will
  be reflected in arguments. */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  printf ("ARG1 = %s\nARG2 = %s\nOUTPUT_FILE = %s\n"
          "VERBOSE = %s\nSILENT = %s\n",
          arguments.args[0], arguments.args[1],
          arguments.output_file,
          arguments.verbose ? "yes" : "no",
          arguments.silent ? "yes" : "no");

  exit (0);
}
```

14.3.11.4 A Program Using Multiple Combined Argp Parsers

This program uses the same features as Example 3, but it has more options and presents more structure in the '`--help`' output. It also illustrates how you can "steal" the remainder of the input arguments past a certain point for programs that accept a list of items. It also illustrates the *key* value `ARGP_KEY_NO_ARGS`, which is only given if no non-option arguments were supplied to the program (see Section 14.3.5.1 [Special Keys for Argp Parser Functions], page 395).

For structuring help output, two features are used: *headers* and a two-part option string. The *headers* are entries in the options vector (see Section 14.3.4 [Specifying Options in an Argp Parser], page 392). The first four fields are zero. The two-part documentation string is in the variable `doc`, which allows documentation both before and after the options (see Section 14.3.3 [Specifying Argp Parsers], page 391). The two parts of `doc` are separated by a vertical-tab character (`'\v'`, or `'\013'`). By convention, the documentation before the options is a short string stating what the program does. After any options, it is longer, describing the behavior in more detail. All documentation strings are automatically filled for output, although newlines may be included to force a line break at a particular point. In addition, documentation strings are passed to the `gettext` function, for possible translation into the current locale.

```
/* Argp Example 4—a program with somewhat more complicated options */

/* This program uses the same features as Example 3, but it has more
   options and somewhat more structure in the –help output.  It
   also shows how you can "steal" the remainder of the input
   arguments past a certain point, for programs that accept a
   list of items.  It also shows the special argp KEY value
   ARGP_KEY_NO_ARGS, which is only given if no non-option
   arguments were supplied to the program.

   For structuring the help output, two features are used:
   *headers*, which are entries in the options vector with the
   first four fields being zero, and a two-part documentation
   string (in the variable DOC), which allows documentation both
   before and after the options; the two parts of DOC are
   separated by a vertical-tab character ('\v', or '\013').  By
   convention, the documentation before the options is just a
   short string saying what the program does, and afterwards it
   is longer, describing the behavior in more detail.  All
   documentation strings are automatically filled for output,
   although newlines may be included to force a line break at a
   particular point.  All documentation strings are also passed to
   the 'gettext' function, for possible translation into the
```

```
  current locale. */

#include <stdlib.h>
#include <error.h>
#include <argp.h>

const char *argp_program_version =
  "argp-ex4 1.0";
const char *argp_program_bug_address =
  "<bug-gnu-utils@prep.ai.mit.edu>";

/* Program documentation */
static char doc[] =
  "Argp Example 4---a program with somewhat more complicated\
options\
\vThis part of the documentation comes *after* the options;\
 note that the text is automatically filled, but it's possible\
 to force a line-break, e.g.\n<-- here.";

/* A description of the arguments we accept */
static char args_doc[] = "ARG1 [STRING...]";

/* Keys for options without short options */
#define OPT_ABORT  1            /* --abort */

/* The options we understand */
static struct argp_option options[] = {
  {"verbose",  'v', 0,       0, "Produce verbose output" },
  {"quiet",    'q', 0,       0, "Don't produce any output" },
  {"silent",   's', 0,       OPTION_ALIAS },
  {"output",   'o', "FILE",  0,
   "Output to FILE instead of standard output" },

  {0,0,0,0, "The following options should be grouped together:" },
  {"repeat",   'r', "COUNT", OPTION_ARG_OPTIONAL,
   "Repeat the output COUNT (default 10) times"},
  {"abort",    OPT_ABORT, 0, 0, "Abort before showing any output"},

  { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
```

```
  char *arg1;                     /* arg1 */
  char **strings;                 /* [string...] */
  int silent, verbose, abort;     /* '-s', '-v', '--abort' */
  char *output_file;              /* file arg to '--output' */
  int repeat_count;               /* count arg to '--repeat' */
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
  struct arguments *arguments = state->input;

  switch (key)
    {
    case 'q': case 's':
      arguments->silent = 1;
      break;
    case 'v':
      arguments->verbose = 1;
      break;
    case 'o':
      arguments->output_file = arg;
      break;
    case 'r':
      arguments->repeat_count = arg ? atoi (arg) : 10;
      break;
    case OPT_ABORT:
      arguments->abort = 1;
      break;

    case ARGP_KEY_NO_ARGS:
      argp_usage (state);

    case ARGP_KEY_ARG:
      /* Here we know that state->arg_num == 0, since we
     force argument parsing to end before any more arguments can
     get here. */
      arguments->arg1 = arg;

      /* Now we consume all the rest of the arguments.
     state->next is the index in state->argv of the
```

```
      next argument to be parsed, which is the first string
      we're interested in, so we can just use
      &state->argv[state->next] as the value for
      arguments->strings.

      In addition, by setting state->next to the end
      of the arguments, we can force argp to stop parsing here and
      return. */
        arguments->strings = &state->argv[state->next];
        state->next = state->argc;

        break;

      default:
        return ARGP_ERR_UNKNOWN;
      }
    return 0;
}

/* Our argp parser */
static struct argp argp = { options, parse_opt, args_doc, doc };

int main (int argc, char **argv)
{
  int i, j;
  struct arguments arguments;

  /* Default values */
  arguments.silent = 0;
  arguments.verbose = 0;
  arguments.output_file = "-";
  arguments.repeat_count = 1;
  arguments.abort = 0;

  /* Parse our arguments; every option seen by parse_opt will be
  reflected in arguments. */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  if (arguments.abort)
    error (10, 0, "ABORTED");

  for (i = 0; i < arguments.repeat_count; i++)
    {
      printf ("ARG1 = %s\n", arguments.arg1);
```

```
          printf ("STRINGS = ");
          for (j = 0; arguments.strings[j]; j++)
            printf (j == 0 ? "%s" : ", %s", arguments.strings[j]);
          printf ("\n");
          printf ("OUTPUT_FILE = %s\nVERBOSE = %s\nSILENT = %s\n",
                  arguments.output_file,
                  arguments.verbose ? "yes" : "no",
                  arguments.silent ? "yes" : "no");
        }

      exit (0);
    }
```

14.3.12 Argp User Customization

The formatting of argp '`--help`' output may be controlled to some extent by a program's users, by setting the `ARGP_HELP_FMT` environment variable to a comma-separated list of tokens. White space is ignored:

'`dup-args`'
'`no-dup-args`'
: These turn *duplicate-argument-mode* on or off. In duplicate argument mode, if an option that accepts an argument has multiple names, the argument is shown for each name. Otherwise, it is only shown for the first long option. A note is subsequently printed so the user knows that it applies to other names as well. The default is '`no-dup-args`', which is less consistent, but prettier.

'`dup-args-note`'
'`no-dup-args-note`'
: These will enable or disable the note informing the user of suppressed option argument duplication. The default is '`dup-args-note`'.

'`short-opt-col=`*n*'
: This prints the first short option in column *n*. The default is 2.

'`long-opt-col=`*n*'
: This prints the first long option in column *n*. The default is 6.

'`doc-opt-col=`*n*'
: This prints 'documentation options' (see Section 14.3.4.1 [Flags for Argp Options], page 393) in column *n*. The default is 2.

'`opt-doc-col=`*n*'
: This prints the documentation for options starting in column *n*. The default is 29.

‘header-col=*n*’
: This will indent the group headers that document groups of options to column *n*. The default is 1.

‘usage-indent=*n*’
: This will indent continuation lines in ‘Usage:’ messages to column *n*. The default is 12.

‘rmargin=*n*’
: This will word wrap help output at or before column *n*. The default is 79.

14.3.12.1 Parsing of Suboptions

Having a single level of options is sometimes not enough. There might be too many options that have to be available or a set of options that are closely related.

For these cases, some programs use suboptions. One of the most prominent programs is certainly mount(8). The -o option takes one argument, which itself is a comma-separated list of options. To ease the programming of code like this, the function getsubopt is available.

int **getsubopt** (char **optionp, const char* const *tokens, char **valuep) — Function

The *optionp* parameter must be a pointer to a variable containing the address of the string to process. When the function returns, the reference is updated to point to the next suboption or to the terminating ‘\0’ character if there are no more suboptions available.

The *tokens* parameter references an array of strings containing the known suboptions. All strings must be ‘\0’ terminated and, to mark the end, a null pointer must be stored. When getsubopt finds a possible legal suboption, it compares it with all strings available in the *tokens* array and returns the index in the string as the indicator.

In case the suboption has an associated value introduced by an ‘=’ character, a pointer to the value is returned in *valuep*. The string is ‘\0’ terminated. If no argument is available, *valuep* is set to the null pointer. By doing this, the caller can check whether a necessary value is given or whether no unexpected value is present.

In case the next suboption in the string is not mentioned in the *tokens* array, the starting address of the suboption including a possible value is returned in *valuep*, and the return value of the function is ‘-1’.

14.3.13 Parsing of Suboptions Example

The code that might appear in the mount(8) program is a perfect example of the use of getsubopt:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int do_all;
const char *type;
int read_size;
int write_size;
int read_only;

enum
{
  RO_OPTION = 0,
  RW_OPTION,
  READ_SIZE_OPTION,
  WRITE_SIZE_OPTION,
  THE_END
};

const char *mount_opts[] =
{
  [RO_OPTION] = "ro",
  [RW_OPTION] = "rw",
  [READ_SIZE_OPTION] = "rsize",
  [WRITE_SIZE_OPTION] = "wsize",
  [THE_END] = NULL
};

int
main (int argc, char *argv[])
{
  char *subopts, *value;
  int opt;

  while ((opt = getopt (argc, argv, "at:o:")) != -1)
    switch (opt)
      {
      case 'a':
        do_all = 1;
        break;
      case 't':
        type = optarg;
        break;
      case 'o':
```

```
          subopts = optarg;
          while (*subopts != '\0')
            switch (getsubopt (&subopts, mount_opts, &value))
              {
              case RO_OPTION:
                read_only = 1;
                break;
              case RW_OPTION:
                read_only = 0;
                break;
              case READ_SIZE_OPTION:
                if (value == NULL)
                  abort ();
                read_size = atoi (value);
                break;
              case WRITE_SIZE_OPTION:
                if (value == NULL)
                  abort ();
                write_size = atoi (value);
                break;
              default:
                /* Unknown suboption */
                printf ("Unknown suboption `%s'\n", value);
                break;
              }
            break;
          default:
            abort ();
          }

        /* Do the real work.  */

        return 0;
      }
```

14.4 Environment Variables

When a program is executed, it receives information about the context in which it was invoked in two ways. The first mechanism uses the *argv* and *argc* arguments to its `main` function, and is discussed in Section 14.1 [Program Arguments], page 379. The second mechanism uses *environment variables* and is discussed in this section.

The *argv* mechanism is typically used to pass command-line arguments specific to the particular program being invoked. The environment, on the other hand, keeps track of information that is shared by many programs, changes infrequently and is less frequently used.

The environment variables discussed in this section are the same environment variables that you set using assignments and the `export` command in the shell. Programs executed from the shell inherit all of the environment variables from the shell.

Standard environment variables are used for information about the user's home directory, terminal type, current locale and so on; you can define additional variables for other purposes. The set of all environment variables that have values is collectively known as the *environment*.

Names of environment variables are case-sensitive and must not contain the character '='. System-defined environment variables are invariably uppercase.

The values of environment variables can be anything that can be represented as a string. A value must not contain an embedded null character, since this is assumed to terminate the string.

14.4.1 Environment Access

The value of an environment variable can be accessed with the `getenv` function. This is declared in the header file '`stdlib.h`'. All of the following functions can be safely used in multithreaded programs. It is ensured that concurrent modifications to the environment do not lead to errors.

`char *` **getenv** (`const char *`*name*) Function

This function returns a string that is the value of the environment variable *name*. You must not modify this string. In some non-Unix systems not using the GNU library, it might be overwritten by subsequent calls to `getenv` (but not by any other library function). If the environment variable *name* is not defined, the value is a null pointer.

`int` **putenv** (`char *`*string*) Function

The `putenv` function adds or removes definitions from the environment. If the *string* is of the form '*name*=*value*', the definition is added to the environment. Otherwise, the *string* is interpreted as the name of an environment variable, and any definition for this variable in the environment is removed.

The difference to the `setenv` function is that the exact string given as the parameter *string* is put into the environment. If the user should change the string after the `putenv` call, this will reflect automatically in the environment. This also requires that *string* not be an automatic variable that could be deleted or could fall out of scope before the variable is removed from the environment. The same applies to dynamically allocated variables that are freed later.

This function is part of the extended Unix interface. Since it was also available in old SVID libraries, you should define either *_XOPEN_SOURCE* or *_SVID_SOURCE* before including any header.

`int` **setenv** (`const char *`*name*`, const char *`*value*`, int` *replace*) Function

The `setenv` function can be used to add a new definition to the environment. The entry with the name *name* is replaced by the value '*name*=*value*'. This is also true if *value* is the empty string. To do this, a new string is created and the string's *name* and *value* are copied. A null pointer for the *value* parameter is illegal. If the environment already contains an entry with key *name*, then the *replace* parameter controls the action. If replace is zero, nothing happens. Otherwise, the old entry is replaced by the new one.

You cannot remove an entry completely using this function.

This function was originally part of the BSD library, but is now part of the Unix standard.

`int` **unsetenv** (`const char *`*name*) Function

Using this function, you can remove an entry completely from the environment. If the environment contains an entry with the key *name*, this whole entry is removed. A call to this function is equivalent to a call to `putenv` when the *value* part of the string is empty.

The function returns `-1` if *name* is a null pointer, points to an empty string, or points to a string containing an '=' character. It returns `0` if the call succeeded.

This function was originally part of the BSD library but is now part of the Unix standard. The BSD version had no return value, though.

There is one more function to modify the whole environment. This function is said to be used in the POSIX.9 (POSIX bindings for Fortran 77) and so one would expect it to have made it to POSIX.1. But this never happened. We still provide this function as a GNU extension to enable writing standard compliant Fortran environments.

`int` **clearenv** (`void`) Function

The `clearenv` function removes all entries from the environment. Using `putenv` and `setenv`, new entries can be added again later.

If the function is successful, it returns `0`. Otherwise, the return value is nonzero.

You can deal directly with the underlying representation of environment objects to add more variables to the environment (for example, to communicate with another program you are about to execute).[2]

[2] Ibid., "Executing a File".

`char **` **environ** Variable

The environment is represented as an array of strings. Each string is of the format '*name*=*value*'. The order in which strings appear in the environment is not significant, but the same *name* must not appear more than once. The last element of the array is a null pointer.

This variable is declared in the header file '`unistd.h`'.

If you just want to get the value of an environment variable, use `getenv`.

Unix systems and the GNU system pass the initial value of `environ` as the third argument to `main` (see Section 14.1 [Program Arguments], page 379).

14.4.2 Standard Environment Variables

These environment variables have standard meanings. This doesn't mean that they are always present in the environment; but if these variables *are* present, they have these meanings. You shouldn't try to use these environment variable names for some other purpose.

`HOME`

This is a string representing the user's *home directory* or initial default working directory.

The user can set `HOME` to any value. If you need to make sure to obtain the proper home directory for a particular user, you should not use `HOME`; instead, look up the user's name in the user database.[3]

For most purposes, it is better to use `HOME`, precisely because this lets the user specify the value.

`LOGNAME`

This is the name that the user used to log in. Since the value in the environment can be tweaked arbitrarily, this is not a reliable way to identify the user who is running a program; a function like `getlogin` is better for that purpose.[4]

For most purposes, it is better to use `LOGNAME`, precisely because this lets the user specify the value.

`PATH`

A *path* is a sequence of directory names that is used for searching for a file. The variable `PATH` holds a path used for searching for programs to be run.

The `execlp` and `execvp` functions[5] use this environment variable, as do many shells and other utilities that are implemented in terms of those functions.

---

[3] Ibid., "User Database".

[4] Ibid., "Identifying Who Logged In".

[5] Ibid., "Executing a File".

The syntax of a path is a sequence of directory names separated by colons. An empty string instead of a directory name stands for the current directory.[6]

A typical value for this environment variable might be a string like:

```
:/bin:/etc:/usr/bin:/usr/new/X11:/usr/new:/usr/local/bin
```

This means that if the user tries to execute a program named `foo`, the system will look for files named '`foo`', '`/bin/foo`', '`/etc/foo`' and so on. The first of these files that exists is the one that is executed.

`TERM`

This specifies the kind of terminal that is receiving program output. Some programs can make use of this information to take advantage of special escape sequences or terminal modes supported by particular kinds of terminals. Many programs that use the termcap library[7] use the `TERM` environment variable, for example.

`TZ`

This specifies the time zone. See Section 10.4.7 [Specifying the Time Zone with TZ], page 306, for information about the format of this string and how it is used.

`LANG`

This specifies the default locale to use for attribute categories where neither `LC_ALL` nor the specific environment variable for that category is set. See Chapter 7 [Locales and Internationalization], page 181, for more information about locales.

`LC_ALL`

If this environment variable is set, it overrides the selection for all the locales done using the other `LC_*` environment variables. The value of the other `LC_*` environment variables is simply ignored in this case.

`LC_COLLATE`

This specifies what locale to use for string sorting.

`LC_CTYPE`

This specifies what locale to use for character sets and character classification.

`LC_MESSAGES`

This specifies what locale to use for printing messages and parsing responses.

---

[6] Ibid., "Working Directory".

[7] See Richard M. Stallman, "Finding a Terminal Description: tgetent" in *The Termcap Manual: The Termcap Library and Data Base*, 2nd ed. (Boston, MA: GNU Press, December 1992), `http://www.gnu.org/software/termutils/manual/termcap-1.3/termcap.html`.

`LC_MONETARY`
: This specifies what locale to use for formatting monetary values.

`LC_NUMERIC`
: This specifies what locale to use for formatting numbers.

`LC_TIME`
: This specifies what locale to use for formatting date and time values.

`NLSPATH`
: This specifies the directories in which the `catopen` function looks for message translation catalogs.

`_POSIX_OPTION_ORDER`
: If this environment variable is defined, it suppresses the usual reordering of command-line arguments by `getopt` and `argp_parse` (see Section 14.1.1 [Program Argument Syntax Conventions], page 380).

14.5 System Calls

A system call is a request for service that a program makes of the kernel. The service is generally something that only the kernel has the privilege to do, such as doing I/O. Programmers don't normally need to be concerned with system calls because there are functions in the GNU C Library to do virtually everything that system calls do. These functions work by making system calls themselves. For example, there is a system call that changes the permissions of a file, but you don't need to know about it because you can just use the GNU C Library's `chmod` function.

System calls are sometimes called kernel calls.

However, there are times when you want to make a system call explicitly, and for that, the GNU C Library provides the `syscall` function. `syscall` is harder to use and less portable than functions like `chmod`, but easier and more portable than coding the system call in assembler instructions.

`syscall` is most useful when you are working with a system call that is special to your system or is newer than the GNU C Library you are using. `syscall` is implemented in an entirely generic way; the function does not know anything about what a particular system call does or even if it is valid.

The description of `syscall` in this section assumes a certain protocol for system calls on the various platforms on which the GNU C Library runs. That protocol is not defined by any strong authority, but we won't describe it here either, because anyone who is coding `syscall` probably won't accept anything less than kernel and C library source code as a specification of the interface between them anyway.

`syscall` is declared in '`unistd.h`'.

`long int` **syscall** (`long int` *sysno*`, ...`) Function

`syscall` performs a generic system call.

sysno is the system call number. Each kind of system call is identified by a number. Macros for all the possible system call numbers are defined in 'sys/syscall.h'

The remaining arguments are the arguments for the system call, in order, and their meanings depend on the kind of system call. Each kind of system call has a definite number of arguments, from zero to five. If you code more arguments than the system call takes, the extra ones to the right are ignored.

The return value is the return value from the system call, unless the system call failed. In that case, `syscall` returns `-1` and sets `errno` to an error code that the system call returned. System calls do not return `-1` when they succeed.

If you specify an invalid *sysno*, `syscall` returns `-1` with `errno` = `ENOSYS`.

Here is an example:

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

...

int rc;

rc = syscall(SYS_chmod, "/etc/passwd", 0444);

if (rc == -1)
   fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

This, if all of the compatibility stars are aligned, is equivalent to the following preferable code:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

...

int rc;

rc = chmod("/etc/passwd", 0444);
if (rc == -1)
   fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

14.6 Program Termination

The usual way for a program to terminate is simply for its `main` function to return. The *exit status value* returned from the `main` function is used to report information back to the process's parent process or shell.

A program can also terminate normally by calling the `exit` function.

In addition, programs can be terminated by signals.[8] The `abort` function causes a signal that kills the program.

14.6.1 Normal Termination

A process terminates normally when its program signals it is done by calling `exit`. Returning from `main` is equivalent to calling `exit`, and the value that `main` returns is used as the argument to `exit`.

`void` **exit** (`int` *status*) Function

The `exit` function tells the system that the program is done, which causes it to terminate the process.

status is the program's exit status, which becomes part of the process's termination status. This function does not return.

Normal termination causes the following actions:

1. Functions that were registered with the `atexit` or `on_exit` functions are called in the reverse order of their registration. This mechanism allows your application to specify its own "clean-up" actions to be performed at program termination. Typically, this is used to do things like saving program state information in a file, or unlocking locks in shared data bases.
2. All open streams are closed, writing out any buffered output data (see Section 17.4 [Closing Streams], page 444). In addition, temporary files opened with the `tmpfile` function are removed.[9]
3. `_exit` is called, terminating the program (see Section 14.6.5 [Termination Internals], page 428).

14.6.2 Exit Status

When a program exits, it can return to the parent process a small amount of information about the cause of termination, using the *exit status*. This is a value between 0 and 255 that the exiting process passes as an argument to `exit`.

Normally, you should use the exit status to report very broad information about success or failure. You can't provide a lot of detail about the reasons for the failure, and most parent processes would not want much detail anyway.

---

[8] Ibid., "Signal Handling".

[9] Ibid., "Temporary Files".

There are conventions for what sorts of status values certain programs should return. The most common convention is simply 0 for success and 1 for failure. Programs that perform comparison use a different convention: they use status 1 to indicate a mismatch, and status 2 to indicate an inability to compare. Your program should follow an existing convention if an existing convention makes sense for it.

A general convention reserves status values 128 and up for special purposes. In particular, the value 128 is used to indicate failure to execute another program in a subprocess. This convention is not universally obeyed, but it is a good idea to follow it in your programs.

Warning: Don't try to use the number of errors as the exit status. This is actually not very useful; a parent process would generally not care how many errors occurred. Worse than that, it does not work, because the status value is truncated to 8 bits. Thus, if the program tried to report 256 errors, the parent would receive a report of 0 errors—that is, success.

For the same reason, it does not work to use the value of `errno` as the exit status—these can exceed 255.

Portability Note: Some non-POSIX systems use different conventions for exit status values. For greater portability, you can use the macros `EXIT_SUCCESS` and `EXIT_FAILURE` for the conventional status value for success and failure, respectively. They are declared in the file '`stdlib.h`'.

`int` **EXIT_SUCCESS** Macro

This macro can be used with the `exit` function to indicate successful program completion.

On POSIX systems, the value of this macro is `0`. On other systems, the value might be some other (possibly nonconstant) integer expression.

`int` **EXIT_FAILURE** Macro

This macro can be used with the `exit` function to indicate unsuccessful program completion in a general sense.

On POSIX systems, the value of this macro is `1`. On other systems, the value might be some other (possibly nonconstant) integer expression. Other nonzero status values also indicate failures. Certain programs use different nonzero status values to indicate particular kinds of "nonsuccess". For example, `diff` uses status value `1` to mean that the files are different, and `2` or more to mean that there was difficulty in opening the files.

Don't confuse a program's exit status with a process's termination status. There are lots of ways a process can terminate besides having its program finish. In the event that the process termination *is* caused by program termination (i.e., `exit`), though, the program's exit status becomes part of the process's termination status.

14.6.3 Clean-Ups on Exit

Your program can arrange to run its own clean-up functions if normal termination happens. If you are writing a library for use in various application programs, then

it is unreliable to insist that all applications call the library's clean-up functions explicitly before exiting. It is much more robust to make the clean-up invisible to the application, by setting up a clean-up function in the library itself using `atexit` or `on_exit`.

`int` **atexit** (`void` (`*`*function*) (`void`)) Function

The `atexit` function registers the function *function* to be called at normal program termination. The *function* is called with no arguments.

The return value from `atexit` is zero on success and nonzero if the function cannot be registered.

`int` **on_exit** (`void` (`*`*function*)(`int` *status*`,` `void` `*`*arg*)`,` `void` `*`*arg*) Function

This function is a somewhat more powerful variant of `atexit`. It accepts two arguments: a function *function* and an arbitrary pointer *arg*. At normal program termination, the *function* is called with two arguments: the *status* value passed to `exit`, and the *arg*.

This function is included in the GNU C Library only for compatibility for SunOS, and may not be supported by other implementations.

Here's a trivial program that illustrates the use of `exit` and `atexit`:

```
#include <stdio.h>
#include <stdlib.h>

void
bye (void)
{
  puts ("Goodbye, cruel world....");
}

int
main (void)
{
  atexit (bye);
  exit (EXIT_SUCCESS);
}
```

When this program is executed, it just prints the message and exits.

14.6.4 Aborting a Program

You can abort your program using the `abort` function. The prototype for this function is in '`stdlib.h`'.

`void` **abort** (`void`) Function

The `abort` function causes abnormal program termination. This does not execute clean-up functions registered with `atexit` or `on_exit`.

This function actually terminates the process by raising a `SIGABRT` signal, and your program can include a handler to intercept this signal.[10]

14.6.5 Termination Internals

The `_exit` function is the primitive used for process termination by `exit`. It is declared in the header file '`unistd.h`'.

`void` **_exit** (`int` *status*) Function

The `_exit` function is the primitive for causing a process to terminate with status *status*. Calling this function does not execute clean-up functions registered with `atexit` or `on_exit`.

`void` **_Exit** (`int` *status*) Function

The `_Exit` function is the ISO C equivalent to `_exit`. The ISO C committee members were not sure whether the definitions of `_exit` and `_Exit` were compatible, so they have not used the POSIX name.

This function was introduced in ISO C99 and is declared in '`stdlib.h`'.

When a process terminates for any reason—either because the program terminates, or as a result of a signal—the following things happen:

- All open file descriptors in the process are closed.[11] Streams are not flushed automatically when the process terminates (see Chapter 17 [Input/Output on Streams], page 439).
- A process exit status is saved to be reported back to the parent process via `wait` or `waitpid`.[12] If the program exited, this status includes the program exit status as its low-order 8 bits.
- Any child processes of the process being terminated are assigned a new parent process. On most systems, including GNU, this is the `init` process, with process ID 1.
- A `SIGCHLD` signal is sent to the parent process.
- If the process is a session leader that has a controlling terminal, then a `SIGHUP` signal is sent to each process in the foreground job, and the controlling terminal is disassociated from that session.[13]
- If termination of a process causes a process group to become orphaned, and any member of that process group is stopped, then a `SIGHUP` signal and a `SIGCONT` signal are sent to each process in the group.[14]

[10] Ibid., "Signal Handling".
[11] Ibid., "Low-Level Input/Output".
[12] Ibid., "Process Completion".
[13] Ibid., "Job Control".
[14] Ibid., "Job Control".

15 Input/Output Overview

Most programs need to do either input (reading data) or output (writing data), or most frequently both, in order to do anything useful. The GNU C Library provides such a large selection of input and output functions that the hardest part is often deciding which function is most appropriate!

This chapter introduces concepts and terminology relating to input and output. Other chapters relating to the GNU I/O facilities are

- Chapter 17 [Input/Output on Streams], page 439, covers the high-level functions that operate on streams, including formatted input and output.
- Loosemore et al., "Low-Level I/O" (see chap. 1, n. 1), covers the basic I/O and control functions on file descriptors.
- Loosemore et al., "File-System Interface", covers functions for operating on directories and for manipulating file attributes such as access modes and ownership.
- Loosemore et al., "Pipes and FIFOs", includes information on the basic interprocess communication facilities.
- Loosemore et al., "Sockets", covers a more complicated interprocess communication facility with support for networking.
- Loosemore et al., "Low-Level Terminal Interface", covers functions for changing how input and output to terminals or other serial devices are processed.

15.1 Input/Output Concepts

Before you can read or write the contents of a file, you must establish a connection or communications channel to the file. This process is called *opening* the file. You can open a file for reading, writing, or both.

The connection to an open file is represented either as a stream or as a file descriptor. You pass this as an argument to the functions that do the actual read or write operations, to tell them which file to operate on. Certain functions expect streams, and others are designed to operate on file descriptors.

When you have finished reading to or writing from the file, you can terminate the connection by *closing* the file. Once you have closed a stream or file descriptor, you cannot do any more input or output operations on it.

15.1.1 Streams and File Descriptors

When you want to do input or output to a file, you have a choice of two basic mechanisms for representing the connection between your program and the file: file descriptors and streams. File descriptors are represented as objects of type `int`, while streams are represented as `FILE *` objects.

File descriptors provide a primitive, low-level interface to input and output operations. Both file descriptors and streams can represent a connection to a device

(such as a terminal), or a pipe or socket for communicating with another process, as well as a normal file. But, if you want to do control operations that are specific to a particular kind of device, you must use a file descriptor; there are no facilities to use streams in this way. You must also use file descriptors if your program needs to do input or output in special modes, such as nonblocking (or polled) input.[1]

Streams provide a higher-level interface, layered on top of the primitive file descriptor facilities. The stream interface treats all kinds of files pretty much alike—the sole exception being the three styles of buffering that you can choose (see Section 17.20 [Stream Buffering], page 504).

The main advantage of using the stream interface is that the set of functions for performing actual input and output operations (as opposed to control operations) on streams is much richer and more powerful than the corresponding facilities for file descriptors. The file descriptor interface provides only simple functions for transferring blocks of characters, but the stream interface also provides powerful formatted input and output functions (`printf` and `scanf`) as well as functions for character- and line-oriented input and output.

Since streams are implemented in terms of file descriptors, you can extract the file descriptor from a stream and perform low-level operations directly on the file descriptor. You can also initially open a connection as a file descriptor and then make a stream associated with that file descriptor.

In general, you should stick with using streams rather than file descriptors, unless there is some specific operation you want to do that can only be done on a file descriptor. If you are a beginning programmer and aren't sure what functions to use, we suggest that you concentrate on the formatted input functions (see Section 17.14 [Formatted Input], page 486) and formatted output functions (see Section 17.12 [Formatted Output], page 460).

If you are concerned about portability of your programs to systems other than GNU, you should also be aware that file descriptors are not as portable as streams. You can expect any system running ISO C to support streams, but non-GNU systems may not support file descriptors at all, or may only implement a subset of the GNU functions that operate on file descriptors. Most of the file descriptor functions in the GNU library are included in the POSIX.1 standard, however.

15.1.2 File Position

One of the attributes of an open file is its *file position* that keeps track of where in the file the next character is to be read or written. In the GNU system, and all POSIX.1 systems, the file position is simply an integer representing the number of bytes from the beginning of the file.

The file position is normally set to the beginning of the file when it is opened, and each time a character is read or written, the file position is incremented. In other words, access to the file is normally *sequential*.

[1] Loosemore et al., "File Status Flags" (see chap. 1, n. 1).

Ordinary files permit read or write operations at any position within the file. Some other kinds of files may also permit this. Files which do permit this are sometimes referred to as *random-access* files. You can change the file position using the `fseek` function on a stream (see Section 17.18 [File Positioning], page 500) or the `lseek` function on a file descriptor.[2] If you try to change the file position on a file that doesn't support random access, you get the `ESPIPE` error.

Streams and descriptors that are opened for *append access* are treated specially for output; output to such files is *always* appended sequentially to the *end* of the file, regardless of the file position. However, the file position is still used to control where in the file reading is done.

If you think about it, you'll realize that several programs can read a given file at the same time. In order for each program to be able to read the file at its own pace, each program must have its own file pointer, which is not affected by anything the other programs do.

In fact, each opening of a file creates a separate file position. Thus, if you open a file twice even in the same program, you get two streams or descriptors with independent file positions.

By contrast, if you open a descriptor and then duplicate it to get another descriptor, these two descriptors share the same file position: changing the file position of one descriptor will affect the other.

15.2 File Names

In order to open a connection to a file, or to perform other operations such as deleting a file, you need some way to refer to the file. Nearly all files have names that are strings—even files that are actually devices such as tape drives or terminals. These strings are called *file names*. You specify the file name to say which file you want to open or operate on.

This section describes the conventions for file names and how the operating system works with them.

15.2.1 Directories

In order to understand the syntax of file names, you need to understand how the file system is organized into a hierarchy of directories.

A *directory* is a file that contains information to associate other files with names; these associations are called *links* or *directory entries*. Sometimes, people speak of "files in a directory", but in reality, a directory only contains pointers to files, not the files themselves.

The name of a file contained in a directory entry is called a *file-name component*. In general, a file name consists of a sequence of one or more such components, separated by the slash character ('`/`'). A file name that is just one component

[2] Ibid., "Input and Output Primitives".

names a file with respect to its directory. A file name with multiple components names a directory, and then a file in that directory, and so on.

Some other documents, such as the POSIX standard, use the term *pathname* for what we call a file name, and either *filename* or *pathname component* for what this manual calls a file-name component. We don't use this terminology because a *path* is something completely different (a list of directories to search), and we think that *pathname* used for something else will confuse users. We always use *file name* and *file-name component* (or sometimes just *component*, where the context is obvious) in GNU documentation. Some macros use the POSIX terminology in their names, such as `PATH_MAX`. These macros are defined by the POSIX standard, so we cannot change their names.

You can find more detailed information about operations on directories in Loosemore et al., "File-System Interface" (see chap. 1, n. 1).

15.2.2 File-Name Resolution

A file name consists of file-name components separated by slash ('`/`') characters. On the systems that the GNU C Library supports, multiple successive '`/`' characters are equivalent to a single '`/`' character.

The process of determining what file a file name refers to is called *file-name resolution*. This is performed by examining the components that make up a file name in left-to-right order, and locating each successive component in the directory named by the previous component. Of course, each of the files that are referenced as directories must actually exist, be directories instead of regular files, and have the appropriate permissions to be accessible by the process; otherwise the file-name resolution fails.

If a file name begins with a '`/`', the first component in the file name is located in the *root directory* of the process (usually all processes on the system have the same root directory). Such a file name is called an *absolute file name*.

Otherwise, the first component in the file name is located in the current working directory.[3] This kind of file name is called a *relative file name*.

The file-name components '`.`' (*dot*) and '`..`' (*dot-dot*) have special meanings. Every directory has entries for these file-name components. The file-name component '`.`' refers to the directory itself, while the file-name component '`..`' refers to its *parent directory* (the directory that contains the link for the directory in question). As a special case, '`..`' in the root directory refers to the root directory itself, since it has no parent; thus '`/..`' is the same as '`/`'.

Here are some examples of file names:

'`/a`'	The file named '`a`', in the root directory
'`/a/b`'	The file named '`b`', in the directory named '`a`' in the root directory
'`a`'	The file named '`a`', in the current working directory

[3] Ibid., "Working Directory".

‘/a/./b’ The same as ‘/a/b’

‘./a’ The file named ‘a’, in the current working directory

‘../a’ The file named ‘a’, in the parent directory of the current working directory

A file name that names a directory may optionally end in a ‘/’. You can specify a file name of ‘/’ to refer to the root directory, but the empty string is not a meaningful file name. If you want to refer to the current working directory, use a file name of ‘.’ or ‘./’.

Unlike some other operating systems, the GNU system doesn’t have any built-in support for file types (or extensions) or file versions as part of its file-name syntax. Many programs and utilities use conventions for file names—for example, files containing C source code usually have names suffixed with ‘.c’—but there is nothing in the file system itself that enforces this kind of convention.

15.2.3 File-Name Errors

Functions that accept file-name arguments usually detect these errno error conditions relating to the file-name syntax or trouble finding the named file. These errors are referred to throughout this manual as the *usual file-name errors*.

EACCES The process does not have search permission for a directory component of the file name.

ENAMETOOLONG
This error is used when either the total length of a file name is greater than PATH_MAX, or when an individual file-name component has a length greater than NAME_MAX.[4]

In the GNU system, there is no imposed limit on overall file-name length, but some file systems may place limits on the length of a component.

ENOENT This error is reported when a file referenced as a directory component in the file name doesn’t exist, or when a component is a symbolic link whose target file does not exist.[5]

ENOTDIR A file that is referenced as a directory component in the file name exists, but it isn’t a directory.

ELOOP Too many symbolic links were resolved while trying to look up the file name. The system has an arbitrary limit on the number of symbolic links that may be resolved in looking up a single file name, as a primitive way to detect loops.[6]

---

[4] Ibid., “Limits on File-System Capacity”.
[5] Ibid., “Symbolic Links”.
[6] Ibid., “Symbolic Links”.

15.2.4 Portability of File Names

The rules for the syntax of file names discussed in Section 15.2 [File Names], page 431, are the rules normally used by the GNU system and by other POSIX systems. However, other operating systems may use other conventions.

There are two reasons why it can be important for you to be aware of file-name portability issues:

- If your program makes assumptions about file-name syntax, or contains embedded literal file-name strings, it is more difficult to get it to run under other operating systems that use different syntax conventions.
- Even if you are not concerned about running your program on machines that run other operating systems, it may still be possible to access files that use different naming conventions. For example, you may be able to access file systems on another computer running a different operating system over a network, or read and write disks in formats used by other operating systems.

The ISO C standard says very little about file-name syntax, only that file names are strings. In addition to varying restrictions on the length of file names and what characters can validly appear in a file name, different operating systems use different conventions and syntax for concepts such as structured directories and file types or extensions. Some concepts, such as file versions, might be supported in some operating systems and not in others.

The POSIX.1 standard allows implementations to put additional restrictions on file-name syntax, concerning what characters are permitted in file names and on the length of file name and file-name component strings. However, in the GNU system, you do not need to worry about these restrictions; any character except the null character is permitted in a file-name string, and there are no limits on the length of file-name strings.

16 Debugging Support

Applications are usually debugged using dedicated debugger programs. But sometimes this is not possible and, in any case, it is useful to provide the developer with as much information as possible at the time the problems are experienced. For this reason, a few functions are provided that a program can use to help the developer more easily locate the problem.

16.1 Backtraces

A *backtrace* is a list of the function calls that are currently active in a thread. The usual way to inspect a backtrace of a program is to use an external debugger such as gdb. However, sometimes it is useful to obtain a backtrace programmatically from within a program, e.g., for the purposes of logging or diagnostics.

The header file '`execinfo.h`' declares three functions that obtain and manipulate backtraces of the current thread.

`int` **backtrace** (`void **`*buffer*`, int` *size*) Function

The `backtrace` function obtains a backtrace for the current thread, as a list of pointers, and places the information into *buffer*. The argument *size* should be the number of `void *` elements that will fit into *buffer*. The return value is the actual number of entries of *buffer* that are obtained, and is at most *size*.

The pointers placed in *buffer* are actually return addresses obtained by inspecting the stack, one return address per stack frame.

Certain compiler optimizations may interfere with obtaining a valid backtrace. Function in-lining causes the in-lined function to not have a stack frame; tail-call optimization replaces one stack frame with another; frame pointer elimination will stop `backtrace` from interpreting the stack contents correctly.

`char **` **backtrace_symbols** (`void *const *`*buffer*`, int` *size*) Function

The `backtrace_symbols` function translates the information obtained from the `backtrace` function into an array of strings. The argument *buffer* should be a pointer to an array of addresses obtained via the `backtrace` function, and *size* is the number of entries in that array (the return value of `backtrace`).

The return value is a pointer to an array of strings, which has *size* entries just like the array *buffer*. Each string contains a printable representation of the corresponding element of *buffer*. It includes the function name (if this can be determined), an offset into the function, and the actual return address (in hexadecimal).

Currently, the function name and offset can only be obtained on systems that use the ELF binary format for programs and libraries. On other systems, only the hexadecimal return address will be present. Also, you may need to pass additional flags to the linker to make the function names available to the program; for example, on systems using GNU ld, you must pass `-rdynamic`.

The return value of `backtrace_symbols` is a pointer obtained via the `malloc` function, and it is the responsibility of the caller to `free` that pointer. Only the return value need be freed, not the individual strings.

The return value is `NULL` if sufficient memory for the strings cannot be obtained.

`void` **backtrace_symbols_fd** (`void *const *`*buffer*`, int` *size*`, int` *fd*) Function

The `backtrace_symbols_fd` function performs the same translation as the function `backtrace_symbols` function. Instead of returning the strings to the caller, it writes the strings to the file descriptor *fd*, one per line. It does not use the `malloc` function, and can therefore be used in situations where that function might fail.

The following program illustrates the use of these functions. Note that the array to contain the return addresses returned by `backtrace` is allocated on the stack. Therefore, code like this can be used in situations where the memory handling via `malloc` does not work anymore (in which case the `backtrace_symbols` has to be replaced by a `backtrace_symbols_fd` call as well). The number of return addresses is normally not very large. Even complicated programs rather seldom have a nesting level of more than fifty, and with two hundred possible entries, probably all programs should be covered.

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>

/* Obtain a backtrace and print it to stdout. */
void
print_trace (void)
{
  void *array[10];
  size_t size;
  char **strings;
  size_t i;

  size = backtrace (array, 10);
  strings = backtrace_symbols (array, size);

  printf ("Obtained %zd stack frames.\n", size);

  for (i = 0; i < size; i++)
     printf ("%s\n", strings[i]);

  free (strings);
}
```

```
/* A dummy function to make the backtrace more interesting */
void
dummy_function (void)
{
  print_trace ();
}

int
main (void)
{
  dummy_function ();
  return 0;
}
```

17 Input/Output on Streams

This chapter describes the functions for creating streams and performing input and output operations on them. As discussed in Chapter 15 [Input/Output Overview], page 429, a stream is a fairly abstract, high-level concept representing a communications channel to a file, device, or process.

17.1 Streams

For historical reasons, the type of the C data structure that represents a stream is called `FILE` rather than *stream*. Since most of the library functions deal with objects of type `FILE *`, sometimes the term *file pointer* is also used to mean *stream*. This leads to unfortunate confusion over terminology in many books on C. This manual, however, is careful to use the terms *file* and *stream* only in the technical sense.

The `FILE` type is declared in the header file '`stdio.h`'.

FILE Data Type

This is the data type used to represent stream objects. A `FILE` object holds all of the internal state information about the connection to the associated file, including such things as the file position indicator and buffering information. Each stream also has error and end-of-file status indicators that can be tested with the `ferror` and `feof` functions (see Section 17.15 [End-of-File and Errors], page 497).

`FILE` objects are allocated and managed internally by the input/output library functions. Don't try to create your own objects of type `FILE`; let the library do it. Your programs should deal only with pointers to these objects (that is, `FILE *` values) rather than the objects themselves.

17.2 Standard Streams

When the `main` function of your program is invoked, it already has three predefined streams open and available for use. These represent the "standard" input and output channels that have been established for the process.

These streams are declared in the header file '`stdio.h`'.

`FILE *` **stdin** Variable

The *standard input* stream, which is the normal source of input for the program

`FILE *` **stdout** Variable

The *standard output* stream, which is used for normal output from the program

`FILE *` **stderr** Variable

The *standard error* stream, which is used for error messages and diagnostics issued by the program

In the GNU system, you can specify what files or processes correspond to these streams using the pipe and redirection facilities provided by the shell.[1] Most other operating systems provide similar mechanisms, but the details of how to use them can vary.

In the GNU C Library, `stdin`, `stdout`, and `stderr` are normal variables that you can set just like any others. For example, to redirect the standard output to a file, you could do:

```
fclose (stdout);
stdout = fopen ("standard-output-file", "w");
```

However, in other systems, `stdin`, `stdout` and `stderr` are macros that you cannot assign to in the normal way. But you can use `freopen` to get the effect of closing one and reopening it (see Section 17.3 [Opening Streams], page 440).

The three streams `stdin`, `stdout` and `stderr` are not unoriented at program start (see Section 17.6 [Streams in Internationalized Applications], page 448).

17.3 Opening Streams

Opening a file with the `fopen` function creates a new stream and establishes a connection between the stream and a file. This may involve creating a new file.

Everything described in this section is declared in the header file ‘`stdio.h`’.

`FILE *` **fopen** (`const char *`*filename*`, const char *`*opentype*) Function

The `fopen` function opens a stream for I/O to the file *filename*, and returns a pointer to the stream.

The *opentype* argument is a string that controls how the file is opened and specifies attributes of the resulting stream. It must begin with one of the following sequences of characters:

‘`r`’ Open an existing file for reading only.

‘`w`’ Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.

‘`a`’ Open a file for append access; that is, writing at the end of file only. If the file already exists, its initial contents are unchanged and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.

[1] The primitives shells use to implement these facilities are described in Loosemore et al., “File-System Interface” (see chap. 1, n. 1).

‘r+’ Open an existing file for both reading and writing. The initial contents of the file are unchanged and the initial file position is at the beginning of the file.

‘w+’ Open a file for both reading and writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.

‘a+’ Open or create file for both reading and appending. If the file exists, its initial contents are unchanged. Otherwise, a new file is created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

As you can see, ‘+’ requests a stream that can do both input and output. The ISO standard says that when using such a stream, you must call `fflush` (see Section 17.20 [Stream Buffering], page 504) or a file positioning function such as `fseek` (see Section 17.18 [File Positioning], page 500) when switching from reading to writing or vice versa. Otherwise, internal buffers might not be emptied properly. The GNU C Library does not have this limitation; you can do arbitrary reading and writing operations on a stream in any order.

Additional characters may appear after these to specify flags for the call. Always put the mode (‘r’, ‘w+’, etc.) first; that is the only part guaranteed to be understood by all systems.

The GNU C Library defines one additional character for use in *opentype*: the character ‘x’ insists on creating a new file—if a file *filename* already exists, `fopen` fails rather than opening it. If you use ‘x’, it is guaranteed that you will not clobber an existing file. This is equivalent to the `O_EXCL` option to the `open` function.[2]

The character ‘b’ in *opentype* has a standard meaning; it requests a binary stream rather than a text stream. But this makes no difference in POSIX systems (including the GNU system). If both ‘+’ and ‘b’ are specified, they can appear in either order (see Section 17.17 [Text and Binary Streams], page 499).

If the *opentype* string contains the sequence `ccs=`*STRING*, then *STRING* is taken as the name of a coded character set and `fopen` will mark the stream as wide-oriented with appropriate conversion functions in place to convert from and to the character set *STRING*. Any other stream is opened initially unoriented, and the orientation is decided with the first file operation. If the first operation is a wide-character operation, the stream is not only marked as wide-oriented—the conversion functions to convert to the coded character set used for the current locale are also loaded. This will not change anymore from this point on, even if the locale selected for the `LC_CTYPE` category is changed.

Any other characters in *opentype* are simply ignored. They may be meaningful in other systems.

If the open fails, `fopen` returns a null pointer.

[2] Ibid., “Opening and Closing Files”.

When the sources are compiling with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is in fact `fopen64`, since the LFS interface transparently replaces the old interface.

You can have multiple streams (or file descriptors) pointing to the same file open at the same time. If you do only input, this will be straightforward, but you must be careful if any output streams are included.[3] This is equally true whether the streams are in one program (not usual) or in several programs (which can easily happen). It may be advantageous to use the file-locking facilities to avoid simultaneous access.[4]

`FILE *` **fopen64** (`const char *`*filename*`, const char *`*opentype*) Function

This function is similar to `fopen`, but the stream it returns a pointer for is opened using `open64`. Therefore, this stream can be used even on files larger then 2^31 bytes on 32-bit machines.

The return type is still `FILE *`. There is no special `FILE` type for the LFS interface.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `fopen` and so transparently replaces the old interface.

`int` **FOPEN_MAX** Macro

The value of this macro is an integer constant expression that represents the minimum number of streams that the implementation guarantees can be open simultaneously. You might be able to open more than this many streams, but that is not guaranteed. The value of this constant is at least eight, which includes the three standard streams `stdin`, `stdout` and `stderr`. In POSIX.1 systems, this value is determined by the `OPEN_MAX` parameter.[5] In BSD and GNU, it is controlled by the `RLIMIT_NOFILE` resource limit.[6]

`FILE *` **freopen** (`const char *`*filename*`, const char *`*opentype*`, FILE *`*stream*) Function

This function is like a combination of `fclose` and `fopen`. It first closes the stream referred to by *stream*, ignoring any errors that are detected in the process. Because errors are ignored, you should not use `freopen` on an output stream if you have actually done any output using the stream. Then the file named by *filename* is opened with mode *opentype* as for `fopen`, and associated with the same stream object *stream*.

If the operation fails, a null pointer is returned; otherwise, `freopen` returns *stream*.

---

[3] Ibid., “Dangers of Mixing Streams and Descriptors”.

[4] Ibid., “File Locks”.

[5] Ibid., “General Capacity-Limits”.

[6] Ibid., “Limiting Resource Usage”.

`freopen` has traditionally been used to connect a standard stream such as `stdin` with a file of your own choice. This is useful in programs in which use of a standard stream for certain purposes is hard-coded. In the GNU C Library, you can simply close the standard streams and open new ones with `fopen`. But other systems lack this ability, so using `freopen` is more portable.

When the sources are compiling with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is in fact `freopen64`, since the LFS interface transparently replaces the old interface.

`FILE *` **freopen64** (`const char *`*filename*`, const char *`*opentype*`, FILE *`*stream*) Function

This function is similar to `freopen`. The only difference is that on 32-bit machines, the stream returned is able to read beyond the 2^31-bytes limits imposed by the normal interface. The stream pointed to by *stream* need not be opened using `fopen64` or `freopen64`, since its mode is not important for this function.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `freopen` and so transparently replaces the old interface.

In some situations, it is useful to know whether a given stream is available for reading or writing. This information is normally not available and would have to be remembered separately. Solaris introduced a few functions to get this information from the stream descriptor and these functions are also available in the GNU C Library.

`int` **__freadable** (`FILE *`*stream*) Function

The `__freadable` function determines whether the stream *stream* was opened to allow reading. In this case, the return value is nonzero. For write-only streams, the function returns zero.

This function is declared in ‘`stdio_ext.h`’.

`int` **__fwritable** (`FILE *`*stream*) Function

The `__fwritable` function determines whether the stream *stream* was opened to allow writing. In this case, the return value is nonzero. For read-only streams, the function returns zero.

This function is declared in ‘`stdio_ext.h`’.

For slightly different kinds of problems, there are two more functions. They provide even finer-grained information.

`int` **__freading** (`FILE *`*stream*) Function

The `__freading` function determines whether the stream *stream* was last read from or whether it is opened read-only. In this case, the return value is nonzero. Otherwise, it is zero. Determining whether a stream opened for reading and

writing was last used for writing allows you to draw conclusions about the content of the buffer, among other things.

This function is declared in ‘stdio_ext.h’.

`int` **__fwriting** (`FILE` *stream*) Function

The `__fwriting` function determines whether the stream *stream* was last written to or whether it is opened write-only. In this case, the return value is nonzero. Otherwise, it is zero.

This function is declared in ‘stdio_ext.h’.

17.4 Closing Streams

When a stream is closed with `fclose`, the connection between the stream and the file is canceled. After you have closed a stream, you cannot perform any additional operations on it.

`int` **fclose** (`FILE` *stream*) Function

This function causes *stream* to be closed and the connection to the corresponding file to be broken. Any buffered output is written, and any buffered input is discarded. The `fclose` function returns a value of `0` if the file was closed successfully, and `EOF` if an error was detected.

It is important to check for errors when you call `fclose` to close an output stream, because real, everyday errors can be detected at this time. For example, when `fclose` writes the remaining buffered output, it might get an error because the disk is full. Even if you know the buffer is empty, errors can still occur when closing a file if you are using NFS.

The function `fclose` is declared in ‘stdio.h’.

To close all streams currently available, the GNU C Library provides another function.

`int` **fcloseall** (`void`) Function

This function causes all open streams of the process to be closed and the connection to corresponding files to be broken. All buffered data is written, and any buffered input is discarded. The `fcloseall` function returns a value of `0` if all the files were closed successfully, and `EOF` if an error was detected.

This function should be used only in special situations, such as when an error occurred and the program must be aborted. Normally, each single stream should be closed separately so that problems with individual streams can be identified. It is also problematic since the standard streams (see Section 17.2 [Standard Streams], page 439) will also be closed.

The function `fcloseall` is declared in ‘stdio.h’.

If the `main` function to your program returns, or if you call the `exit` function (see Section 14.6.1 [Normal Termination], page 425), all open streams are automatically closed properly. If your program terminates in any other manner, such as by

calling the `abort` function (see Section 14.6.4 [Aborting a Program], page 427), or from a fatal signal[7], open streams might not be closed properly. Buffered output might not be flushed, and files may be incomplete. For more information on buffering of streams, see Section 17.20 [Stream Buffering], page 504.

17.5 Streams and Threads

Streams can be used in multithreaded applications in the same way they are used in single-threaded applications. But the programmer must be aware of the possible complications. It is also important to know about these if the program one writes never use threads, since the design and implementation of many stream functions is heavily influenced by the requirements added by multithreaded programming.

The POSIX standard requires that, by default, the stream operations are atomic—issuing two stream operations for the same stream in two threads at the same time will cause the operations to be executed as if they were issued sequentially. The buffer operations performed while reading or writing are protected from other uses of the same stream. To do this, each stream has an internal lock object that has to be (implicitly) acquired before any work can be done.

But there are situations where this is not enough, and there are also situations where this is not wanted. The implicit locking is not enough if the program requires more than one stream function call to happen atomically. One example would be if an output line a program wants to generate is created by several function calls. The functions by themselves would ensure only atomicity of their own operation, but not atomicity over all the function calls. For this, it is necessary to perform the stream locking in the application code.

`void` **flockfile** (`FILE *`*stream*) Function

The `flockfile` function acquires the internal locking object associated with the stream *stream*. This ensures that no other thread can explicitly through `flockfile`/`ftrylockfile` or implicitly through a call of a stream function lock the stream. The thread will block until the lock is acquired. An explicit call to `funlockfile` has to be used to release the lock.

`int` **ftrylockfile** (`FILE *`*stream*) Function

The `ftrylockfile` function tries to acquire the internal locking object associated with the stream *stream* just like `flockfile`. But unlike `flockfile`, this function does not block if the lock is not available. `ftrylockfile` returns zero if the lock was successfully acquired. Otherwise, the stream is locked by another thread.

`void` **funlockfile** (`FILE *`*stream*) Function

The `funlockfile` function releases the internal locking object of the stream *stream*. The stream must have been locked before by a call to `flockfile` or

[7] Ibid., “Signal Handling”.

a successful call of `ftrylockfile`. The implicit locking performed by the stream operations do not count. The `funlockfile` function does not return an error status, and the behavior of a call for a stream that is not locked by the current thread is undefined.

The following example shows how the functions above can be used to generate an output line atomically even in multithreaded applications (yes, the same job could be done with one `fprintf` call, but that is sometimes not possible):

```
FILE *fp;
{
   ...
   flockfile (fp);
   fputs ("This is test number ", fp);
   fprintf (fp, "%d\n", test);
   funlockfile (fp)
}
```

Without the explicit locking, it would be possible for another thread to use the stream *fp* after the `fputs` call return and before `fprintf` was called, with the result that the number does not follow the word '`number`'.

From this description, it might already be clear that the locking objects in streams are no simple mutexes. Since locking the same stream twice in the same thread is allowed, the locking objects must be equivalent to recursive mutexes. These mutexes keep track of the owner and the number of times the lock is acquired. The same number of `funlockfile` calls by the same threads is necessary to unlock the stream completely. For instance:

```
void
foo (FILE *fp)
{
  ftrylockfile (fp);
  fputs ("in foo\n", fp);
  /* This is very wrong!!!  */
  funlockfile (fp);
}
```

It is important here that the `funlockfile` function is only called if the `ftrylockfile` function succeeded in locking the stream. It is therefore always wrong to ignore the result of `ftrylockfile`. And it makes no sense, since otherwise one would use `flockfile`. The result of code like that above is that either `funlockfile` tries to free a stream that hasn't been locked by the current thread, or it frees the stream prematurely. The code should look like this:

```
void
foo (FILE *fp)
{
  if (ftrylockfile (fp) == 0)
    {
```

```
      fputs ("in foo\n", fp);
      funlockfile (fp);
    }
}
```

Now that we covered why it is necessary to have locking, it is necessary to talk about situations when locking is unwanted and what can be done. The locking operations (explicit or implicit) don't come for free. Even if a lock is not taken, the cost is not zero. The operations that have to be performed require memory operations that are safe in multiprocessor environments. With the many local caches involved in such systems, this is quite costly. So it is best to avoid the locking completely if it is not needed—because the code in question is never used in a context where two or more threads may use a stream at a time. This can be determined most of the time for application code; for library code that can be used in many contexts, one should default to be conservative and use locking.

There are two basic mechanisms to avoid locking. The first is to use the `_unlocked` variants of the stream operations. The POSIX standard defines quite a few of those, and the GNU library adds a few more. These variants of the functions behave just like the functions with the name without the suffix, except that they do not lock the stream. Using these functions is very desirable, since they are potentially much faster. This is not only because the locking operation itself is avoided. More importantly, functions like `putc` and `getc` are very simple and traditionally (before the introduction of threads) were implemented as macros that are very fast if the buffer is not empty. With the addition of locking requirements, these functions are no longer implemented as macros, since they would would expand to too much code. But these macros are still available with the same functionality under the new names `putc_unlocked` and `getc_unlocked`. This potentially huge difference in speed also suggests the use of the `_unlocked` functions even if locking is required. The difference is that the locking then has to be performed in the program:

```
void
foo (FILE *fp, char *buf)
{
  flockfile (fp);
  while (*buf != '/')
    putc_unlocked (*buf++, fp);
  funlockfile (fp);
}
```

If in this example, the `putc` function would be used and the explicit locking would be missing, the `putc` function would have to acquire the lock in every call, potentially many times depending on when the loop terminates. Writing it the way illustrated above allows the `putc_unlocked` macro to be used, which means no locking and direct manipulation of the buffer of the stream.

A second way to avoid locking is by using a nonstandard function that was introduced in Solaris and is available in the GNU C Library as well.

`int` **__fsetlocking** (`FILE *`*stream*`, int` *type*) Function

The `__fsetlocking` function can be used to select whether the stream operations will implicitly acquire the locking object of the stream *stream*. By default, this is done, but it can be disabled and reinstated using this function. There are three values defined for the *type* parameter:

`FSETLOCKING_INTERNAL`
: The stream `stream` will from now on use the default internal locking. Every stream operation with the exception of the `_unlocked` variants will implicitly lock the stream.

`FSETLOCKING_BYCALLER`
: After the `__fsetlocking` function returns, the user is responsible for locking the stream. None of the stream operations will implicitly do this anymore until the state is set back to `FSETLOCKING_INTERNAL`.

`FSETLOCKING_QUERY`
: `__fsetlocking` only queries the current locking state of the stream. The return value will be `FSETLOCKING_INTERNAL` or `FSETLOCKING_BYCALLER` depending on the state.

The return value of `__fsetlocking` is either `FSETLOCKING_INTERNAL` or `FSETLOCKING_BYCALLER` depending on the state of the stream before the call.

This function and the values for the *type* parameter are declared in ‘`stdio_ext.h`’.

This function is especially useful when program code has to be used that is written without knowledge of the `_unlocked` functions (or if the programmer was too lazy to use them).

17.6 Streams in Internationalized Applications

ISO C90 introduced the new type `wchar_t` to allow handling larger character sets. What was missing was the ability to output strings of `wchar_t` directly. One had to convert them into multibyte strings using `mbstowcs` (there was no `mbsrtowcs` yet) and then use the normal stream functions. While this is doable, it is very cumbersome, since performing the conversions is not trivial and greatly increases program complexity and size.

The Unix standard (XPG4.2 and later) includes two additional format specifiers for the `printf` and `scanf` families of functions. Printing and reading of single wide characters was made possible using the `%C` specifier, and wide-character strings can be handled with `%S`. These modifiers behave just like `%c` and `%s`, only they expect the corresponding argument to have the wide-character type and that the wide character and string are transformed into/from multibyte strings before being used.

This was a beginning, but it is still not good enough. It is not always desirable to use `printf` and `scanf`. The other, smaller and faster functions cannot handle wide characters. Second, it is not possible to have a format string for `printf` and `scanf` consisting of wide characters. The result is that format strings would have to be generated if they have to contain nonbasic characters.

In the Amendment 1 to ISO C90, a whole new set of functions was added to solve the problem. Most of the stream functions got counterparts that take a wide character or wide-character string instead of a character or string respectively. The new functions operate on the same streams (like `stdout`). This is different from the model of the C++ run-time library where separate streams for wide and normal I/O are used.

Being able to use the same stream for wide and normal operations comes with a restriction: a stream can be used either for wide operations or for normal operations. Once it is decided, there is no way back. Only a call to `freopen` or `freopen64` can reset the *orientation*. The orientation can be decided in three ways:

- If any of the normal character functions is used (this includes the `fread` and `fwrite` functions), the stream is marked as not wide oriented.
- If any of the wide-character functions is used, the stream is marked as wide oriented.
- The `fwide` function can be used to set the orientation either way.

It is important to never mix the use of wide and not-wide operations on a stream. There are no diagnostics issued. The application behavior will simply be strange or the application will simply crash. The `fwide` function can help avoid this.

`int` **fwide** (`FILE *`*stream*`, int` *mode*) Function

The `fwide` function can be used to set and query the state of the orientation of the stream *stream*. If the *mode* parameter has a positive value, the streams get wide oriented, and for negative values they get narrow oriented. It is not possible to overwrite previous orientations with `fwide`—if the stream *stream* was already oriented before the call, nothing is done.

If *mode* is zero, the current orientation state is queried, and nothing is changed.

The `fwide` function returns a negative value, zero, or a positive value if the stream is narrow, not at all, or wide oriented, respectively.

This function was introduced in Amendment 1 to ISO C90 and is declared in ‘`wchar.h`’.

It is generally a good idea to orient a stream as early as possible. This can prevent surprise especially for the standard streams `stdin`, `stdout`, and `stderr`. If some library function in some situations uses one of these streams and this use orients the stream in a different way than the rest of the application expects, one might end up with hard-to-reproduce errors. Remember that no errors are signalled if the streams are used incorrectly. Leaving a stream unoriented after creation is normally only necessary for library functions that create streams that can be used in different contexts.

When writing code that uses streams and that can be used in different contexts, it is important to query the orientation of the stream before using it (unless the rules of the library interface demand a specific orientation). The following little, silly function illustrates this:

```
void
print_f (FILE *fp)
{
  if (fwide (fp, 0) > 0)
    /* Positive return value means wide orientation.  */
    fputwc (L'f', fp);
  else
    fputc ('f', fp);
}
```

In this case, the function `print_f` decides about the orientation of the stream if it was unoriented before (this will not happen if the advice above is followed).

The encoding used for the `wchar_t` values is unspecified, and the user must not make any assumptions about it. For I/O of `wchar_t` values, this means that it is impossible to write these values directly to the stream. This is not what follows from the ISO C locale model either. What happens instead is that the bytes read from or written to the underlying media are first converted into the internal encoding chosen by the implementation for `wchar_t`. The external encoding is determined by the `LC_CTYPE` category of the current locale, or by the '`ccs`' part of the mode specification given to `fopen`, `fopen64`, `freopen` or `freopen64`. How and when the conversion happens is unspecified, and it happens invisibly to the user.

Since a stream is created in the unoriented state, it has at that point no conversion associated with it. The conversion that will be used is determined by the `LC_CTYPE` category selected at the time the stream is oriented. If the locales are changed at the run time, this might produce surprising results unless you pay attention. This is just another good reason to orient the stream explicitly as soon as possible, perhaps with a call to `fwide`.

17.7 Simple Output by Characters or Lines

This section describes functions for performing character- and line-oriented output.

These narrow stream functions are declared in the header file '`stdio.h`' and the wide stream functions in '`wchar.h`'.

`int` **fputc** (`int` *c*, `FILE *`*stream*) Function

The `fputc` function converts the character *c* to type `unsigned char`, and writes it to the stream *stream*. `EOF` is returned if a write error occurs. Otherwise, the character *c* is returned.

`wint_t` **fputwc** (`wchar_t` *wc*, `FILE *`*stream*) Function
The `fputwc` function writes the wide character *wc* to the stream *stream*. `WEOF` is returned if a write error occurs. Otherwise, the character *wc* is returned.

`int` **fputc_unlocked** (`int` *c*, `FILE *`*stream*) Function
The `fputc_unlocked` function is equivalent to the `fputc` function, except that it does not implicitly lock the stream.

`wint_t` **fputwc_unlocked** (`wint_t` *wc*, `FILE *`*stream*) Function
The `fputwc_unlocked` function is equivalent to the `fputwc` function, except that it does not implicitly lock the stream.
This function is a GNU extension.

`int` **putc** (`int` *c*, `FILE *`*stream*) Function
This is just like `fputc`, except that most systems implement it as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. `putc` is usually the best function to use for writing a single character.

`wint_t` **putwc** (`wchar_t` *wc*, `FILE *`*stream*) Function
This is just like `fputwc`, except that it can be implemented as a macro, making it faster. One consequence is that it may evaluate the *stream* argument more than once, which is an exception to the general rule for macros. `putwc` is usually the best function to use for writing a single wide character.

`int` **putc_unlocked** (`int` *c*, `FILE *`*stream*) Function
The `putc_unlocked` function is equivalent to the `putc` function, except that it does not implicitly lock the stream.

`wint_t` **putwc_unlocked** (`wchar_t` *wc*, `FILE *`*stream*) Function
The `putwc_unlocked` function is equivalent to the `putwc` function, except that it does not implicitly lock the stream.
This function is a GNU extension.

`int` **putchar** (`int` *c*) Function
The `putchar` function is equivalent to `putc` with `stdout` as the value of the *stream* argument.

`wint_t` **putwchar** (`wchar_t` *wc*) Function
The `putwchar` function is equivalent to `putwc` with `stdout` as the value of the *stream* argument.

`int` **putchar_unlocked** (`int` *c*) Function
The `putchar_unlocked` function is equivalent to the `putchar` function, except that it does not implicitly lock the stream.

`wint_t` **putwchar_unlocked** (`wchar_t` *wc*) Function

The `putwchar_unlocked` function is equivalent to the `putwchar` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`int` **fputs** (`const char *`*s*`, FILE *`*stream*) Function

The function `fputs` writes the string *s* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string.

This function returns `EOF` if a write error occurs, and otherwise a nonnegative value.

For example:

```
fputs ("Are ", stdout);
fputs ("you ", stdout);
fputs ("hungry?\n", stdout);
```

outputs the text, '`Are you hungry?`', followed by a newline.

`int` **fputws** (`const wchar_t *`*ws*`, FILE *`*stream*) Function

The function `fputws` writes the wide-character string *ws* to the stream *stream*. The terminating null character is not written. This function does *not* add a newline character, either. It outputs only the characters in the string.

This function returns `WEOF` if a write error occurs, and otherwise a nonnegative value.

`int` **fputs_unlocked** (`const char *`*s*`, FILE *`*stream*) Function

The `fputs_unlocked` function is equivalent to the `fputs` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`int` **fputws_unlocked** (`const wchar_t *`*ws*`, FILE *`*stream*) Function

The `fputws_unlocked` function is equivalent to the `fputws` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`int` **puts** (`const char *`*s*) Function

The `puts` function writes the string *s* to the stream `stdout` followed by a newline. The terminating null character of the string is not written. (Note that `fputs` does *not* write a newline as this function does.)

`puts` is the most convenient function for printing simple messages. For example:

```
puts ("This is a message.");
```

outputs the text, '`This is a message.`', followed by a newline.

`int` **putw** (`int` *w*, `FILE *`*stream*) Function
This function writes the word *w* (that is, an `int`) to *stream*. It is provided for compatibility with SVID, but we recommend you use `fwrite` instead (see Section 17.11 [Block Input/Output], page 459).

17.8 Character Input

This section describes functions for performing character-oriented input. These narrow stream functions are declared in the header file '`stdio.h`', and the wide-character functions are declared in '`wchar.h`'.

These functions return an `int` or `wint_t` value (for narrow and wide stream functions respectively) that is either a character of input, or the special value `EOF`/`WEOF` (usually -1). For the narrow stream functions, it is important to store the result of these functions in a variable of type `int` instead of `char`, even when you plan to use it only as a character. Storing `EOF` in a `char` variable truncates its value to the size of a character, so that it is no longer distinguishable from the valid character '`(char) -1`'. So always use an `int` for the result of `getc` and friends, and check for `EOF` after the call; once you have verified that the result is not `EOF`, you can be sure that it will fit in a '`char`' variable without loss of information.

`int` **fgetc** (`FILE *`*stream*) Function
This function reads the next character as an `unsigned char` from the stream *stream* and returns its value, converted to an `int`. If an end-of-file condition or read error occurs, `EOF` is returned instead.

`wint_t` **fgetwc** (`FILE *`*stream*) Function
This function reads the next wide character from the stream *stream* and returns its value. If an end-of-file condition or read error occurs, `WEOF` is returned instead.

`int` **fgetc_unlocked** (`FILE *`*stream*) Function
The `fgetc_unlocked` function is equivalent to the `fgetc` function, except that it does not implicitly lock the stream.

`wint_t` **fgetwc_unlocked** (`FILE *`*stream*) Function
The `fgetwc_unlocked` function is equivalent to the `fgetwc` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`int` **getc** (`FILE *`*stream*) Function
This is just like `fgetc`, except that it is permissible (and typical) for it to be implemented as a macro that evaluates the *stream* argument more than once. `getc` is often highly optimized, so it is usually the best function to use to read a single character.

`wint_t` **getwc** (`FILE *`*stream*) Function
This is just like `fgetwc`, except that it is permissible for it to be implemented as a macro that evaluates the *stream* argument more than once. `getwc` can be highly optimized, so it is usually the best function to use to read a single wide character.

`int` **getc_unlocked** (`FILE *`*stream*) Function
The `getc_unlocked` function is equivalent to the `getc` function, except that it does not implicitly lock the stream.

`wint_t` **getwc_unlocked** (`FILE *`*stream*) Function
The `getwc_unlocked` function is equivalent to the `getwc` function, except that it does not implicitly lock the stream.
This function is a GNU extension.

`int` **getchar** (`void`) Function
The `getchar` function is equivalent to `getc` with `stdin` as the value of the *stream* argument.

`wint_t` **getwchar** (`void`) Function
The `getwchar` function is equivalent to `getwc` with `stdin` as the value of the *stream* argument.

`int` **getchar_unlocked** (`void`) Function
The `getchar_unlocked` function is equivalent to the `getchar` function, except that it does not implicitly lock the stream.

`wint_t` **getwchar_unlocked** (`void`) Function
The `getwchar_unlocked` function is equivalent to the `getwchar` function, except that it does not implicitly lock the stream.
This function is a GNU extension.

Here is an example of a function that does input using `fgetc`. It would work just as well using `getc` instead, or using `getchar ()` instead of `fgetc (stdin)`. The code would also work the same for the wide-character stream functions.

```
int
y_or_n_p (const char *question)
{
  fputs (question, stdout);
  while (1)
    {
      int c, answer;
      /* Write a space to separate answer from question. */
      fputc (' ', stdout);
      /* Read the first character of the line.
```

```
      This should be the answer character, but might not be. */
    c = tolower (fgetc (stdin));
    answer = c;
    /* Discard rest of input line. */
    while (c != '\n' && c != EOF)
      c = fgetc (stdin);
    /* Obey the answer if it was valid. */
    if (answer == 'y')
      return 1;
    if (answer == 'n')
      return 0;
    /* Answer was invalid. Ask for valid answer. */
    fputs ("Please answer y or n:", stdout);
  }
}
```

`int` **getw** (`FILE *`*stream*) Function

This function reads a word (that is, an `int`) from *stream*. It's provided for compatibility with SVID. We recommend you use `fread` instead (see Section 17.11 [Block Input/Output], page 459). Unlike `getc`, any `int` value could be a valid result. `getw` returns `EOF` when it encounters end-of-file or an error, but there is no way to distinguish this from an input word with value -1.

17.9 Line-Oriented Input

Since many programs interpret input on the basis of lines, it is convenient to have functions to read a line of text from a stream.

Standard C has functions to do this, but they aren't very safe: null characters and even (for `gets`) long lines can confuse them. So the GNU library provides the nonstandard `getline` function that makes it easy to read lines reliably.

Another GNU extension, `getdelim`, generalizes `getline`. It reads a delimited record, defined as everything through the next occurrence of a specified delimiter character.

All these functions are declared in '`stdio.h`'.

`ssize_t` **getline** (`char **`*lineptr*`, size_t *`*n*`, FILE *`*stream*) Function

This function reads an entire line from *stream*, storing the text (including the newline and a terminating null character) in a buffer and storing the buffer address in `*`*lineptr*.

Before calling `getline`, you should place in `*`*lineptr* the address of a buffer `*`*n* bytes long, allocated with `malloc`. If this buffer is long enough to hold the line, `getline` stores the line in this buffer. Otherwise, `getline` makes the buffer bigger using `realloc`, storing the new buffer address back in `*`*lineptr*

and the increased size back in **n* (see Section 3.2.2 [Unconstrained Allocation], page 42).

If you set **lineptr* to a null pointer, and **n* to zero, before the call, then `getline` allocates the initial buffer for you by calling `malloc`.

In either case, when `getline` returns, **lineptr* is a `char *` that points to the text of the line.

When `getline` is successful, it returns the number of characters read (including the newline, but not including the terminating null). This value enables you to distinguish null characters that are part of the line from the null character inserted as a terminator.

This function is a GNU extension, but it is the recommended way to read lines from a stream. The alternative standard functions are unreliable.

If an error occurs or end of file is reached without any bytes read, `getline` returns `-1`.

`ssize_t` **getdelim** (`char **`*lineptr*`, size_t *`*n*`, int` *delimiter*`, FILE *`*stream*) Function

This function is like `getline`, except that the character that tells it to stop reading is not necessarily newline. The argument *delimiter* specifies the delimiter character; `getdelim` keeps reading until it sees that character (or end of file).

The text is stored in *lineptr*, including the delimiter character and a terminating null. Like `getline`, `getdelim` makes *lineptr* bigger if it isn't big enough.

`getline` is in fact implemented in terms of `getdelim`, just like this:

```
ssize_t
getline (char **lineptr, size_t *n, FILE *stream)
{
  return getdelim (lineptr, n, '\n', stream);
}
```

`char *` **fgets** (`char *`*s*`, int` *count*`, FILE *`*stream*) Function

The `fgets` function reads characters from the stream *stream* up to and including a newline character and stores them in the string *s*, adding a null character to mark the end of the string. You must supply *count* characters worth of space in *s*, but the number of characters read is at most *count* − 1. The extra character space is used to hold the null character at the end of the string.

If the system is already at end of file when you call `fgets`, then the contents of the array *s* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *s*.

Warning: If the input data has a null character, you can't tell. So don't use `fgets` unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message. We recommend using `getline` instead of `fgets`.

`wchar_t *` **fgetws** (`wchar_t *`*ws*`, int` *count*`, FILE *`*stream*) Function

The `fgetws` function reads wide characters from the stream *stream* up to and including a newline character and stores them in the string *ws*, adding a null wide character to mark the end of the string. You must supply *count* wide-characters worth of space in *ws*, but the number of characters read is at most *count* − 1. The extra character space is used to hold the null wide character at the end of the string.

If the system is already at end of file when you call `fgetws`, then the contents of the array *ws* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *ws*.

Warning: If the input data has a null wide character (which are null bytes in the input stream), you can't tell. So don't use `fgetws` unless you know the data cannot contain a null. Don't use it to read files edited by the user because, if the user inserts a null character, you should either handle it properly or print a clear error message.

`char *` **fgets_unlocked** (`char *`*s*`, int` *count*`, FILE *`*stream*) Function

The `fgets_unlocked` function is equivalent to the `fgets` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`wchar_t *` **fgetws_unlocked** (`wchar_t *`*ws*`, int` *count*`, FILE *`*stream*) Function

The `fgetws_unlocked` function is equivalent to the `fgetws` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`char *` **gets** (`char *`*s*) Deprecated function

The function `gets` reads characters from the stream `stdin` up to the next newline character, and stores them in the string *s*. The newline character is discarded (note that this differs from the behavior of `fgets`, which copies the newline character into the string). If `gets` encounters a read error or end of file, it returns a null pointer. Otherwise, it returns *s*.

Warning: The `gets` function is *very dangerous* because it provides no protection against overflowing the string *s*. The GNU library includes it for compatibility only. You should *always* use `fgets` or `getline` instead. To remind you of this, the linker (if using GNU `ld`) will issue a warning whenever you use `gets`.

17.10 Unreading

In parser programs, it is often useful to examine the next character in the input stream without removing it from the stream. This is called *peeking ahead* at the input because your program gets a glimpse of the input it will read next.

Using stream I/O, you can peek ahead at input by first reading it and then *unreading* it (also called *pushing it back* on the stream). Unreading a character makes it available to be input again from the stream, by the next call to `fgetc` or other input function on that stream.

17.10.1 What Unreading Means

Here is a pictorial explanation of unreading. Suppose you have a stream reading a file that contains just six characters, the letters '`foobar`'. Suppose you have read three characters so far. The situation looks like this:

```
f  o  o  b  a  r
      ^
```

so the next input character will be '`b`'.

If instead of reading '`b`', you unread the letter '`o`', you get a situation like this:

```
f  o  o  b  a  r
      |
     o--
     ^
```

so that the next input characters will be '`o`' and '`b`'.

If you unread '`9`' instead of '`o`', you get this situation:

```
f  o  o  b  a  r
      |
     9--
     ^
```

so that the next input characters will be '`9`' and '`b`'.

17.10.2 Using `ungetc` to Do Unreading

The function to unread a character is called `ungetc`, because it reverses the action of `getc`.

`int` **ungetc** (`int` *c*, `FILE *`*stream*) Function

The `ungetc` function pushes back the character *c* onto the input stream *stream*. So the next input from *stream* will read *c* before anything else.

If *c* is `EOF`, `ungetc` does nothing and just returns `EOF`. This lets you call `ungetc` with the return value of `getc` without needing to check for an error from `getc`.

The character that you push back doesn't have to be the same as the last character that was actually read from the stream. In fact, it isn't necessary to actually

read any characters from the stream before unreading them with `ungetc`! But that is a strange way to write a program; usually `ungetc` is used only to unread a character that was just read from the same stream. The GNU C Library supports this even on files opened in binary mode, but other systems might not.

The GNU C Library only supports one character of push back—in other words, it does not work to call `ungetc` twice without doing input in between. Other systems might let you push back multiple characters; then reading from the stream retrieves the characters in the reverse order that they were pushed.

Pushing back characters doesn't alter the file; only the internal buffering for the stream is affected. If a file-positioning function (such as `fseek`, `fseeko` or `rewind`) is called, any pending pushed-back characters are discarded (see Section 17.18 [File Positioning], page 500).

Unreading a character on a stream that is at end of file clears the end-of-file indicator for the stream, because it makes the character of input available. After you read that character, trying to read again will encounter end of file.

`wint_t` **ungetwc** (`wint_t` *wc*, `FILE *`*stream*) Function

The `ungetwc` function behaves just like `ungetc` just that it pushes back a wide character.

Here is an example showing the use of `getc` and `ungetc` to skip over white-space characters. When this function reaches a non-white-space character, it unreads that character to be seen again on the next read operation on the stream:

```
#include <stdio.h>
#include <ctype.h>

void
skip_whitespace (FILE *stream)
{
  int c;
  do
    /* No need to check for EOF because it is not
       isspace, and ungetc ignores EOF.  */
    c = getc (stream);
  while (isspace (c));
  ungetc (c, stream);
}
```

17.11 Block Input/Output

This section describes how to do input and output operations on blocks of data. You can use these functions to read and write binary data, as well as to read and write text in fixed-size blocks instead of by characters or lines.

Binary files are typically used to read and write blocks of data in the same format as is used to represent the data in a running program. In other words, arbitrary

blocks of memory—not just character or string objects—can be written to a binary file, and meaningfully read in again by the same program.

Storing data in binary form is often considerably more efficient than using the formatted I/O functions. Also, for floating-point numbers, the binary form avoids possible loss of precision in the conversion process. On the other hand, binary files can't be examined or modified easily using many standard file utilities (such as text editors), and are not portable between different implementations of the language, or different kinds of computers.

These functions are declared in '`stdio.h`'.

`size_t` **fread** (`void *`*data*`, size_t` *size*`, size_t` *count*`, FILE *`*stream*) Function

This function reads up to *count* objects of size *size* into the array *data*, from the stream *stream*. It returns the number of objects actually read, which might be less than *count* if a read error occurs or the end of the file is reached. This function returns a value of zero (and doesn't read anything) if either *size* or *count* is zero.

If `fread` encounters end of file in the middle of an object, it returns the number of complete objects read, and discards the partial object. Therefore, the stream remains at the actual end of the file.

`size_t` **fread_unlocked** (`void *`*data*`, size_t` *size*`, size_t` *count*`, FILE *`*stream*) Function

The `fread_unlocked` function is equivalent to the `fread` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

`size_t` **fwrite** (`const void *`*data*`, size_t` *size*`, size_t` *count*`, FILE *`*stream*) Function

This function writes up to *count* objects of size *size* from the array *data*, to the stream *stream*. The return value is normally *count*, if the call succeeds. Any other value indicates some sort of error, such as running out of space.

`size_t` **fwrite_unlocked** (`const void *`*data*`, size_t` *size*`, size_t` *count*`, FILE *`*stream*) Function

The `fwrite_unlocked` function is equivalent to the `fwrite` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

17.12 Formatted Output

The functions described in this section (`printf` and related functions) provide a convenient way to perform formatted output. You call `printf` with a *format string* or *template string* that specifies how to format the values of the remaining arguments.

Unless your program is a filter that specifically performs line- or character-oriented processing, using `printf` or one of the other related functions described in this section is usually the easiest and most concise way to perform output. These functions are especially useful for printing error messages, tables of data, and the like.

17.12.1 Formatted Output Basics

The `printf` function can be used to print any number of arguments. The template string argument you supply in a call provides information not only about the number of additional arguments, but also about their types and what style should be used for printing them.

Ordinary characters in the template string are simply written to the output stream as-is, while *conversion specifications* introduced by a '`%`' character in the template cause subsequent arguments to be formatted and written to the output stream. For example:

```
int pct = 37;
char filename[] = "foo.txt";
printf ("Processing of `%s' is %d%% finished.\nPlease be patient.\n",
        filename, pct);
```

produces output like:

```
Processing of `foo.txt' is 37% finished.
Please be patient.
```

This example shows the use of the '`%d`' conversion to specify that an `int` argument should be printed in decimal notation, the '`%s`' conversion to specify printing of a string argument, and the '`%%`' conversion to print a literal '`%`' character.

There are also conversions for printing an integer argument as an unsigned value in octal, decimal, or hexadecimal radix ('`%o`', '`%u`', or '`%x`', respectively); or as a character value ('`%c`').

Floating-point numbers can be printed in normal, fixed-point notation using the '`%f`' conversion or in exponential notation using the '`%e`' conversion. The '`%g`' conversion uses either '`%e`' or '`%f`' format, depending on what is more appropriate for the magnitude of the particular number.

You can control formatting more precisely by writing *modifiers* between the '`%`' and the character that indicates which conversion to apply. These slightly alter the ordinary behavior of the conversion. For example, most conversion specifications permit you to specify a minimum field width and a flag indicating whether you want the result left- or right-justified within the field.

The specific flags and modifiers that are permitted and their interpretation vary depending on the particular conversion. They're all described in more detail in the following sections. Don't worry if this all seems excessively complicated at first; you can almost always get reasonable free-format output without using any of the modifiers at all. The modifiers are mostly used to make the output look "prettier" in tables.

17.12.2 Output Conversion Syntax

This section provides details about the precise syntax of conversion specifications that can appear in a `printf` template string.

Characters in the template string that are not part of a conversion specification are printed as-is to the output stream. Multibyte-character sequences (see Chapter 6 [Character-Set Handling], page 133) are permitted in a template string.

The conversion specifications in a `printf` template string have the general form:

```
% [ param-no $] flags width [ . precision ] type conversion
```

or

```
% [ param-no $] flags width . * [ param-no $] type conversion
```

For example, in the conversion specifier '`%-10.8ld`', the '`-`' is a flag, '`10`' specifies the field width, the precision is '`8`', the letter '`l`' is a type modifier, and '`d`' specifies the conversion style. (This particular type specifier says to print a `long int` argument in decimal notation, with a minimum of eight digits left-justified in a field at least ten characters wide.)

In more detail, output conversion specifications consist of an initial '`%`' character followed in sequence by:

- An optional specification of the parameter used for this format. Normally, the parameters to the `printf` function are assigned to the formats in the order of appearance in the format string. But in some situations (such as message translation), this is not desirable, and this extension allows an explicit parameter to be specified.

  The *param-no* parts of the format must be integers in the range of 1 to the maximum number of arguments present to the function call. Some implementations limit this number to a certainly upper bound. The exact limit can be retrieved by the following constant.

  NL_ARGMAX Macro

  The value of `NL_ARGMAX` is the maximum value allowed for the specification of a positional parameter in a `printf` call. The actual value in effect at run time can be retrieved by using `sysconf` using the `_SC_NL_ARGMAX` parameter.[8]

  Some systems have quite a low limit, such as 9 for System V systems. The GNU C Library has no real limit.

  If any of the formats has a specification for the parameter position, all of them in the format string shall have one. Otherwise, the behavior is undefined.

- Zero or more *flag characters* that modify the normal behavior of the conversion specification

[8] Ibid., "Definition of `sysconf`".

- An optional decimal integer specifying the *minimum field width*; if the normal conversion produces fewer characters than this, the field is padded with spaces to the specified width. This is a *minimum* value; if the normal conversion produces more characters than this, the field is *not* truncated. Normally, the output is right-justified within the field.

  You can also specify a field width of '`*`'. This means that the next argument in the argument list (before the actual value to be printed) is used as the field width. The value must be an `int`. If the value is negative, this means to set the '`-`' flag (see below), and to use the absolute value as the field width.
- An optional *precision* to specify the number of digits to be written for the numeric conversions; if the precision is specified, it consists of a period ('`.`') followed optionally by a decimal integer (which defaults to zero if omitted).

  You can also specify a precision of '`*`'. This means that the next argument in the argument list (before the actual value to be printed) is used as the precision. The value must be an `int`, and is ignored if it is negative. If you specify '`*`' for both the field width and precision, the field width argument precedes the precision argument. Other C library versions may not recognize this syntax.
- An optional *type modifier character*, which is used to specify the data type of the corresponding argument if it differs from the default type; for example, the integer conversions assume a type of `int`, but you can specify '`h`', '`l`', or '`L`' for other integer types.
- A character that specifies the conversion to be applied

The exact options that are permitted and how they are interpreted vary between the different conversion specifiers. See the descriptions of the individual conversions for information about the particular options that they use.

With the '`-Wformat`' option, the GNU C Compiler checks calls to `printf` and related functions. It examines the format string and verifies that the correct number and types of arguments are supplied. There is also a GNU C syntax to tell the compiler that a function you write uses a `printf`-style format string.[9]

17.12.3 Table of Output Conversions

Here is a table summarizing what all the different conversions do:

'`%d`', '`%i`' Print an integer as a signed decimal number (see Section 17.12.4 [Integer Conversions], page 465). '`%d`' and '`%i`' are synonymous for output, but are different when used with `scanf` for input (see Section 17.14.3 [Table of Input Conversions], page 489).

'`%o`' Print an integer as an unsigned octal number (see Section 17.12.4 [Integer Conversions], page 465).

[9] See Richard M. Stallman and the GCC Developer Community, "Decaring Attributes of Functions" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), `http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/`.

‘%u’ Print an integer as an unsigned decimal number (see Section 17.12.4 [Integer Conversions], page 465).

‘%x’, ‘%X’ Print an integer as an unsigned hexadecimal number. ‘%x’ uses lowercase letters and ‘%X’ uses uppercase (see Section 17.12.4 [Integer Conversions], page 465).

‘%f’ Print a floating-point number in normal (fixed-point) notation (see Section 17.12.5 [Floating-Point Conversions], page 467).

‘%e’, ‘%E’ Print a floating-point number in exponential notation. ‘%e’ uses lowercase letters and ‘%E’ uses uppercase (see Section 17.12.5 [Floating-Point Conversions], page 467).

‘%g’, ‘%G’ Print a floating-point number in either normal or exponential notation, whichever is more appropriate for its magnitude. ‘%g’ uses lowercase letters and ‘%G’ uses uppercase (see Section 17.12.5 [Floating-Point Conversions], page 467).

‘%a’, ‘%A’ Print a floating-point number in a hexadecimal fractional notation which the exponent to base 2 represented in decimal digits. ‘%a’ uses lowercase letters and ‘%A’ uses uppercase (see Section 17.12.5 [Floating-Point Conversions], page 467).

‘%c’ Print a single character (see Section 17.12.6 [Other Output Conversions], page 469).

‘%C’ This is an alias for ‘%lc’ that is supported for compatibility with the Unix standard.

‘%s’ Print a string (see Section 17.12.6 [Other Output Conversions], page 469).

‘%S’ This is an alias for ‘%ls’ that is supported for compatibility with the Unix standard.

‘%p’ Print the value of a pointer (see Section 17.12.6 [Other Output Conversions], page 469).

‘%n’ Get the number of characters printed so far (see Section 17.12.6 [Other Output Conversions], page 469). This conversion specification never produces any output.

‘%m’ Print the string corresponding to the value of `errno` (see Section 17.12.6 [Other Output Conversions], page 469). This is a GNU extension.

‘%%’ Print a literal ‘%’ character (see Section 17.12.6 [Other Output Conversions], page 469).

If the syntax of a conversion specification is invalid, unpredictable things will happen, so don’t do this. If there aren’t enough function arguments provided to supply values for all of the conversion specifications in the template string, or if

the arguments are not of the correct types, the results are unpredictable. If you supply more arguments than conversion specifications, the extra argument values are simply ignored; this is sometimes useful.

17.12.4 Integer Conversions

This section describes the options for the '`%d`', '`%i`', '`%o`', '`%u`', '`%x`' and '`%X`' conversion specifications. These conversions print integers in various formats.

The '`%d`' and '`%i`' conversion specifications both print an `int` argument as a signed decimal number; while '`%o`', '`%u`' and '`%x`' print the argument as an unsigned octal, decimal, or hexadecimal number, respectively. The '`%X`' conversion specification is just like '`%x`', except that it uses the characters '`ABCDEF`' as digits instead of '`abcdef`'.

The following flags are meaningful:

'`-`' Left-justify the result in the field (instead of the normal right-justification).

'`+`' For the signed '`%d`' and '`%i`' conversions, print a plus sign if the value is positive.

'` `' For the signed '`%d`' and '`%i`' conversions, if the result doesn't start with a plus or minus sign, prefix it with a space character instead. Since the '`+`' flag ensures that the result includes a sign, this flag is ignored if you supply both of them.

'`#`' For the '`%o`' conversion, this forces the leading digit to be '`0`', as if by increasing the precision. For '`%x`' or '`%X`', this prefixes a leading '`0x`' or '`0X`', respectively, to the result. This doesn't do anything useful for the '`%d`', '`%i`', or '`%u`' conversions. Using this flag produces output that can be parsed by the `strtoul` function (see Section 9.11.1 [Parsing of Integers], page 268) and `scanf` with the '`%i`' conversion (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`'`' Separate the digits into groups as specified by the locale specified for the `LC_NUMERIC` category (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187). This flag is a GNU extension.

'`0`' Pad the field with zeros instead of spaces. The zeros are placed after any indication of sign or base. This flag is ignored if the '`-`' flag is also specified, or if a precision is specified.

If a precision is supplied, it specifies the minimum number of digits to appear; leading zeros are produced if necessary. If you don't specify a precision, the number is printed with as many digits as it needs. If you convert a value of zero with an explicit precision of zero, then no characters at all are produced.

Without a type modifier, the corresponding argument is treated as an `int` (for the signed conversions '`%i`' and '`%d`') or `unsigned int` (for the unsigned conversions '`%o`', '`%u`', '`%x`' and '`%X`'). Recall that since `printf` and friends are

variadic, any `char` and `short` arguments are automatically converted to `int` by the default argument promotions. For arguments of other integer types, you can use these modifiers:

'`hh`' This modifier specifies that the argument is a `signed char` or `unsigned char`, as appropriate. A `char` argument is converted to an `int` or `unsigned int` by the default argument promotions anyway, but the '`h`' modifier says to convert it back to a `char` again.

This modifier was introduced in ISO C99.

'`h`' This modifier specifies that the argument is a `short int` or `unsigned short int`, as appropriate. A `short` argument is converted to an `int` or `unsigned int` by the default argument promotions anyway, but the '`h`' modifier says to convert it back to a `short` again.

'`j`' This modifier specifies that the argument is an `intmax_t` or `uintmax_t`, as appropriate.

It was introduced in ISO C99.

'`l`' This modifier specifies that the argument is a `long int` or `unsigned long int`, as appropriate. Two '`l`' characters is like the '`L`' modifier, below.

If used with '`%c`' or '`%s`', the corresponding parameter is considered as a wide character or wide-character string respectively. This use of '`l`' was introduced in Amendment 1 to ISO C90.

'`L`'
'`ll`'
'`q`' This modifier specifies that the argument is a `long long int`. (This type is an extension supported by the GNU C Compiler. On systems that don't support extra-long integers, this is the same as `long int`.)

The '`q`' modifier is another name for the same thing, which comes from 4.4 BSD; a `long long int` is sometimes called a *quad* `int`.

'`t`' This modifier specifies that the argument is a `ptrdiff_t`.

It was introduced in ISO C99.

'`z`'
'`Z`' This modifier specifies that the argument is a `size_t`.

'`z`' was introduced in ISO C99. '`Z`' is a GNU extension pre-dating this addition and should not be used in new code.

Here is an example. Using the template string:

```
"|%5d|%-5d|%+5d|%+-5d|% 5d|%05d|%5.0d|%5.2d|%d|\n"
```

to print numbers using the different options for the '`%d`' conversion gives results like:

```
|    0|0    |   +0|+0   |    0|00000|     |   00|0|
|    1|1    |   +1|+1   |    1|00001|    1|   01|1|
|   -1|-1   |   -1|-1   |   -1|-0001|   -1|  -01|-1|
|100000|100000|+100000|+100000| 100000|100000|100000|100000|100000|
```

In particular, notice what happens in the last case where the number is too large to fit in the minimum field width specified.

Here are some more examples showing how unsigned integers print under various format options, using the template string:

```
"|%5u|%5o|%5x|%5X|%#5o|%#5x|%#5X|%#10.8x|\n"
|    0|    0|    0|    0|    0|    0|    0|  00000000|
|    1|    1|    1|    1|   01|  0x1|  0X1|0x00000001|
|100000|303240|186a0|186A0|0303240|0x186a0|0X186A0|0x000186a0|
```

17.12.5 Floating-Point Conversions

This section discusses the conversion specifications for floating-point numbers: the '`%f`', '`%e`', '`%E`', '`%g`' and '`%G`' conversions.

The '`%f`' conversion prints its argument in fixed-point notation, producing output of the form [−]*ddd*`.`*ddd*, where the number of digits following the decimal point is controlled by the precision you specify.

The '`%e`' conversion prints its argument in exponential notation, producing output of the form [−]*d*`.`*ddd*`e`[`+`|`−`]*dd*. Again, the number of digits following the decimal point is controlled by the precision. The exponent always contains at least two digits. The '`%E`' conversion is similar, but the exponent is marked with the letter '`E`' instead of '`e`'.

The '`%g`' and '`%G`' conversions print the argument in the style of '`%e`' or '`%E`', respectively, if the exponent would be less than -4 or greater than or equal to the precision. Otherwise, they use the '`%f`' style. A precision of `0`, is taken as 1. is Trailing zeros are removed from the fractional portion of the result, and a decimal-point character appears only if it is followed by a digit.

The '`%a`' and '`%A`' conversions are meant for representing floating-point numbers exactly in textual form so that they can be exchanged as texts between different programs and/or machines. The numbers are represented in the form [−]`0x`*h*`.`*hhh*`p`[`+`|`−`]*dd*. At the left of the decimal-point character exactly one digit is printed. This character is only `0` if the number is de-normalized. Otherwise, the value is unspecified; it is implementation dependent how many bits are used. The number of hexadecimal digits on the right side of the decimal-point character is equal to the precision. If the precision is zero, it is determined to be large enough to provide an exact representation of the number or it is large enough to distinguish two adjacent values if the `FLT_RADIX` is not a power of 2.[10] For the '`%a`' conversion, lowercase characters are used to represent the hexadecimal number, and the prefix and exponent sign are printed as `0x` and `p` respectively. Otherwise, uppercase characters are used, and `0X` and `P` are used for the representation of prefix and

[10] Ibid., "Floating-Point Parameters".

exponent string. The exponent to the base of two is printed as a decimal number using at least one digit but at most as many digits as necessary to represent the value exactly.

If the value to be printed represents infinity or a NaN, the output is `[-]inf` or `nan`, respectively, if the conversion specifier is '`%a`', '`%e`', '`%f`' or '`%g`'. The output is `[-]INF` or `NAN`, respectively, if the conversion is '`%A`', '`%E`' or '`%G`'.

The following flags can be used to modify the behavior:

'`-`' Left-justify the result in the field. Normally the result is right-justified.

'`+`' Always include a plus or minus sign in the result.

'` `' If the result doesn't start with a plus or minus sign, prefix it with a space instead. Since the '`+`' flag ensures that the result includes a sign, this flag is ignored if you supply both of them.

'`#`' Specify that the result should always include a decimal point, even if no digits follow it. For the '`%g`' and '`%G`' conversions, this also forces trailing zeros after the decimal point to be left in place where they would otherwise be removed.

'`'`' Separate the digits of the integer part of the result into groups as specified by the locale specified for the `LC_NUMERIC` category (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187). This flag is a GNU extension.

'`0`' Pad the field with zeros instead of spaces; the zeros are placed after any sign. This flag is ignored if the '`-`' flag is also specified.

The precision specifies how many digits follow the decimal-point character for the '`%f`', '`%e`' and '`%E`' conversions. For these conversions, the default precision is `6`. If the precision is explicitly `0`, this suppresses the decimal-point character entirely. For the '`%g`' and '`%G`' conversions, the precision specifies how many significant digits to print. Significant digits are the first digit before the decimal point and all the digits after it. If the precision is `0` or not specified for '`%g`' or '`%G`', it is treated like a value of `1`. If the value being printed cannot be expressed accurately in the specified number of digits, the value is rounded to the nearest number that fits.

Without a type modifier, the floating-point conversions use an argument of type `double`. (By the default argument promotions, any `float` arguments are automatically converted to `double`.) The following type modifier is supported:

'`L`' An uppercase '`L`' specifies that the argument is a `long double`.

Here are some examples showing how numbers print using the various floating-point conversions. All of the numbers were printed using this template string:

```
"|%13.4a|%13.4f|%13.4e|%13.4g|\n"
```

Here is the output:

```
|  0x0.0000p+0|       0.0000|   0.0000e+00|            0|
|  0x1.0000p-1|       0.5000|   5.0000e-01|          0.5|
```

```
|  0x1.0000p+0|       1.0000|   1.0000e+00|            1|
| -0x1.0000p+0|      -1.0000|  -1.0000e+00|           -1|
|  0x1.9000p+6|     100.0000|   1.0000e+02|          100|
|  0x1.f400p+9|    1000.0000|   1.0000e+03|         1000|
| 0x1.3880p+13|   10000.0000|   1.0000e+04|        1e+04|
| 0x1.81c8p+13|   12345.0000|   1.2345e+04|    1.234e+04|
| 0x1.86a0p+16|  100000.0000|   1.0000e+05|        1e+05|
| 0x1.e240p+16|  123456.0000|   1.2346e+05|    1.235e+05|
```

Notice how the ‘%g’ conversion drops trailing zeros.

17.12.6 Other Output Conversions

This section describes miscellaneous conversions for printf.

The ‘%c’ conversion prints a single character. In case there is no ‘l’ modifier, the int argument is first converted to an unsigned char. Then, if used in a wide stream function, the character is converted into the corresponding wide character. The ‘-’ flag can be used to specify left-justification in the field, but no other flags are defined, and no precision or type modifier can be given. For example:

```
printf ("%c%c%c%c%c", 'h', 'e', 'l', 'l', 'o');
```

prints ‘hello’.

If there is a ‘l’ modifier present, the argument is expected to be of type wint_t. If used in a multibyte function, the wide character is converted into a multibyte character before being added to the output. In this case, more than 1 output byte can be produced.

The ‘%s’ conversion prints a string. If no ‘l’ modifier is present, the corresponding argument must be of type char * (or const char *). If used in a wide stream function the string is first converted in a wide-character string. A precision can be specified to indicate the maximum number of characters to write. Otherwise, characters in the string up to but not including the terminating null character are written to the output stream. The ‘-’ flag can be used to specify left-justification in the field, but no other flags or type modifiers are defined for this conversion. For example:

```
printf ("%3s%-6s", "no", "where");
```

prints ‘ nowhere ’.

If there is a ‘l’ modifier present, the argument is expected to be of type wchar_t (or const wchar_t *).

If you accidentally pass a null pointer as the argument for a ‘%s’ conversion, the GNU library prints it as ‘(null)’. We think this is more useful than crashing. But it’s not good practice to pass a null argument intentionally.

The ‘%m’ conversion prints the string corresponding to the error code in errno (see Section 2.3 [Error Messages], page 32). Thus:

```
fprintf (stderr, "can't open '%s': %m\n", filename);
```

is equivalent to:

```
fprintf (stderr, "can't open '%s': %s\n", filename, strerror (errno));
```

The '%m' conversion is a GNU C Library extension.

The '%p' conversion prints a pointer value. The corresponding argument must be of type void *. In practice, you can use any type of pointer.

In the GNU system, nonnull pointers are printed as unsigned integers, as if a '%#x' conversion were used. Null pointers print as '(nil)'. (Pointers might print differently in other systems.)

For example:

```
printf ("%p", "testing");
```

prints '0x' followed by a hexadecimal number—the address of the string constant "testing". It does not print the word 'testing'.

You can supply the '-' flag with the '%p' conversion to specify left-justification, but no other flags, precision, or type modifiers are defined.

The '%n' conversion is unlike any of the other output conversions. It uses an argument that must be a pointer to an int, but instead of printing anything, it stores the number of characters printed so far by this call at that location. The 'h' and 'l' type modifiers are permitted to specify that the argument is of type short int * or long int * instead of int *, but no flags, field width, or precision are permitted.

For example:

```
int nchar;
printf ("%d %s%n\n", 3, "bears", &nchar);
```

prints:

```
3 bears
```

and sets nchar to 7, because '3 bears' is seven characters.

The '%%' conversion prints a literal '%' character. This conversion doesn't use an argument, and no flags, field width, precision, or type modifiers are permitted.

17.12.7 Formatted Output Functions

This section describes how to call printf and related functions. Prototypes for these functions are in the header file 'stdio.h'. Because these functions take a variable number of arguments, you *must* declare prototypes for them before using them. Of course, the easiest way to make sure you have all the right prototypes is to just include 'stdio.h'.

int **printf** (const char *template, ...) Function

The printf function prints the optional arguments under the control of the template string *template* to the stream stdout. It returns the number of characters printed, or a negative value if there was an output error.

`int` **wprintf** (`const wchar_t *`*template*`, ...`) Function

The `wprintf` function prints the optional arguments under the control of the wide template string *template* to the stream `stdout`. It returns the number of wide characters printed, or a negative value if there was an output error.

`int` **fprintf** (`FILE *`*stream*`, const char *`*template*`, ...`) Function

This function is just like `printf`, except that the output is written to the stream *stream* instead of `stdout`.

`int` **fwprintf** (`FILE *`*stream*`, const wchar_t *`*template*`, ...`) Function

This function is just like `wprintf`, except that the output is written to the stream *stream* instead of `stdout`.

`int` **sprintf** (`char *`*s*`, const char *`*template*`, ...`) Function

This is like `printf`, except that the output is stored in the character array *s* instead of written to a stream. A null character is written to mark the end of the string.

The `sprintf` function returns the number of characters stored in the array *s*, not including the terminating null character.

The behavior of this function is undefined if copying takes place between objects that overlap—for example, if *s* is also given as an argument to be printed under control of the '`%s`' conversion (see Section 5.4 [Copying and Concatenation], page 93).

Warning: The `sprintf` function can be *dangerous* because it can potentially output more characters than can fit in the allocation size of the string *s*. Remember that the field width given in a conversion specification is only a *minimum* value.

To avoid this problem, you can use `snprintf` or `asprintf`, described below.

`int` **swprintf** (`wchar_t *`*s*`, size_t` *size*`, const wchar_t *`*template*`, ...`) Function

This is like `wprintf`, except that the output is stored in the wide-character array *ws* instead of written to a stream. A null wide character is written to mark the end of the string. The *size* argument specifies the maximum number of characters to produce. The trailing null character is counted toward this limit, so you should allocate at least *size* wide characters for the string *ws*.

The return value is the number of characters generated for the given input, excluding the trailing null. If not all output fits into the provided buffer, a negative value is returned. You should try again with a bigger output string. This is different from how `snprintf` handles this situation.

The corresponding narrow stream function takes fewer parameters. `swprintf` in fact corresponds to the `snprintf` function. Since the `sprintf` function can be dangerous and should be avoided, the ISO C committee refused to make

the same mistake again and decided to not define a function exactly corresponding to `sprintf`.

`int` **snprintf** (`char *`*s*`, size_t` *size*`, const char *`*template*`, ...`) Function

The `snprintf` function is similar to `sprintf`, except that the *size* argument specifies the maximum number of characters to produce. The trailing null character is counted toward this limit, so you should allocate at least *size* characters for the string *s*.

The return value is the number of characters that would be generated for the given input, excluding the trailing null. If this value is greater or equal to *size*, not all characters from the result have been stored in *s*. You should try again with a bigger output string. Here is an example of doing this:

```
/* Construct a message describing the value of a variable
   whose name is name and whose value is value. */
char *
make_message (char *name, char *value)
{
  /* Guess we need no more than 100 chars of space. */
  int size = 100;
  char *buffer = (char *) xmalloc (size);
  int nchars;


  if (buffer == NULL)
    return NULL;

 /* Try to print in the allocated space. */
  nchars = snprintf (buffer, size, "value of %s is %s",
                     name, value);


  if (nchars >= size)
    {
      /* Reallocate buffer now that we know
    how much space is needed. */
      buffer = (char *) xrealloc (buffer, nchars + 1);

      if (buffer != NULL)
        /* Try again. */
        snprintf (buffer, size, "value of %s is %s",
                  name, value);
    }
```

```
  /* The last call worked, return the string. */
  return buffer;
}
```

In practice, it is often easier just to use `asprintf` (see Section 17.12.8 [Dynamically Allocating Formatted Output], page 473).

Attention: In versions of the GNU C Library prior to 2.1, the return value is the number of characters stored, not including the terminating null; unless there was not enough space in *s* to store the result, in which case `-1` is returned. This was changed in order to comply with the ISO C99 standard.

17.12.8 Dynamically Allocating Formatted Output

The functions in this section do formatted output and place the results in dynamically allocated memory.

`int` **asprintf** (`char **`*ptr*`, const char *`*template*`, ...`) Function

This function is similar to `sprintf`, except that it dynamically allocates a string (as with `malloc`; see Section 3.2.2 [Unconstrained Allocation], page 42) to hold the output, instead of putting the output in a buffer you allocate in advance. The *ptr* argument should be the address of a `char *` object, and `asprintf` stores a pointer to the newly allocated string at that location.

The return value is the number of characters allocated for the buffer, or less than zero if an error occurred. Usually this means that the buffer could not be allocated.

Here is how to use `asprintf` to get the same result as the `snprintf` example, but more easily:

```
/* Construct a message describing the value of a variable
   whose name is name and whose value is value. */
char *
make_message (char *name, char *value)
{
  char *result;
  if (asprintf (&result, "value of %s is %s", name, value) < 0)
    return NULL;
  return result;
}
```

`int` **obstack_printf** (`struct obstack *`*obstack*`, const char *`*template*`, ...`) Function

This function is similar to `asprintf`, except that it uses the obstack *obstack* to allocate the space (see Section 3.2.4 [Obstacks], page 59).

The characters are written onto the end of the current object. To get at them, you must finish the object with `obstack_finish` (see Section 3.2.4.6 [Growing Objects], page 64).

17.12.9 Variable Arguments Output Functions

The functions `vprintf` and friends are provided so that you can define your own variadic `printf`-like functions that make use of the same internals as the built-in formatted output functions.

The most natural way to define such functions would be to use a language construct to say, "Call `printf` and pass this template plus all of my arguments after the first five." But there is no way to do this in C, and it would be hard to provide a way, since at the C language level, there is no way to tell how many arguments your function received.

Since that method is impossible, we provide alternative functions, the `vprintf` series, which lets you pass a `va_list` to describe "all of my arguments after the first five."

When it is sufficient to define a macro rather than a real function, the GNU C Compiler provides a way to do this much more easily with macros.[11]

For example:

```
#define myprintf(a, b, c, d, e, rest...) \
            printf (mytemplate , ## rest)
```

See section "Macros with Variable Numbers of Arguments" in *Using GNU CC*, for details. But this is limited to macros, and does not apply to real functions at all.

Before calling `vprintf` or the other functions listed in this section, you *must* call `va_start` to initialize a pointer to the variable arguments.[12] Then you can call `va_arg` to fetch the arguments that you want to handle yourself. This advances the pointer past those arguments.

Once your `va_list` pointer is pointing at the argument of your choice, you are ready to call `vprintf`. That argument and all subsequent arguments that were passed to your function are used by `vprintf` along with the template that you specified separately.

In some other systems, the `va_list` pointer may become invalid after the call to `vprintf`, so you must not use `va_arg` after you call `vprintf`. Instead, you should call `va_end` to retire the pointer from service. However, you can safely call `va_start` on another pointer variable and begin fetching the arguments again through that pointer. Calling `vprintf` does not destroy the argument list of your function, merely the particular pointer that you passed to it.

GNU C does not have such restrictions. You can safely continue to fetch arguments from a `va_list` pointer after passing it to `vprintf`, and `va_end` is a no-op. Note, however, that subsequent `va_arg` calls will fetch the same arguments that `vprintf` previously used.

Prototypes for these functions are declared in '`stdio.h`'.

---

[11] See Richard M. Stallman and the GCC Developer Community, "Macros with a Variable Number of Arguments" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/.

[12] Ibid., "Variadic Functions".

`int` **vprintf** (`const char *`*template*`, va_list` *ap*) Function
This function is similar to `printf` except that, instead of taking a variable number of arguments directly, it takes an argument list pointer *ap*.

`int` **vwprintf** (`const wchar_t *`*template*`, va_list` *ap*) Function
This function is similar to `wprintf` except that, instead of taking a variable number of arguments directly, it takes an argument list pointer *ap*.

`int` **vfprintf** (`FILE *`*stream*`, const char *`*template*`,` Function
`va_list` *ap*)
This is the equivalent of `fprintf` with the variable argument list specified directly as for `vprintf`.

`int` **vfwprintf** (`FILE *`*stream*`, const wchar_t *`*template*`,` Function
`va_list` *ap*)
This is the equivalent of `fwprintf` with the variable argument list specified directly as for `vwprintf`.

`int` **vsprintf** (`char *`*s*`, const char *`*template*`, va_list` Function
ap)
This is the equivalent of `sprintf` with the variable argument list specified directly as for `vprintf`.

`int` **vswprintf** (`wchar_t *`*s*`, size_t` *size*`, const wchar_t` Function
`*`*template*`, va_list` *ap*)
This is the equivalent of `swprintf` with the variable argument list specified directly as for `vwprintf`.

`int` **vsnprintf** (`char *`*s*`, size_t` *size*`, const char` Function
`*`*template*`, va_list` *ap*)
This is the equivalent of `snprintf` with the variable argument list specified directly as for `vprintf`.

`int` **vasprintf** (`char **`*ptr*`, const char *`*template*`,` Function
`va_list` *ap*)
The `vasprintf` function is the equivalent of `asprintf` with the variable argument list specified directly as for `vprintf`.

`int` **obstack_vprintf** (`struct obstack *`*obstack*`, const` Function
`char *`*template*`, va_list` *ap*)
The `obstack_vprintf` function is the equivalent of `obstack_printf` with the variable argument list specified directly as for `vprintf`.

Here is an example showing how you might use `vfprintf`. This is a function that prints error messages to the stream `stderr`, along with a prefix indicating the name of the program (see Section 2.3 [Error Messages], page 32, for a description of `program_invocation_short_name`).

```
#include <stdio.h>
#include <stdarg.h>

void
eprintf (const char *template, ...)
{
  va_list ap;
  extern char *program_invocation_short_name;

  fprintf (stderr, "%s: ", program_invocation_short_name);
  va_start (ap, template);
  vfprintf (stderr, template, ap);
  va_end (ap);
}
```

You could call `eprintf` like this:

```
eprintf ("file `%s' does not exist\n", filename);
```

In GNU C, there is a special construct you can use to let the compiler know that a function uses a `printf`-style format string. Then it can check the number and types of arguments in each call to the function, and warn you when they do not match the format string. For example, take this declaration of `eprintf`:

```
void eprintf (const char *template, ...)
        __attribute__ ((format (printf, 1, 2)));
```

This tells the compiler that `eprintf` uses a format string like `printf` (as opposed to `scanf`; see Section 17.14 [Formatted Input], page 486); the format string appears as the first argument and the arguments to satisfy the format begin with the second.[13]

17.12.10 Parsing a Template String

You can use the function `parse_printf_format` to obtain information about the number and types of arguments that are expected by a given template string. This function permits interpreters that provide interfaces to `printf` to avoid passing along invalid arguments from the user's program, which could cause a crash.

[13] See Richard M. Stallman and the GCC Developer Community, "Declaring Attributes of Functions" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/.

All the symbols described in this section are declared in the header file 'printf.h'.

`size_t` **parse_printf_format** (`const char *`*template*`,` `size_t` *n*`,` `int *`*argtypes*) Function

This function returns information about the number and types of arguments expected by the `printf` template string *template*. The information is stored in the array *argtypes*; each element of this array describes one argument. This information is encoded using the various 'PA_' macros, listed below.

The argument *n* specifies the number of elements in the array *argtypes*. This is the maximum number of elements that `parse_printf_format` will try to write.

`parse_printf_format` returns the total number of arguments required by *template*. If this number is greater than *n*, then the information returned describes only the first *n* arguments. If you want information about additional arguments, allocate a bigger array and call `parse_printf_format` again.

The argument types are encoded as a combination of a basic type and modifier flag bits.

`int` **PA_FLAG_MASK** Macro

This macro is a bitmask for the type modifier flag bits. You can write the expression `(argtypes[i] & PA_FLAG_MASK)` to extract just the flag bits for an argument, or `(argtypes[i] & ~PA_FLAG_MASK)` to extract just the basic type code.

Here are symbolic constants that represent the basic types; they stand for integer values:

`PA_INT` This specifies that the base type is `int`.

`PA_CHAR` This specifies that the base type is `int`, cast to `char`.

`PA_STRING`
: This specifies that the base type is `char *`, a null-terminated string.

`PA_POINTER`
: This specifies that the base type is `void *`, an arbitrary pointer.

`PA_FLOAT`
: This specifies that the base type is `float`.

`PA_DOUBLE`
: This specifies that the base type is `double`.

`PA_LAST` You can define additional base types for your own programs as offsets from `PA_LAST`. For example, if you have data types '`foo`' and '`bar`' with their own specialized `printf` conversions, you could define encodings for these types as:

```
#define PA_FOO  PA_LAST
#define PA_BAR  (PA_LAST + 1)
```

Here are the flag bits that modify a basic type. They are combined with the code for the basic type using inclusive-or.

`PA_FLAG_PTR`
: If this bit is set, it indicates that the encoded type is a pointer to the base type, rather than an immediate value. For example, '`PA_INT|PA_FLAG_PTR`' represents the type '`int *`'.

`PA_FLAG_SHORT`
: If this bit is set, it indicates that the base type is modified with `short`. This corresponds to the '`h`' type modifier.

`PA_FLAG_LONG`
: If this bit is set, it indicates that the base type is modified with `long`. This corresponds to the '`l`' type modifier.

`PA_FLAG_LONG_LONG`
: If this bit is set, it indicates that the base type is modified with `long long`. This corresponds to the '`L`' type modifier.

`PA_FLAG_LONG_DOUBLE`
: This is a synonym for `PA_FLAG_LONG_LONG`, used by convention with a base type of `PA_DOUBLE` to indicate a type of `long double`.

17.12.11 Example of Parsing a Template String

Here is an example of decoding argument types for a format string. We assume this is part of an interpreter that contains arguments of type `NUMBER`, `CHAR`, `STRING` and `STRUCTURE` (and perhaps others that are not valid here).

```
/* Test whether the nargs specified objects
   in the vector args are valid
   for the format string format;
   if so, return 1.
   If not, return 0 after printing an error message.  */

int
validate_args (char *format, int nargs, OBJECT *args)
{
  int *argtypes;
  int nwanted;

  /* Get the information about the arguments.
     Each conversion specification must be at least two characters
     long, so there cannot be more specifications than half the
```

```
    length of the string.  */

argtypes = (int *) alloca (strlen (format) / 2 * sizeof (int));
nwanted = parse_printf_format (string, nelts, argtypes);

/* Check the number of arguments.  */
if (nwanted > nargs)
  {
    error ("too few arguments (at least %d required)", nwanted);
    return 0;
  }

/* Check the C type wanted for each argument
    and see if the object given is suitable.  */
for (i = 0; i < nwanted; i++)
  {
    int wanted;

    if (argtypes[i] & PA_FLAG_PTR)
      wanted = STRUCTURE;
    else
      switch (argtypes[i] & ~PA_FLAG_MASK)
        {
        case PA_INT:
        case PA_FLOAT:
        case PA_DOUBLE:
          wanted = NUMBER;
          break;
        case PA_CHAR:
          wanted = CHAR;
          break;
        case PA_STRING:
          wanted = STRING;
          break;
        case PA_POINTER:
          wanted = STRUCTURE;
          break;
        }
    if (TYPE (args[i]) != wanted)
      {
        error ("type mismatch for arg number %d", i);
        return 0;
      }
  }
```

```
    return 1;
}
```

17.13 Customizing `printf`

The GNU C Library lets you define your own custom conversion specifiers for `printf` template strings, to teach `printf` clever ways to print the important data structures of your program.

The way you do this is by registering the conversion with the function `register_printf_function` (see Section 17.13.1 [Registering New Conversions], page 480). One of the arguments you pass to this function is a pointer to a handler function that produces the actual output (see Section 17.13.3 [Defining the Output Handler], page 482, for information on how to write this function).

You can also install a function that just returns information about the number and type of arguments expected by the conversion specifier (see Section 17.12.10 [Parsing a Template String], page 476).

The facilities of this section are declared in the header file '`printf.h`'.

Portability Note: The ability to extend the syntax of `printf` template strings is a GNU extension. ISO standard C has nothing similar.

17.13.1 Registering New Conversions

The function to register a new output conversion is `register_printf_function`, declared in '`printf.h`'.

`int` **register_printf_function** (`int` *spec*, `printf_function` *handler-function*, `printf_arginfo_function` *arginfo-function*) — Function

This function defines the conversion specifier character *spec*. Thus, if *spec* is `'Y'`, it defines the conversion '`%Y`'. You can redefine the built-in conversions like '`%s`', but flag characters like '`#`' and type modifiers like '`l`' can never be used as conversions; calling `register_printf_function` for those characters has no effect. It is advisable not to use lowercase letters, since the ISO C standard warns that additional lowercase letters may be standardized in future editions of the standard.

The *handler-function* is the function called by `printf` and friends when this conversion appears in a template string (see Section 17.13.3 [Defining the Output Handler], page 482, for information about how to define a function to pass as this argument). If you specify a null pointer, any existing handler function for *spec* is removed.

The *arginfo-function* is the function called by `parse_printf_format` when this conversion appears in a template string (see Section 17.12.10 [Parsing a Template String], page 476).

Attention: In the GNU C Library versions before 2.0, the *arginfo-function* function did not need to be installed unless the user used the `parse_printf_format` function. This has changed. Now a call to any of the `printf` functions will call this function when this format specifier appears in the format string.

The return value is `0` on success, and `-1` on failure (which occurs if *spec* is out of range).

You can redefine the standard output conversions, but this is probably not a good idea because of the potential for confusion. Library routines written by other people could break if you do this.

17.13.2 Conversion Specifier Options

If you define a meaning for '`%A`', what if the template contains '`%+23A`' or '`%-#A`'? To implement a sensible meaning for these, the handler, when called, needs to be able to get the options specified in the template.

Both the *handler-function* and *arginfo-function* accept an argument that points to a `struct printf_info`, which contains information about the options appearing in an instance of the conversion specifier. This data type is declared in the header file '`printf.h`'.

struct printf_info Type

This structure is used to pass information about the options appearing in an instance of a conversion specifier in a `printf` template string to the handler and arginfo functions for that specifier. It contains the following members:

`int prec` This is the precision specified. The value is `-1` if no precision was specified. If the precision was given as '`*`', the `printf_info` structure passed to the handler function contains the actual value retrieved from the argument list. But the structure passed to the arginfo function contains a value of `INT_MIN`, since the actual value is not known.

`int width`
This is the minimum field width specified. The value is `0` if no width was specified. If the field width was given as '`*`', the `printf_info` structure passed to the handler function contains the actual value retrieved from the argument list. But the structure passed to the arginfo function contains a value of `INT_MIN`, since the actual value is not known.

`wchar_t spec`
This is the conversion specifier character specified. It's stored in the structure so that you can register the same handler function for multiple characters, but still have a way to tell them apart when the handler function is called.

`unsigned int is_long_double`
: This is a Boolean that is true if the '`L`', '`ll`', or '`q`' type modifier was specified. For integer conversions, this indicates `long long int`, as opposed to `long double` for floating-point conversions.

`unsigned int is_char`
: This is a Boolean that is true if the '`hh`' type modifier was specified.

`unsigned int is_short`
: This is a Boolean that is true if the '`h`' type modifier was specified.

`unsigned int is_long`
: This is a Boolean that is true if the '`l`' type modifier was specified.

`unsigned int alt`
: This is a Boolean that is true if the '`#`' flag was specified.

`unsigned int space`
: This is a Boolean that is true if the ' ' flag was specified.

`unsigned int left`
: This is a Boolean that is true if the '`-`' flag was specified.

`unsigned int showsign`
: This is a Boolean that is true if the '`+`' flag was specified.

`unsigned int group`
: This is a Boolean that is true if the '`'`' flag was specified.

`unsigned int extra`
: This flag has a special meaning depending on the context. It could be used freely by the user-defined handlers, but when called from the `printf` function, this variable always contains the value `0`.

`unsigned int wide`
: This flag is set if the stream is wide oriented.

`wchar_t pad`
: This is the character to use for padding the output to the minimum field width. The value is `'0'` if the '`0`' flag was specified, and `' '` otherwise.

17.13.3 Defining the Output Handler

Now let's look at how to define the handler and arginfo functions that are passed as arguments to `register_printf_function`.

Compatibility Note: The interface changed in GNU libc version 2.0. Previously, the third argument was of type `va_list *`.

You should define your handler functions with a prototype like:

```
int function (FILE *stream, const struct printf_info *info,
              const void *const *args)
```

The *stream* argument passed to the handler function is the stream to which it should write output.

The *info* argument is a pointer to a structure that contains information about the various options that were included with the conversion in the template string. You should not modify this structure inside your handler function (see Section 17.13.2 [Conversion Specifier Options], page 481, for a description of this data structure).

The *args* is a vector of pointers to the arguments data. The number of arguments was determined by calling the argument information function provided by the user.

Your handler function should return a value just like `printf` does—it should return the number of characters it has written, or a negative value to indicate an error.

printf_function Data Type

This is the data type that a handler function should have.

If you are going to use `parse_printf_format` in your application, you must also define a function to pass as the *arginfo-function* argument for each new conversion you install with `register_printf_function`.

You have to define these functions with a prototype like:

```
int function (const struct printf_info *info,
              size_t n, int *argtypes)
```

The return value from the function should be the number of arguments the conversion expects. The function should also fill in no more than *n* elements of the *argtypes* array with information about the types of each of these arguments. This information is encoded using the various '`PA_`' macros. (This is the same calling convention that `parse_printf_format` itself uses.)

printf_arginfo_function Data Type

This type is used to describe functions that return information about the number and type of arguments used by a conversion specifier.

17.13.4 `printf` Extension Example

Here is an example showing how to define a `printf` handler function. This program defines a data structure called a `Widget` and defines the '`%W`' conversion to print information about `Widget *` arguments, including the pointer value and the name stored in the data structure. The '`%W`' conversion supports the minimum field width and left-justification options, but ignores everything else.

```
#include <stdio.h>
#include <stdlib.h>
#include <printf.h>
```

```
typedef struct
{
  char *name;
}
Widget;


int
print_widget (FILE *stream,
              const struct printf_info *info,
              const void *const *args)
{
  const Widget *w;
  char *buffer;
  int len;

  /* Format the output into a string.  */
  w = *((const Widget **) (args[0]));
  len = asprintf (&buffer, "<Widget %p: %s>", w, w->name);
  if (len == -1)
    return -1;

  /* Pad to the minimum field width and print to the stream.  */
  len = fprintf (stream, "%*s",
                 (info->left ? -info->width : info->width),
                 buffer);

  /* Clean up and return.  */
  free (buffer);
  return len;
}


int
print_widget_arginfo (const struct printf_info *info, size_t n,
                      int *argtypes)
{
  /* We always take exactly one argument and this is a pointer to the
  structure.  */
  if (n > 0)
    argtypes[0] = PA_POINTER;
  return 1;
}
```

```
int
main (void)
{
  /* Make a widget to print. */
  Widget mywidget;
  mywidget.name = "mywidget";

  /* Register the print function for widgets. */
  register_printf_function ('W', print_widget, print_widget_arginfo);

  /* Now print the widget. */
  printf ("|%W|\n", &mywidget);
  printf ("|%35W|\n", &mywidget);
  printf ("|%-35W|\n", &mywidget);

  return 0;
}
```

The output produced by this program looks like:

```
|<Widget 0xffeffb7c: mywidget>|
|      <Widget 0xffeffb7c: mywidget>|
|<Widget 0xffeffb7c: mywidget>      |
```

17.13.5 Predefined `printf` Handlers

The GNU libc also contains a concrete and useful application of the `printf` handler extension. There are two functions available that implement a special way to print floating-point numbers.

`int` **printf_size** (`FILE` **fp*, `const struct printf_info` **info*, `const void *const` **args*) Function

Print a given floating-point number as for the format `%f`, except that there is a postfix character indicating the divisor for the number to make this less than 1000. There are two possible divisors: powers of 1024 or powers of 1000. Which one is used depends on the format character specified while registering this handler. If the character is lowercase, 1024 is used. For uppercase characters, 1000 is used.

The postfix tag corresponds to bytes, kilobytes, megabytes, gigabytes, etc. The full table is

low	Multiplier	From	Upper	Multiplier
␣	1		␣	1
k	$2^{10} = 1024$	kilo	K	$10^3 = 1000$
m	2^{20}	mega	M	10^6
g	2^{30}	giga	G	10^9
t	2^{40}	tera	T	10^{12}
p	2^{50}	peta	P	10^{15}
e	2^{60}	exa	E	10^{18}
z	2^{70}	zetta	Z	10^{21}
y	2^{80}	yotta	Y	10^{24}

The default precision is 3, i.e., 1024 is printed with a lowercase format character as if it were `%.3fk` and will yield `1.000k`.

Due to the requirements of `register_printf_function`, we must also provide the function that returns information about the arguments.

`int` **printf_size_info** (`const struct printf_info *`*info*`,` `size_t` *n*`,` `int *`*argtypes*) Function

This function will return in *argtypes* the information about the used parameters in the way the `vfprintf` implementation expects it. The format always takes one argument.

To use these functions, both functions must be registered with a call like:

```
register_printf_function ('B', printf_size, printf_size_info);
```

Here we register the functions to print numbers as powers of 1000, since the format character `'B'` is an uppercase character. If we would additionally use `'b'` in a line like:

```
register_printf_function ('b', printf_size, printf_size_info);
```

we could also print using a power of 1024. All that is different in these two lines is the format specifier. The `printf_size` function knows about the difference between lowercase and uppercase format specifiers.

The use of `'B'` and `'b'` is no coincidence. It is the preferred way to use this functionality, since it is available on some other systems that also use format specifiers.

17.14 Formatted Input

The functions described in this section (`scanf` and related functions) provide facilities for formatted input analogous to the formatted output facilities. These functions provide a mechanism for reading arbitrary values under the control of a *format string* or *template string*.

17.14.1 Formatted Input Basics

Calls to `scanf` are superficially similar to calls to `printf`, in that arbitrary arguments are read under the control of a template string. While the syntax of

the conversion specifications in the template is very similar to that for `printf`, the interpretation of the template is oriented more toward free-format input and simple pattern matching, rather than fixed-field formatting. For example, most `scanf` conversions skip over any amount of white space (including spaces, tabs and newlines) in the input file, and there is no concept of precision for the numeric input conversions as there is for the corresponding output conversions. Ordinarily, non-white-space characters in the template are expected to match characters in the input stream exactly, but a matching failure is distinct from an input error on the stream.

Another area of difference between `scanf` and `printf` is that you must remember to supply pointers rather than immediate values as the optional arguments to `scanf`; the values that are read are stored in the objects that the pointers point to. Even experienced programmers tend to forget this occasionally, so if your program is getting strange errors that seem to be related to `scanf`, you might want to double-check this.

When a *matching failure* occurs, `scanf` returns immediately, leaving the first nonmatching character as the next character to be read from the stream. The normal return value from `scanf` is the number of values that were assigned, so you can use this to determine if a matching error happened before all the expected values were read.

The `scanf` function is typically used for things like reading in the contents of tables. For example, here is a function that uses `scanf` to initialize an array of `double`:

```
void
readarray (double *array, int n)
{
  int i;
  for (i=0; i<n; i++)
    if (scanf (" %lf", &(array[i])) != 1)
      invalid_input_error ();
}
```

The formatted input functions are not used as frequently as the formatted output functions. Partly, this is because it takes some care to use them properly. Another reason is that it is difficult to recover from a matching error.

If you are trying to read input that doesn't match a single, fixed pattern, you may be better off using a tool such as Flex to generate a lexical scanner,[14] or Bison to generate a parser,[15] rather than using `scanf`.

---

[14] See G.T. Nicol, *Flex: The Lexical Scanner Generator* (Boston, MA: GNU Press, February 1993), http://www.gnu.org/software/flex/manual/.

[15] See Charles Donnelly and Richard M. Stallman, *The Bison Manual* (Boston, MA: GNU Press, September 2003), http://www.gnu.org/software/bison/manual/.

17.14.2 Input Conversion Syntax

A `scanf` template string is a string that contains ordinary multibyte characters interspersed with conversion specifications that start with '`%`'.

Any white-space character (as defined by the `isspace` function; see Section 4.1 [Classification of Characters], page 79) in the template causes any number of white-space characters in the input stream to be read and discarded. The white-space characters that are matched need not be exactly the same white-space characters that appear in the template string. For example, write '`, `' in the template to recognize a comma with optional white space before and after.

Other characters in the template string that are not part of conversion specifications must match characters in the input stream exactly; if this is not the case, a matching failure occurs.

The conversion specifications in a `scanf` template string have the general form:

`%` *flags width type conversion*

In more detail, an input conversion specification consists of an initial '`%`' character followed in sequence by:

- An optional *flag character* '`*`', which says to ignore the text read for this specification; when `scanf` finds a conversion specification that uses this flag, it reads input as directed by the rest of the conversion specification, but it discards this input, does not use a pointer argument, and does not increment the count of successful assignments.
- An optional flag character '`a`' (valid with string conversions only), which requests allocation of a buffer long enough to store the string in (see Section 17.14.6 [Dynamically Allocating String Conversions], page 494). This is a GNU extension.
- An optional decimal integer that specifies the *maximum field width*; reading of characters from the input stream stops either when this maximum is reached or when a nonmatching character is found, whichever happens first. Most conversions discard initial white-space characters (those that don't are explicitly documented), and these discarded characters don't count toward the maximum field width. String input conversions store a null character to mark the end of the input; the maximum field width does not include this terminator.
- An optional *type modifier character*; for example, you can specify a type modifier of '`l`' with integer conversions such as '`%d`' to specify that the argument is a pointer to a `long int` rather than a pointer to an `int`.
- A character that specifies the conversion to be applied

The exact options that are permitted and how they are interpreted vary between the different conversion specifiers. See the descriptions of the individual conversions for information about the particular options that they allow.

With the '`-Wformat`' option, the GNU C Compiler checks calls to `scanf` and related functions. It examines the format string and verifies that the correct number

and types of arguments are supplied. There is also a GNU C syntax to tell the compiler that a function you write uses a `scanf`-style format string.[16]

17.14.3 Table of Input Conversions

Here is a table that summarizes the various conversion specifications:

'`%d`' Match an optionally signed integer written in decimal (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`%i`' Match an optionally signed integer in any of the formats that the C language defines for specifying an integer constant (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`%o`' Match an unsigned integer written in octal radix (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`%u`' Match an unsigned integer written in decimal radix (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`%x`', '`%X`' Match an unsigned integer written in hexadecimal radix (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`%e`', '`%f`', '`%g`', '`%E`', '`%G`'
Match an optionally signed floating-point number (see Section 17.14.4 [Numeric Input Conversions], page 490).

'`%s`'
Match a string containing only non-white-space characters (see Section 17.14.5 [String Input Conversions], page 492). The presence of the '`l`' modifier determines whether the output is stored as a wide-character string or a multibyte string. If '`%s`' is used in a wide-character function, the string is converted as with multiple calls to `wcrtomb` into a multibyte string. This means that the buffer must provide room for `MB_CUR_MAX` bytes for each wide character read. In case '`%ls`' is used in a multibyte function, the result is converted into wide characters, as with multiple calls of `mbrtowc`, before being stored in the user-provided buffer.

'`%S`' This is an alias for '`%ls`' that is supported for compatibility with the Unix standard.

'`%[`' Match a string of characters that belong to a specified set (see Section 17.14.5 [String Input Conversions], page 492). The presence of the '`l`' modifier determines whether the output is stored as a wide-character string or a multibyte string. If '`%[`' is used in a wide-character function, the string is converted as with multiple calls to

[16] See Richard M. Stallman and the GCC Developer Community, "Declaring Attributes of Functions" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), `http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/`.

wcrtomb into a multibyte string. This means that the buffer must provide room for MB_CUR_MAX bytes for each wide character read. In case '%l[' is used in a multibyte function, the result is converted into wide characters, as with multiple calls of mbrtowc, before being stored in the user-provided buffer.

'%c' Match a string of one or more characters; the number of characters read is controlled by the maximum field width given for the conversion (see Section 17.14.5 [String Input Conversions], page 492).

If the '%c' is used in a wide stream function, the read value is converted from a wide character to the corresponding multibyte character before storing it. This conversion can produce more than 1 byte of output, and therefore the provided buffer must be large enough for up to MB_CUR_MAX bytes for each character. If '%lc' is used in a multibyte function, the input is treated as a multibyte sequence (and not bytes), and the result is converted as with calls to mbrtowc.

'%C' This is an alias for '%lc' that is supported for compatibility with the Unix standard.

'%p' Match a pointer value in the same implementation-defined format used by the '%p' output conversion for printf (see Section 17.14.7 [Other Input Conversions], page 494).

'%n' This conversion doesn't read any characters; it records the number of characters read so far by this call (see Section 17.14.7 [Other Input Conversions], page 494).

'%%' This matches a literal '%' character in the input stream. No corresponding argument is used (see Section 17.14.7 [Other Input Conversions], page 494).

If the syntax of a conversion specification is invalid, the behavior is undefined. If there aren't enough function arguments provided to supply addresses for all the conversion specifications in the template strings that perform assignments, or if the arguments are not of the correct types, the behavior is also undefined. On the other hand, extra arguments are simply ignored.

17.14.4 Numeric Input Conversions

This section describes the scanf conversions for reading numeric values.

The '%d' conversion matches an optionally signed integer in decimal radix. The syntax that is recognized is the same as that for the strtol function (see Section 9.11.1 [Parsing of Integers], page 268) with the value 10 for the *base* argument.

The '%i' conversion matches an optionally signed integer in any of the formats that the C language defines for specifying an integer constant. The syntax that is recognized is the same as that for the strtol function (see Section 9.11.1 [Parsing of Integers], page 268) with the value 0 for the *base* argument. You can print

integers in this syntax with `printf` by using the '`#`' flag character with the '`%x`', '`%o`', or '`%d`' conversion (see Section 17.12.4 [Integer Conversions], page 465).

For example, any of the strings '`10`', '`0xa`' or '`012`' could be read in as integers under the '`%i`' conversion. Each of these specifies a number with decimal value `10`.

The '`%o`', '`%u`' and '`%x`' conversions match unsigned integers in octal, decimal and hexadecimal radices, respectively. The syntax that is recognized is the same as that for the `strtoul` function (see Section 9.11.1 [Parsing of Integers], page 268) with the appropriate value (`8`, `10`, or `16`) for the *base* argument.

The '`%X`' conversion is identical to the '`%x`' conversion. They both permit either uppercase or lowercase letters to be used as digits.

The default type of the corresponding argument for the `%d` and `%i` conversions is `int *`, and `unsigned int *` for the other integer conversions. You can use the following type modifiers to specify other sizes of integer:

'`hh`' Specify that the argument is a `signed char *` or `unsigned char *`.

This modifier was introduced in ISO C99.

'`h`' Specify that the argument is a `short int *` or `unsigned short int *`.

'`j`' Specify that the argument is a `intmax_t *` or `uintmax_t *`.

This modifier was introduced in ISO C99.

'`l`' Specify that the argument is a `long int *` or `unsigned long int *`. Two '`l`' characters is like the '`L`' modifier, below.

If used with '`%c`' or '`%s`', the corresponding parameter is considered as a pointer to a wide character or wide-character string respectively. This use of '`l`' was introduced in Amendment 1 to ISO C90.

'`ll`'
'`L`'
'`q`' Specify that the argument is a `long long int *` or `unsigned long long int *` (the `long long` type is an extension supported by the GNU C Compiler. For systems that don't provide extra-long integers, this is the same as `long int`).

The '`q`' modifier is another name for the same thing, which comes from 4.4 BSD; a `long long int` is sometimes called a *quad* `int`.

'`t`' Specify that the argument is a `ptrdiff_t *`.

This modifier was introduced in ISO C99.

'`z`' Specify that the argument is a `size_t *`.

This modifier was introduced in ISO C99.

All of the '`%e`', '`%f`', '`%g`', '`%E`' and '`%G`' input conversions are interchangeable. They all match an optionally signed floating-point number, in the same syntax as for the `strtod` function (see Section 9.11.2 [Parsing of Floats], page 273).

For the floating-point input conversions, the default argument type is `float *`. This is different from the corresponding output conversions, where the default type is `double`; remember that `float` arguments to `printf` are converted to `double` by the default argument promotions, but `float *` arguments are not promoted to `double *`. You can specify other sizes of float using these type modifiers:

'`l`' Specify that the argument is of type `double *`.

'`L`' Specify that the argument is of type `long double *`.

For all of the above number-parsing formats, there is an additional optional flag '`'`'. When this flag is given, the `scanf` function expects the number represented in the input string to be formatted according to the grouping rules of the currently selected locale (see Section 7.6.1.1 [Generic Numeric Formatting Parameters], page 187).

If the '`C`' or '`POSIX`' locale is selected, there is no difference. But for a locale that specifies values for the appropriate fields in the locale, the input must have the correct form in the input. Otherwise, the longest prefix with a correct form is processed.

17.14.5 String Input Conversions

This section describes the `scanf` input conversions for reading string and character values: '`%s`', '`%S`', '`%[`', '`%c`' and '`%C`'.

You have two options for how to receive the input from these conversions:

- Provide a buffer to store it in. This is the default. You should provide an argument of type `char *` or `wchar_t *` (the latter if the '`l`' modifier is present).

  Warning: To make a robust program, you must make sure that the input (plus its terminating null) cannot possibly exceed the size of the buffer you provide. In general, the only way to do this is to specify a maximum field width one less than the buffer size. *If you provide the buffer, always specify a maximum field width to prevent overflow.*

- Ask `scanf` to allocate a big enough buffer, by specifying the '`a`' flag character. This is a GNU extension. You should provide an argument of type `char **` for the buffer address to be stored in (see Section 17.14.6 [Dynamically Allocating String Conversions], page 494).

The '`%c`' conversion is the simplest—it matches a fixed number of characters, always. The maximum field width says how many characters to read; if you don't specify the maximum, the default is 1. This conversion doesn't append a null character to the end of the text it reads. It also does not skip over initial white-space characters. It reads precisely the next *n* characters, and fails if it cannot get that many. Since there is always a maximum field width with '`%c`' (whether specified, or 1 by default), you can always prevent overflow by making the buffer long enough.

If the format is '`%lc`' or '`%C`', the function stores wide characters that are converted using the conversion determined at the time the stream was opened from the external byte stream. The number of bytes read from the medium is limited by `MB_CUR_LEN` * *n*, but at most *n* wide characters get stored in the output string.

The '`%s`' conversion matches a string of non-white-space characters. It skips and discards initial white space, but stops when it encounters more white space after having read something. It stores a null character at the end of the text that it reads.

For example, reading the input:

```
hello, world
```

with the conversion '`%10c`' produces `" hello, wo"`, but reading the same input with the conversion '`%10s`' produces `"hello,"`.

Warning: If you do not specify a field width for '`%s`', then the number of characters read is limited only by where the next white-space character appears. This almost certainly means that invalid input can make your program crash—which is a bug.

The '`%ls`' and '`%S`' format are handled just like '`%s`', except that the external byte-sequence is converted using the conversion associated with the stream to wide characters with their own encoding. A width or precision specified with the format do not directly determine how many bytes are read from the stream since they measure wide characters. But an upper limit can be computed by multiplying the value of the width or precision by `MB_CUR_MAX`.

To read in characters that belong to an arbitrary set of your choice, use the '`%[`' conversion. You specify the set between the '`[`' character and a following '`]`' character, using the same syntax used in regular expressions. As special cases:

- A literal '`]`' character can be specified as the first character of the set.
- An embedded '`-`' character (that is, one that is not the first or last character of the set) is used to specify a range of characters.
- If a caret character '`^`' immediately follows the initial '`[`', then the set of allowed input characters is the everything *except* the characters listed.

The '`%[`' conversion does not skip over initial white-space characters.

Here are some examples of '`%[`' conversions and what they mean:

'`%25[1234567890]`'
: Match a string of up to twenty-five digits.

'`%25[][]`'
: Match a string of up to twenty-five square brackets.

'`%25[^ \f\n\r\t\v]`'
: Match a string up to twenty-five characters long that doesn't contain any of the standard white-space characters. This is slightly different from '`%s`', because if the input begins with a white-space character, '`%[`' reports a matching failure while '`%s`' simply discards the initial white space.

‘%25[a-z]’
: Match up to twenty-five lowercase characters.

As for ‘%c’ and ‘%s’, the ‘%[’ format is also modified to produce wide characters if the ‘l’ modifier is present. All that was said about ‘%ls’ above is true for ‘%l[’.

The ‘%s’ and ‘%[’ conversions are *dangerous* if you don’t specify a maximum width or use the ‘a’ flag, because input too long would overflow whatever buffer you have provided for it. No matter how long your buffer is, a user could supply input that is longer. A well-written program reports invalid input with a comprehensible error message, not with a crash.

17.14.6 Dynamically Allocating String Conversions

A GNU extension to formatted input lets you safely read a string with no maximum size. Using this feature, you don’t supply a buffer; instead, `scanf` allocates a buffer big enough to hold the data and gives you its address. To use this feature, write ‘a’ as a flag character, as in ‘%as’ or ‘%a[0-9a-z]’.

The pointer argument you supply for where to store the input should have type `char **`. The `scanf` function allocates a buffer and stores its address in the word that the argument points to. You should free the buffer with `free` when you no longer need it.

Here is an example of using the ‘a’ flag with the ‘%[...]’ conversion specification to read a *variable assignment* of the form ‘*variable* = *value*’.

```
{
  char *variable, *value;

  if (2 > scanf ("%a[a-zA-Z0-9] = %a[^\n]\n",
                 &variable, &value))
    {
      invalid_input_error ();
      return 0;
    }

  ...
}
```

17.14.7 Other Input Conversions

This section describes the miscellaneous input conversions.

The ‘%p’ conversion is used to read a pointer value. It recognizes the same syntax used by the ‘%p’ output conversion for `printf` (see Section 17.12.6 [Other Output Conversions], page 469); that is, a hexadecimal number, just as the ‘%x’ conversion accepts. The corresponding argument should be of type `void **`; that is, the address of a place to store a pointer.

The resulting pointer value is not guaranteed to be valid if it was not originally written during the same program execution that reads it in.

The '`%n`' conversion produces the number of characters read so far by this call. The corresponding argument should be of type `int *`. This conversion works in the same way as the '`%n`' conversion for `printf` (see Section 17.12.6 [Other Output Conversions], page 469, for an example).

The '`%n`' conversion is the only mechanism for determining the success of literal matches or conversions with suppressed assignments. If the '`%n`' follows the locus of a matching failure, then no value is stored for it since `scanf` returns before processing the '`%n`'. If you store `-1` in that argument slot before calling `scanf`, the presence of `-1` after `scanf` indicates an error occurred before the '`%n`' was reached.

Finally, the '`%%`' conversion matches a literal '`%`' character in the input stream, without using an argument. This conversion does not permit any flags, field width, or type modifier to be specified.

17.14.8 Formatted Input Functions

Here are the descriptions of the functions for performing formatted input. Prototypes for these functions are in the header file '`stdio.h`'.

`int` **scanf** (`const char *`*template*`, ...`) Function

The `scanf` function reads formatted input from the stream `stdin` under the control of the template string *template*. The optional arguments are pointers to the places that receive the resulting values.

The return value is normally the number of successful assignments. If an end-of-file condition is detected before any matches are performed, including matches against white-space and literal characters in the template, then `EOF` is returned.

`int` **wscanf** (`const wchar_t *`*template*`, ...`) Function

The `wscanf` function reads formatted input from the stream `stdin` under the control of the template string *template*. The optional arguments are pointers to the places which receive the resulting values.

The return value is normally the number of successful assignments. If an end-of-file condition is detected before any matches are performed, including matches against white-space and literal characters in the template, then `WEOF` is returned.

`int` **fscanf** (`FILE *`*stream*`, const char *`*template*`, ...`) Function

This function is just like `scanf`, except that the input is read from the stream *stream* instead of `stdin`.

`int` **fwscanf** (`FILE *`*stream*`, const wchar_t *`*template*`, ...`) Function

This function is just like `wscanf`, except that the input is read from the stream *stream* instead of `stdin`.

`int` **sscanf** (`const char *`*s*`, const char *`*template*`, ...`) Function
This is like `scanf`, except that the characters are taken from the null-terminated string *s* instead of from a stream. Reaching the end of the string is treated as an end-of-file condition.

The behavior of this function is undefined if copying takes place between objects that overlap—for example, if *s* is also given as an argument to receive a string read under control of the '`%s`', '`%S`', or '`%[`' conversion.

`int` **swscanf** (`const wchar_t *`*ws*`, const char *`*template*`, ...`) Function
This is like `wscanf`, except that the characters are taken from the null-terminated string *ws* instead of from a stream. Reaching the end of the string is treated as an end-of-file condition.

The behavior of this function is undefined if copying takes place between objects that overlap—for example, if *ws* is also given as an argument to receive a string read under control of the '`%s`', '`%S`', or '`%[`' conversion.

17.14.9 Variable Arguments Input Functions

The functions `vscanf` and friends are provided so that you can define your own variadic `scanf`-like functions that make use of the same internals as the built-in formatted output functions. These functions are analogous to the `vprintf` series of output functions (see Section 17.12.9 [Variable Arguments Output Functions], page 474, for important information on how to use them).

Portability Note: The functions listed in this section were introduced in ISO C99 and were previously available as GNU extensions.

`int` **vscanf** (`const char *`*template*`, va_list` *ap*) Function
This function is similar to `scanf`, but instead of taking a variable number of arguments directly, it takes an argument list pointer *ap* of type `va_list`.[17]

`int` **vwscanf** (`const wchar_t *`*template*`, va_list` *ap*) Function
This function is similar to `wscanf`, but instead of taking a variable number of arguments directly, it takes an argument list pointer *ap* of type `va_list`.[18]

`int` **vfscanf** (`FILE *`*stream*`, const char *`*template*`, va_list` *ap*) Function
This is the equivalent of `fscanf` with the variable argument list specified directly as for `vscanf`.

`int` **vfwscanf** (`FILE *`*stream*`, const wchar_t *`*template*`, va_list` *ap*) Function
This is the equivalent of `fwscanf` with the variable argument list specified directly as for `vwscanf`.

[17] Ibid., "Variadic Functions".
[18] Ibid., "Variadic Functions".

`int` **vsscanf** (`const char *`*s*`, const char *`*template*`, va_list` *ap*) Function

This is the equivalent of `sscanf` with the variable argument list specified directly as for `vscanf`.

`int` **vswscanf** (`const wchar_t *`*s*`, const wchar_t *`*template*`, va_list` *ap*) Function

This is the equivalent of `swscanf` with the variable argument list specified directly as for `vwscanf`.

In GNU C, there is a special construct you can use to let the compiler know that a function uses a `scanf`-style format string. Then it can check the number and types of arguments in each call to the function, and warn you when they do not match the format string.[19]

17.15 End-of-File and Errors

Many of the functions described in this chapter return the value of the macro `EOF` to indicate unsuccessful completion of the operation. Since `EOF` is used to report both end-of-file and random errors, it's often better to use the `feof` function to check explicitly for end of file and `ferror` to check for errors. These functions check indicators that are part of the internal state of the stream object, indicators set if the appropriate condition was detected by a previous I/O operation on that stream.

`int` **EOF** Macro

This macro is an integer value that is returned by a number of narrow stream functions to indicate an end-of-file condition, or some other error situation. With the GNU library, `EOF` is `-1`. In other libraries, its value may be some other negative number.

This symbol is declared in '`stdio.h`'.

`int` **WEOF** Macro

This macro is an integer value that is returned by a number of wide stream functions to indicate an end-of-file condition, or some other error situation. With the GNU library, `WEOF` is `-1`. In other libraries, its value may be some other negative number.

This symbol is declared in '`wchar.h`'.

`int` **feof** (`FILE *`*stream*) Function

The `feof` function returns nonzero if and only if the end-of-file indicator for the stream *stream* is set.

This symbol is declared in '`stdio.h`'.

[19] See Richard M. Stallman and the GCC Developer Community, "Declaring Attributes of Functions" in *Using GCC: The GNU Compiler Collection Reference Manual* (Boston, MA: GNU Press, October 2003), `http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/`.

`int` **feof_unlocked** (`FILE *`*stream*) Function

The `feof_unlocked` function is equivalent to the `feof` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

This symbol is declared in '`stdio.h`'.

`int` **ferror** (`FILE *`*stream*) Function

The `ferror` function returns nonzero if and only if the error indicator for the stream *stream* is set, indicating that an error has occurred on a previous operation on the stream.

This symbol is declared in '`stdio.h`'.

`int` **ferror_unlocked** (`FILE *`*stream*) Function

The `ferror_unlocked` function is equivalent to the `ferror` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

This symbol is declared in '`stdio.h`'.

In addition to setting the error indicator associated with the stream, the functions that operate on streams also set `errno` in the same way as the corresponding low-level functions that operate on file descriptors. For example, all of the functions that perform output to a stream—such as `fputc`, `printf` and `fflush`—are implemented in terms of `write`, and all of the `errno` error conditions defined for `write` are meaningful for these functions.[20]

17.16 Recovering from Errors

You may explicitly clear the error and EOF flags with the `clearerr` function.

`void` **clearerr** (`FILE *`*stream*) Function

This function clears the end-of-file and error indicators for the stream *stream*.

The file-positioning functions (see Section 17.18 [File Positioning], page 500) also clear the end-of-file indicator for the stream.

`void` **clearerr_unlocked** (`FILE *`*stream*) Function

The `clearerr_unlocked` function is equivalent to the `clearerr` function, except that it does not implicitly lock the stream.

This function is a GNU extension.

It is *not* correct to just clear the error flag and retry a failed stream operation. After a failed write, any number of characters since the last buffer flush may have

[20] For more information about the descriptor-level I/O functions, see Loosemore et al., "Low-Level Input/Output".

been committed to the file, while some buffered data may have been discarded. Merely retrying can thus cause lost or repeated data.

A failed read may leave the file pointer in an inappropriate position for a second try. In both cases, you should seek to a known position before retrying.

Most errors that can happen are not recoverable—a second try will always fail again in the same way. So usually it is best to give up and report the error to the user, rather than install complicated recovery logic.

One important exception is `EINTR`.[21] Many stream I/O implementations will treat it as an ordinary error, which can be quite inconvenient. You can avoid this hassle by installing all signals with the `SA_RESTART` flag.

For similar reasons, setting nonblocking I/O on a stream's file descriptor is not usually advisable.

17.17 Text and Binary Streams

The GNU system and other POSIX-compatible operating systems organize all files as uniform sequences of characters. However, some other systems make a distinction between files containing text and files containing binary data, and the input and output facilities of ISO C provide for this distinction. This section tells you how to write programs portable to such systems.

When you open a stream, you can specify either a *text stream* or a *binary stream*. You indicate that you want a binary stream by specifying the '`b`' modifier in the *opentype* argument to `fopen` (see Section 17.3 [Opening Streams], page 440). Without this option, `fopen` opens the file as a text stream.

Text and binary streams differ in several ways:

- The data read from a text stream is divided into *lines* that are terminated by newline (`'\n'`) characters, while a binary stream is simply a long series of characters. A text stream might on some systems fail to handle lines more than 254 characters long (including the terminating newline character).
- On some systems, text files can contain only printing characters, horizontal tab characters, and newlines, and so text streams may not support other characters. However, binary streams can handle any character value.
- Space characters that are written immediately preceding a newline character in a text stream may disappear when the file is read in again.
- More generally, there need not be a one-to-one mapping between characters that are read from or written to a text stream, and the characters in the actual file.

Since a binary stream is always more capable and more predictable than a text stream, you might wonder what purpose text streams serve. Why not simply always use binary streams? The answer is that on these operating systems, text and binary streams use different file formats, and the only way to read or write "an ordinary file of text" that can work with other text-oriented programs is through a text stream.

[21] Ibid., "Primitives Interrupted by Signals".

In the GNU library, and on all POSIX systems, there is no difference between text streams and binary streams. When you open a stream, you get the same kind of stream regardless of whether you ask for binary. This stream can handle any file content, and has none of the restrictions that text streams sometimes have.

17.18 File Positioning

The *file position* of a stream describes where in the file the stream is currently reading or writing. I/O on the stream advances the file position through the file. In the GNU system, the file position is represented as an integer, which counts the number of bytes from the beginning of the file (see Section 15.1.2 [File Position], page 430).

During I/O to an ordinary disk file, you can change the file position whenever you wish, so as to read or write any portion of the file. Some other kinds of files may also permit this. Files that support changing the file position are sometimes referred to as *random-access* files.

You can use the functions in this section to examine or modify the file position indicator associated with a stream. The symbols listed below are declared in the header file ‘`stdio.h`’.

`long int` **ftell** (`FILE *`*stream*) Function

This function returns the current file position of the stream *stream*.

This function can fail if the stream doesn’t support file positioning, or if the file position can’t be represented in a `long int`, and possibly for other reasons as well. If a failure occurs, a value of `-1` is returned.

`off_t` **ftello** (`FILE *`*stream*) Function

The `ftello` function is similar to `ftell`, except that it returns a value of type `off_t`. Systems that support this type use it to describe all file positions, unlike the POSIX specification, which uses a long int. The two are not necessarily the same size. Therefore, using `ftell` can lead to problems if the implementation is written on top of a POSIX compliant low-level I/O implementation, and using `ftello` is preferable whenever it is available.

If this function fails, it returns `(off_t) -1`. This can happen due to missing support for file positioning or internal errors. Otherwise, the return value is the current file position.

The function is an extension defined in the Unix Single Specification version 2.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `ftello64`—the LFS interface transparently replaces the old interface.

`off64_t` **ftello64** (`FILE *`*stream*) Function

This function is similar to `ftello` except that the return value is of type `off64_t`. This also requires that the stream *stream* was opened using either

`fopen64`, `freopen64` or `tmpfile64`, since otherwise the underlying file operations to position the file pointer beyond the 2^31 bytes limit might fail.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `ftello` and so transparently replaces the old interface.

`int` **fseek** (`FILE *`*stream*`, long int` *offset*`, int` *whence*) Function

The `fseek` function is used to change the file position of the stream *stream*. The value of *whence* must be one of the constants `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, to indicate whether the *offset* is relative to the beginning of the file, the current file position, or the end of the file, respectively.

This function returns a value of zero if the operation was successful, and a nonzero value to indicate failure. A successful call also clears the end-of-file indicator of *stream* and discards any characters that were pushed back by the use of `ungetc`.

`fseek` either flushes any buffered output before setting the file position, or else remembers it so it will be written later in its proper place in the file.

`int` **fseeko** (`FILE *`*stream*`, off_t` *offset*`, int` *whence*) Function

This function is similar to `fseek`, but it corrects a problem with `fseek` in a system with POSIX types. Using a value of type `long int` for the offset is not compatible with POSIX. `fseeko` uses the correct type `off_t` for the *offset* parameter.

For this reason, it is a good idea to prefer `ftello` whenever it is available, since its functionality is (if different at all) closer the underlying definition.

The functionality and return value are the same as for `fseek`.

The function is an extension defined in the Unix Single Specification version 2.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, this function is in fact `fseeko64`—the LFS interface transparently replaces the old interface.

`int` **fseeko64** (`FILE *`*stream*`, off64_t` *offset*`, int` *whence*) Function

This function is similar to `fseeko` except that the *offset* parameter is of type `off64_t`. This also requires that the stream *stream* was opened using either `fopen64`, `freopen64` or `tmpfile64`, since otherwise the underlying file operations to position the file pointer beyond the 2^31 bytes limit might fail.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `fseeko` and so transparently replaces the old interface.

Portability Note: In non-POSIX systems, `ftell`, `ftello`, `fseek` and `fseeko` might work reliably only on binary streams (see Section 17.17 [Text and Binary Streams], page 499).

The following symbolic constants are defined for use as the *whence* argument to `fseek`. They are also used with the `lseek` function[22] and to specify offsets for file locks.[23]

`int` **SEEK_SET** Macro
This is an integer constant which, when used as the *whence* argument to the `fseek` or `fseeko` function, specifies that the offset provided is relative to the beginning of the file.

`int` **SEEK_CUR** Macro
This is an integer constant which, when used as the *whence* argument to the `fseek` or `fseeko` function, specifies that the offset provided is relative to the current file position.

`int` **SEEK_END** Macro
This is an integer constant which, when used as the *whence* argument to the `fseek` or `fseeko` function, specifies that the offset provided is relative to the end of the file.

`void` **rewind** (`FILE *`*stream*) Function
The `rewind` function positions the stream *stream* at the beginning of the file. It is equivalent to calling `fseek` or `fseeko` on the *stream* with an *offset* argument of `0L` and a *whence* argument of `SEEK_SET`, except that the return value is discarded and the error indicator for the stream is reset.

These three aliases for the '`SEEK_...`' constants exist for the sake of compatibility with older BSD systems. They are defined in two different header files: '`fcntl.h`' and '`sys/file.h`'.

`L_SET` An alias for `SEEK_SET`

`L_INCR` An alias for `SEEK_CUR`

`L_XTND` An alias for `SEEK_END`

17.19 Portable File-Position Functions

On the GNU system, the file position is truly a character count. You can specify any character count value as an argument to `fseek` or `fseeko` and get reliable results for any random-access file. However, some ISO C systems do not represent file positions in this way.

On some systems where text streams truly differ from binary streams, it is impossible to represent the file position of a text stream as a count of characters from the

[22] Ibid., "Input and Output Primitives".
[23] Ibid., "Control Operations on Files".

beginning of the file. For example, the file position on some systems must encode both a record offset within the file, and a character offset within the record.

As a consequence, if you want your programs to be portable to these systems, you must observe certain rules:

- The value returned from `ftell` on a text stream has no predictable relationship to the number of characters you have read so far. The only thing you can rely on is that you can use it subsequently as the *offset* argument to `fseek` or `fseeko` to move back to the same file position.
- In a call to `fseek` or `fseeko` on a text stream, either the *offset* must be zero, or *whence* must be `SEEK_SET` and the *offset* must be the result of an earlier call to `ftell` on the same stream.
- The value of the file-position indicator of a text stream is undefined while there are characters that have been pushed back with `ungetc` that haven't been read or discarded (see Section 17.10 [Unreading], page 458).

But even if you observe these rules, you may still have trouble with long files, because `ftell` and `fseek` use a `long int` value to represent the file position. This type may not have room to encode all the file positions in a large file. Using the `ftello` and `fseeko` functions might help here, since the `off_t` type is expected to be able to hold all file-position values but this still does not help to handle additional information that must be associated with a file position.

So if you do want to support systems with peculiar encodings for the file positions, it is better to use the functions `fgetpos` and `fsetpos` instead. These functions represent the file position using the data type `fpos_t`, whose internal representation varies from system to system.

These symbols are declared in the header file '`stdio.h`'.

fpos_t Data Type

This is the type of an object that can encode information about the file position of a stream, for use by the functions `fgetpos` and `fsetpos`.

In the GNU system, `fpos_t` is an opaque data structure that contains internal data to represent file-offset and conversion-state information. In other systems, it might have a different internal representation.

When compiling with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this type is in fact equivalent to `fpos64_t`, since the LFS interface transparently replaces the old interface.

fpos64_t Data Type

This is the type of an object that can encode information about the file position of a stream, for use by the functions `fgetpos64` and `fsetpos64`.

In the GNU system, `fpos64_t` is an opaque data structure that contains internal data to represent file-offset and conversion-state information. In other systems, it might have a different internal representation.

`int` **fgetpos** (`FILE *`*stream*`, fpos_t *`*position*) Function
This function stores the value of the file-position indicator for the stream *stream* in the `fpos_t` object pointed to by *position*. If successful, `fgetpos` returns zero. Otherwise, it returns a nonzero value and stores an implementation-defined positive value in `errno`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, the function is in fact `fgetpos64`—the LFS interface transparently replaces the old interface.

`int` **fgetpos64** (`FILE *`*stream*`, fpos64_t *`*position*) Function
This function is similar to `fgetpos`, but the file position is returned in a variable of type `fpos64_t` to which *position* points.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `fgetpos`, and so transparently replaces the old interface.

`int` **fsetpos** (`FILE *`*stream*`, const fpos_t *`*position*) Function
This function sets the file-position indicator for the stream *stream* to the position *position*, which must have been set by a previous call to `fgetpos` on the same stream. If successful, `fsetpos` clears the end-of-file indicator on the stream, discards any characters that were pushed back by the use of `ungetc`, and returns a value of zero. Otherwise, `fsetpos` returns a nonzero value and stores an implementation-defined positive value in `errno`.

When the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit system, the function is in fact `fsetpos64`—the LFS interface transparently replaces the old interface.

`int` **fsetpos64** (`FILE *`*stream*`, const fpos64_t *`*position*) Function
This function is similar to `fsetpos`, but the file position used for positioning is provided in a variable of type `fpos64_t` to which *position* points.

If the sources are compiled with `_FILE_OFFSET_BITS == 64` on a 32-bit machine, this function is available under the name `fsetpos`, and so transparently replaces the old interface.

17.20 Stream Buffering

Characters that are written to a stream are normally accumulated and transmitted asynchronously to the file in a block, instead of appearing as soon as they are output by the application program. Similarly, streams often retrieve input from the host environment in blocks rather than on a character-by-character basis. This is called *buffering*.

If you are writing programs that do interactive input and output using streams, you need to understand how buffering works when you design the user interface to your program. Otherwise, you might find that output (such as progress or prompt

messages) doesn't appear when you intended it to, or displays some other unexpected behavior.

This section deals only with controlling when characters are transmitted between the stream and the file or device, and *not* with how things like echoing, flow control and the like are handled on specific classes of devices.[24]

You can bypass the stream buffering facilities altogether by using the low-level input and output functions that operate on file descriptors instead.[25]

17.20.1 Buffering Concepts

There are three different kinds of buffering strategies:

- Characters written to or read from an *unbuffered* stream are transmitted individually to or from the file as soon as possible.
- Characters written to a *line buffered* stream are transmitted to the file in blocks when a newline character is encountered.
- Characters written to or read from a *fully buffered* stream are transmitted to or from the file in blocks of arbitrary size.

Newly opened streams are normally fully buffered, with one exception: a stream connected to an interactive device such as a terminal is initially line buffered (see Section 17.20.3 [Controlling Which Kind of Buffering], page 506, for information on how to select a different kind of buffering). Usually, the automatic selection gives you the most convenient kind of buffering for the file or device you open.

The use of line buffering for interactive devices implies that output messages ending in a newline will appear immediately—which is usually what you want. Output that doesn't end in a newline might or might not show up immediately, so if you want them to appear immediately, you should flush buffered output explicitly with `fflush`, as described in Section 17.20.2 [Flushing Buffers], page 505.

17.20.2 Flushing Buffers

Flushing output on a buffered stream means transmitting all accumulated characters to the file. There are many circumstances when buffered output on a stream is flushed automatically:

- When you try to do output and the output buffer is full
- When the stream is closed (see Section 17.4 [Closing Streams], page 444)
- When the program terminates by calling `exit` (see Section 14.6.1 [Normal Termination], page 425)
- When a newline is written, if the stream is line buffered
- Whenever an input operation on *any* stream actually reads data from its file

[24] For information on common control operations on terminal devices, see Loosemore et al., "Low-Level Terminal Interface".

[25] Ibid., "Low-Level Input/Output".

If you want to flush the buffered output at another time, call `fflush`, which is declared in the header file '`stdio.h`'.

`int` **fflush** (`FILE *`*stream*) Function

This function causes any buffered output on *stream* to be delivered to the file. If *stream* is a null pointer, then `fflush` causes buffered output on *all* open output streams to be flushed.

This function returns `EOF` if a write error occurs, or zero otherwise.

`int` **fflush_unlocked** (`FILE *`*stream*) Function

The `fflush_unlocked` function is equivalent to the `fflush` function, except that it does not implicitly lock the stream.

The `fflush` function can be used to flush all streams currently opened. While this is useful in some situations, it often does more than necessary. It could be used when terminal input is required and the programmer wants to be sure that all output is visible on the terminal. But this means that only line-buffered streams have to be flushed. Solaris introduced a function especially for this. It was always available in the GNU C Library in some form but never officially exported.

`void` **_flushlbf** (`void`) Function

The `_flushlbf` function flushes all line buffered streams currently opened.

This function is declared in the '`stdio_ext.h`' header.

Compatibility Note: Some operating systems have been known to be so thoroughly fixated on line-oriented input and output that flushing a line buffered stream causes a newline to be written! Fortunately, this "feature" seems to be becoming less common. You do not need to worry about this in the GNU system.

In some situations, it might be useful to not flush the output pending for a stream but instead simply forget it. If transmission is costly and the output is not needed anymore, this is valid reasoning. In this situation, a nonstandard function introduced in Solaris and available in the GNU C Library can be used.

`void` **__fpurge** (`FILE *`*stream*) Function

The `__fpurge` function causes the buffer of the stream *stream* to be emptied. If the stream is currently in read mode, all input in the buffer is lost. If the stream is in output mode, the buffered output is not written to the device (or whatever other underlying storage), and the buffer is cleared.

This function is declared in '`stdio_ext.h`'.

17.20.3 Controlling Which Kind of Buffering

After opening a stream (but before any other operations have been performed on it), you can explicitly specify what kind of buffering you want it to have using the `setvbuf` function.

The facilities listed in this section are declared in the header file '`stdio.h`'.

`int` **setvbuf** (`FILE *`*stream*`, char *`*buf*`, int` *mode*`, size_t` *size*) Function

This function is used to specify that the stream *stream* should have the buffering mode *mode*, which can be either `_IOFBF` (for full buffering), `_IOLBF` (for line buffering), or `_IONBF` (for unbuffered input/output).

If you specify a null pointer as the *buf* argument, then `setvbuf` allocates a buffer itself using `malloc`. This buffer will be freed when you close the stream.

Otherwise, *buf* should be a character array that can hold at least *size* characters. You should not free the space for this array as long as the stream remains open and this array remains its buffer. You should usually either allocate it statically, or `malloc` (see Section 3.2.2 [Unconstrained Allocation], page 42) the buffer. Using an automatic array is not a good idea unless you close the file before exiting the block that declares the array.

While the array remains a stream buffer, the stream I/O functions will use the buffer for their internal purposes. You shouldn't try to access the values in the array directly while the stream is using it for buffering.

The `setvbuf` function returns zero on success, or a nonzero value if the value of *mode* is not valid, or if the request could not be honored.

`int` **_IOFBF** Macro

The value of this macro is an integer constant expression that can be used as the *mode* argument to the `setvbuf` function to specify that the stream should be fully buffered.

`int` **_IOLBF** Macro

The value of this macro is an integer constant expression that can be used as the *mode* argument to the `setvbuf` function to specify that the stream should be line buffered.

`int` **_IONBF** Macro

The value of this macro is an integer constant expression that can be used as the *mode* argument to the `setvbuf` function to specify that the stream should be unbuffered.

`int` **BUFSIZ** Macro

The value of this macro is an integer constant expression that is good to use for the *size* argument to `setvbuf`. This value is guaranteed to be at least `256`.

The value of `BUFSIZ` is chosen on each system so as to make stream I/O efficient. So it is a good idea to use `BUFSIZ` as the size for the buffer when you call `setvbuf`.

Actually, you can get an even better value to use for the buffer size by means of the `fstat` system call—it is found in the `st_blksize` field of the file attributes.[26]

[26] Ibid., "Attribute Meanings".

Sometimes people also use `BUFSIZ` as the allocation size of buffers used for related purposes, such as strings used to receive a line of input with `fgets` (see Section 17.8 [Character Input], page 453). There is no particular reason to use `BUFSIZ` for this instead of any other integer, except that it might lead to doing I/O in chunks of an efficient size.

`void` **setbuf** (`FILE *`*stream*`, char *`*buf*) Function

If *buf* is a null pointer, the effect of this function is equivalent to calling `setvbuf` with a *mode* argument of `_IONBF`. Otherwise, it is equivalent to calling `setvbuf` with *buf*, and a *mode* of `_IOFBF` and a *size* argument of `BUFSIZ`.

The `setbuf` function is provided for compatibility with old code. Use `setvbuf` in all new programs.

`void` **setbuffer** (`FILE *`*stream*`, char *`*buf*`, size_t` *size*) Function

If *buf* is a null pointer, this function makes *stream* unbuffered. Otherwise, it makes *stream* fully buffered using *buf* as the buffer. The *size* argument specifies the length of *buf*.

This function is provided for compatibility with old BSD code. Use `setvbuf` instead.

`void` **setlinebuf** (`FILE *`*stream*) Function

This function makes *stream* be line buffered, and allocates the buffer for you.

This function is provided for compatibility with old BSD code. Use `setvbuf` instead.

It is possible to query whether a given stream is line buffered or not using a nonstandard function introduced in Solaris and available in the GNU C Library.

`int` **__flbf** (`FILE *`*stream*) Function

The `__flbf` function will return a nonzero value in case the stream *stream* is line buffered. Otherwise, the return value is zero.

This function is declared in the '`stdio_ext.h`' header.

Two more extensions allow you to determine the size of the buffer and how much of it is used. These functions were also introduced in Solaris.

`size_t` **__fbufsize** (`FILE *`*stream*) Function

The `__fbufsize` function returns the size of the buffer in the stream *stream*. This value can be used to optimize the use of the stream.

This function is declared in the '`stdio_ext.h`' header.

`size_t` **__fpending** (`FILE *`*stream*) Function

The `__fpending` function returns the number of bytes currently in the output buffer. For a wide-oriented stream, the measuring unit is wide characters. This function should not be used on buffers in read mode or opened read-only.

This function is declared in the '`stdio_ext.h`' header.

17.21 Other Kinds of Streams

The GNU library provides ways for you to define additional kinds of streams that do not necessarily correspond to an open file.

One such type of stream takes input from or writes output to a string. These kinds of streams are used internally to implement the `sprintf` and `sscanf` functions. You can also create such a stream explicitly, using the functions described in Section 17.21.1 [String Streams], page 509.

More generally, you can define streams that do input/output to arbitrary objects using functions supplied by your program. This protocol is discussed in Section 17.21.3 [Programming Your Own Custom Streams], page 512.

Portability Note: The facilities described in this section are specific to GNU. Other systems or C implementations might or might not provide equivalent functionality.

17.21.1 String Streams

The `fmemopen` and `open_memstream` functions allow you to do I/O to a string or memory buffer. These facilities are declared in '`stdio.h`'.

`FILE *` **fmemopen** (`void *`*buf*`, size_t` *size*`, const char *`*opentype*) Function

This function opens a stream that allows the access specified by the *opentype* argument, that reads from or writes to the buffer specified by the argument *buf*. This array must be at least *size* bytes long.

If you specify a null pointer as the *buf* argument, `fmemopen` dynamically allocates an array *size* bytes long (as with `malloc`; see Section 3.2.2 [Unconstrained Allocation], page 42). This is really only useful if you are going to write things to the buffer and then read them back in again, because you have no way of actually getting a pointer to the buffer (for this, try `open_memstream`, below). The buffer is freed when the stream is closed.

The argument *opentype* is the same as in `fopen` (see Section 17.3 [Opening Streams], page 440). If the *opentype* specifies append mode, then the initial file position is set to the first null character in the buffer. Otherwise, the initial file position is at the beginning of the buffer.

When a stream open for writing is flushed or closed, a null character (zero byte) is written at the end of the buffer if it fits. You should add an extra byte to the *size* argument to account for this. Attempts to write more than *size* bytes to the buffer result in an error.

For a stream open for reading, null characters (0 bytes) in the buffer do not count as end of file. Read operations indicate end of file only when the file position advances past *size* bytes. So, if you want to read characters from a null-terminated string, you should supply the length of the string as the *size* argument.

Here is an example of using `fmemopen` to create a stream for reading from a string:

```
#include <stdio.h>

static char buffer[] = "foobar";

int
main (void)
{
  int ch;
  FILE *stream;

  stream = fmemopen (buffer, strlen (buffer), "r");
  while ((ch = fgetc (stream)) != EOF)
    printf ("Got %c\n", ch);
  fclose (stream);

  return 0;
}
```

This program produces the following output:

```
Got f
Got o
Got o
Got b
Got a
Got r
```

`FILE *` **open_memstream** (`char **`*ptr*`, size_t *`*sizeloc*) Function

This function opens a stream for writing to a buffer. The buffer is allocated dynamically (as with `malloc`; see Section 3.2.2 [Unconstrained Allocation], page 42) and grown as necessary.

When the stream is closed with `fclose` or flushed with `fflush`, the locations *ptr* and *sizeloc* are updated to contain the pointer to the buffer and its size. The values thus stored remain valid only as long as no further output on the stream takes place. If you do more output, you must flush the stream again to store new values before you use them again.

A null character is written at the end of the buffer. This null character is *not* included in the size value stored at *sizeloc*.

You can move the stream's file position with `fseek` or `fseeko` (see Section 17.18 [File Positioning], page 500). Moving the file position past the end of the data already written fills the intervening space with zeros.

Here is an example of using `open_memstream`:

```
#include <stdio.h>

int
main (void)
{
  char *bp;
  size_t size;
  FILE *stream;

  stream = open_memstream (&bp, &size);
  fprintf (stream, "hello");
  fflush (stream);
  printf ("buf = `%s', size = %d\n", bp, size);
  fprintf (stream, ", world");
  fclose (stream);
  printf ("buf = `%s', size = %d\n", bp, size);

  return 0;
}
```

This program produces the following output:

```
buf = `hello', size = 5
buf = `hello, world', size = 12
```

17.21.2 Obstack Streams

You can open an output stream that puts it data in an obstack (see Section 3.2.4 [Obstacks], page 59).

`FILE *` **open_obstack_stream** (`struct obstack *`*obstack*) Function

This function opens a stream for writing data into the obstack *obstack*. This starts an object in the obstack and makes it grow as data is written (see Section 3.2.4.6 [Growing Objects], page 64).

Calling `fflush` on this stream updates the current size of the object to match the amount of data that has been written. After a call to `fflush`, you can examine the object temporarily.

You can move the file position of an obstack stream with `fseek` or `fseeko` (see Section 17.18 [File Positioning], page 500). Moving the file position past the end of the data written fills the intervening space with zeros.

To make the object permanent, update the obstack with `fflush`, and then use `obstack_finish` to finalize the object and get its address. The following write to the stream starts a new object in the obstack, and later writes add to that object until you do another `fflush` and `obstack_finish`.

But how do you find out how long the object is? You can get the length in bytes by calling `obstack_object_size` (see Section 3.2.4.8 [Status of an Obstack], page 67), or you can null-terminate the object like this:

```
obstack_1grow (obstack, 0);
```

Whichever one you do (you can do both if you wish), you must do it *before* calling `obstack_finish`.

Here is a sample function that uses `open_obstack_stream`:

```
char *
make_message_string (const char *a, int b)
{
  FILE *stream = open_obstack_stream (&message_obstack);
  output_task (stream);
  fprintf (stream, ": ");
  fprintf (stream, a, b);
  fprintf (stream, "\n");
  fclose (stream);
  obstack_1grow (&message_obstack, 0);
  return obstack_finish (&message_obstack);
}
```

17.21.3 Programming Your Own Custom Streams

This section describes how you can make a stream that gets input from an arbitrary data source or writes output to an arbitrary data sink programmed by you. We call these *custom streams*. The functions and types described here are all GNU extensions.

17.21.3.1 Custom Streams and Cookies

Inside every custom stream is a special object called the *cookie*. This is an object supplied by you that records where to fetch or store the data read or written. It is up to you to define a data type to use for the cookie. The stream functions in the library never refer directly to its contents, and they don't even know what the type is; they record its address with type `void *`.

To implement a custom stream, you must specify *how* to fetch or store the data in the specified place. You do this by defining *hook functions* to read, write, change file position, and close the stream. All four of these functions will be passed the stream's cookie so they can tell where to fetch or store the data. The library functions don't know what's inside the cookie, but your functions will know.

When you create a custom stream, you must specify the cookie pointer, and also the four hook functions stored in a structure of type `cookie_io_functions_t`.

These facilities are declared in '`stdio.h`'.

cookie_io_functions_t Data Type

This is a structure type that holds the functions that define the communications protocol between the stream and its cookie. It has the following members:

`cookie_read_function_t *read`
: This is the function that reads data from the cookie. If the value is a null pointer instead of a function, then read operations on this stream always return `EOF`.

`cookie_write_function_t *write`
: This is the function that writes data to the cookie. If the value is a null pointer instead of a function, then data written to the stream is discarded.

`cookie_seek_function_t *seek`
: This is the function that performs the equivalent of file positioning on the cookie. If the value is a null pointer instead of a function, calls to `fseek` or `fseeko` on this stream can only seek to locations within the buffer; any attempt to seek outside the buffer will return an `ESPIPE` error.

`cookie_close_function_t *close`
: This function performs any appropriate clean-up on the cookie when closing the stream. If the value is a null pointer instead of a function, nothing special is done to close the cookie when the stream is closed.

`FILE *` **fopencookie** (`void *`*cookie*`, const char *`*opentype*`, cookie_io_functions_t` *io-functions*) Function

This function actually creates the stream for communicating with the *cookie* using the functions in the *io-functions* argument. The *opentype* argument is interpreted as for `fopen` (see Section 17.3 [Opening Streams], page 440)—but note that the *truncate on open* option is ignored. The new stream is fully buffered.

The `fopencookie` function returns the newly created stream, or a null pointer in case of an error.

17.21.3.2 Custom Stream Hook Functions

Here are more details on how you should define the four hook functions that a custom stream needs.

You should define the function to read data from the cookie as:

```
ssize_t reader (void *cookie, char *buffer, size_t size)
```

This is very similar to the `read` function.[27] Your function should transfer up to *size* bytes into the *buffer*, and return the number of bytes read, or zero to indicate end of file. You can return a value of `-1` to indicate an error.

[27] Ibid., “I/O Primitives”.

You should define the function to write data to the cookie as:

```
ssize_t writer (void *cookie, const char *buffer, size_t size)
```

This is very similar to the `write` function.[28] Your function should transfer up to *size* bytes from the buffer, and return the number of bytes written. You can return a value of `-1` to indicate an error.

You should define the function to perform seek operations on the cookie as:

```
int seeker (void *cookie, fpos_t *position, int whence)
```

For this function, the *position* and *whence* arguments are interpreted as for `fgetpos` (see Section 17.19 [Portable File-Position Functions], page 502). In the GNU library, `fpos_t` is equivalent to `off_t` or `long int`, and simply represents the number of bytes from the beginning of the file.

After doing the seek operation, your function should store the resulting file position relative to the beginning of the file in *position*. Your function should return a value of `0` on success and `-1` to indicate an error.

You should define the function to do clean-up operations on the cookie appropriate for closing the stream as:

```
int cleaner (void *cookie)
```

Your function should return `-1` to indicate an error, and `0` otherwise.

cookie_read_function Data Type
This is the data type that the read function for a custom stream should have. If you declare the function as shown above, this is the type it will have.

cookie_write_function Data Type
This is the data type of the write function for a custom stream.

cookie_seek_function Data Type
This is the data type of the seek function for a custom stream.

cookie_close_function Data Type
This is the data type of the close function for a custom stream.

17.22 Formatted Messages

On systems that are based on System V, messages of programs (especially the system tools) are printed in a strict form using the `fmtmsg` function. The uniformity sometimes helps the user to interpret messages, and the strictness tests of the `fmtmsg` function ensure that the programmer follows some minimal requirements.

[28] Ibid., “I/O Primitives”.

17.22.1 Printing Formatted Messages

Messages can be printed to standard error and/or to the console. To select the destination, the programmer can use the following two values, bit-wise OR combined if wanted, for the *classification* parameter of `fmtmsg`:

`MM_PRINT`
: Display the message in standard error.

`MM_CONSOLE`
: Display the message on the system console.

The erroneous piece of the system can be signalled by exactly one of the following values that also is bit-wise ORed with the *classification* parameter to `fmtmsg`:

`MM_HARD` The source of the condition is some hardware.

`MM_SOFT` The source of the condition is some software.

`MM_FIRM` The source of the condition is some firmware.

A third component of the *classification* parameter to `fmtmsg` can describe the part of the system that detects the problem. This is done by using exactly one of the following values:

`MM_APPL` The erroneous condition is detected by the application.

`MM_UTIL` The erroneous condition is detected by a utility.

`MM_OPSYS`
: The erroneous condition is detected by the operating system.

A last component of *classification* can signal the results of this message. Exactly one of the following values can be used:

`MM_RECOVER`
: It is a recoverable error.

`MM_NRECOV`
: It is a nonrecoverable error.

`int` **fmtmsg** (`long int` *classification*, `const char *`*label*, `int` *severity*, `const char *`*text*, `const char *`*action*, `const char *`*tag*) Function

Display a message described by its parameters on the device(s) specified in the *classification* parameter. The *label* parameter identifies the source of the message. The string should consist of two colon-separated parts where the first part has not more than ten and the second part not more than fourteen characters. The *text* parameter describes the condition of the error, the *action* parameter describes possible steps to recover from the error, and the *tag* parameter is a

reference to the online documentation where more information can be found. It should contain the *label* value and a unique identification number.

Each of the parameters can be a special value that means this value is to be omitted. The symbolic names for these values are

`MM_NULLLBL`
: Ignore *label* parameter.

`MM_NULLSEV`
: Ignore *severity* parameter.

`MM_NULLMC`
: Ignore *classification* parameter. This implies that nothing is actually printed.

`MM_NULLTXT`
: Ignore *text* parameter.

`MM_NULLACT`
: Ignore *action* parameter.

`MM_NULLTAG`
: Ignore *tag* parameter.

There is another way certain fields can be omitted from the output to standard error. This is described below in the description of environment variables influencing the behavior.

The *severity* parameter can have one of the values in the following table:

`MM_NOSEV`
: Nothing is printed; this value is the same as `MM_NULLSEV`.

`MM_HALT` This value is printed as `HALT`.

`MM_ERROR`
: This value is printed as `ERROR`.

`MM_WARNING`
: This value is printed as `WARNING`.

`MM_INFO` This value is printed as `INFO`.

The numeric values of these five macros are between `0` and `4`. Using the environment variable `SEV_LEVEL`, or using the `addseverity` function, one can add more severity levels with their corresponding string to print. This is described below (see Section 17.22.2 [Adding Severity Classes], page 518).

If no parameter is ignored, the output looks like this:

label`:` *severity-string*`:` *text*
`TO FIX:` *action tag*

The colons, newline characters and the `TO FIX` string are inserted if necessary, i.e., if the corresponding parameter is not ignored.

This function is specified in the *X/Open Portability Guide*.[29] It is also available on all systems derived from System V.

The function returns the value `MM_OK` if no error occurred. If only the printing to standard error failed, it returns `MM_NOMSG`. If printing to the console fails, it returns `MM_NOCON`. If nothing is printed, `MM_NOTOK` is returned. Among situations where all outputs fail, this last value is also returned if a parameter value is incorrect.

There are two environment variables that influence the behavior of `fmtmsg`. The first is `MSGVERB`. It is used to control the output actually happening on standard error (*not* the console output). Each of the five fields can explicitly be enabled. To do this, the user has to put the `MSGVERB` variable with a format like the following in the environment before calling the `fmtmsg` function the first time:

`MSGVERB=`*keyword*`[:`*keyword*`[:...]]`

Valid *keyword*s are `label`, `severity`, `text`, `action` and `tag`. If the environment variable is not given or is an empty string, an unsupported keyword is given, or the value is somehow else invalid, no part of the message is masked out.

The second environment variable that influences the behavior of `fmtmsg` is `SEV_LEVEL`. This variable and the change in the behavior of `fmtmsg` are not specified in the *X/Open Portability Guide*.[30] It is available in System V systems, though. It can be used to introduce new severity levels. By default, only the five severity levels described above are available. Any other numeric value would make `fmtmsg` print nothing.

If the user puts `SEV_LEVEL` with a format like:

`SEV_LEVEL=[`*description*`[:`*description*`[:...]]]`

in the environment of the process before the first call to `fmtmsg`, where *description* has a value of the form:

severity-keyword`,` *level*`,` *printstring*

The *severity-keyword* part is not used by `fmtmsg`, but it has to be present. The *level* part is a string representation of a number. The numeric value must be a number greater than 4. This value must be used in the *severity* parameter of `fmtmsg` to select this class. It is not possible to overwrite any of the predefined classes. The *printstring* is the string printed when a message of this class is processed by `fmtmsg` (see above; `fmtsmg` does not print the numeric value but instead the string representation).

---

[29] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

[30] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

17.22.2 Adding Severity Classes

There is another way to introduce severity classes besides using the environment variable `SEV_LEVEL`. This simplifies the task of introducing new classes in a running program. One could use the `setenv` or `putenv` function to set the environment variable, but this is toilsome.

`int` **addseverity** (`int` *severity*, `const char *`*string*) Function

This function allows the introduction of new severity classes that can be addressed by the *severity* parameter of the `fmtmsg` function. The *severity* parameter of `addseverity` must match the value for the parameter with the same name of `fmtmsg`, and *string* is the string printed in the actual messages instead of the numeric value.

If *string* is `NULL`, the severity class with the numeric value according to *severity* is removed.

It is not possible to overwrite or remove one of the default severity classes. All calls to `addseverity` with *severity* set to one of the values for the default classes will fail.

The return value is `MM_OK` if the task was successfully performed. If the return value is `MM_NOTOK`, something went wrong. This could mean that no more memory is available, or a class is not available when it has to be removed.

This function is not specified in the *X/Open Portability Guide*[31], although the `fmtsmg` function is. It is available on System V systems.

17.22.3 How to Use `fmtmsg` and `addseverity`

Here is a simple example program to illustrate the use of both of the functions described in this section:

```
#include <fmtmsg.h>

int
main (void)
{
  addseverity (5, "NOTE2");
  fmtmsg (MM_PRINT, "only1field", MM_INFO, "text2", "action2", "tag2");
  fmtmsg (MM_PRINT, "UX:cat", 5, "invalid syntax", "refer to manual",
          "UX:cat:001");
  fmtmsg (MM_PRINT, "label:foo", 6, "text", "action", "tag");
  return 0;
}
```

[31] X/Open Company, *X/Open Portability Guide,* Issue 4, Version 2 (Reading, UK: X/Open Company, Ltd., 1994).

The second call to `fmtmsg` illustrates a use of this function as it usually occurs on System V systems, which heavily use this function. It seems worthwhile to give a short explanation here of how this system works on System V. The value of the *label* field (`UX:cat`) says that the error occurred in the Unix program `cat`. The explanation of the error follows and the value for the *action* parameter is *refer to manual*. One could be more specific here, if necessary. The *tag* field contains, as proposed above, the value of the string given for the *label* parameter, and additionally a unique ID (`001` in this case). For a GNU environment, this string could contain a reference to the corresponding node in the Info page for the program.

Running this program without specifying the `MSGVERB` and `SEV_LEVEL` function produces the following output:

```
UX:cat: NOTE2: invalid syntax
TO FIX: refer to manual UX:cat:001
```

We see the different fields of the message and how the extra glue (the colons and the `TO FIX` string) is printed. But only one of the three calls to `fmtmsg` produced output. The first call does not print anything because the *label* parameter is not in the correct form. The string must contain two fields, separated by a colon (see Section 17.22.1 [Printing Formatted Messages], page 515). The third `fmtmsg` call produced no output, since the class with the numeric value `6` is not defined. Although a class with numeric value `5` is also not defined by default, the call to `addseverity` introduces it, and the second call to `fmtmsg` produces the above output.

When we change the environment of the program to contain `SEV_LEVEL=XXX,6,NOTE`, we get a different result:

```
UX:cat: NOTE2: invalid syntax
TO FIX: refer to manual UX:cat:001
label:foo: NOTE: text
TO FIX: action tag
```

Now the third call to `fmtmsg` produced some output, and we see how the string `NOTE` from the environment variable appears in the message.

Now we can reduce the output by specifying which fields we are interested in. If we additionally set the environment variable `MSGVERB` to the value `severity:label:action` we get the following output:

```
UX:cat: NOTE2
TO FIX: refer to manual
label:foo: NOTE
TO FIX: action
```

The output produced by the *text* and the *tag* parameters to `fmtmsg` vanished. Please also note that now there is no colon after the `NOTE` and `NOTE2` strings in the output. This is not necessary, since there is no more output on this line because the text is missing.

Appendix A Summary of Library Facilities

This appendix is a complete list of the facilities declared within the header files supplied with the GNU C Library. Each entry also lists the standard or other source from which each facility is derived, and tells you where in the manual you can find more information about how to use it.

`long int a64l (const char *`*string*`)`
: 'stdlib.h' (XPG): Section 5.11 [Encode Binary Data], page 125.

`void abort (void)`
: 'stdlib.h' (ISO): Section 14.6.4 [Aborting a Program], page 427.

`int abs (int` *number*`)`
: 'stdlib.h' (ISO): Section 9.8.1 [Absolute Value], page 258.

`double acos (double` *x*`)`
: 'math.h' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`float acosf (float` *x*`)`
: 'math.h' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`double acosh (double` *x*`)`
: 'math.h' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`float acoshf (float` *x*`)`
: 'math.h' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double acoshl (long double` *x*`)`
: 'math.h' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double acosl (long double` *x*`)`
: 'math.h' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`int adjtime (const struct timeval *`*delta*`, struct timeval *`*olddelta*`)`
: 'sys/time.h' (BSD): Section 10.4.2 [High-Resolution Calendar], page 283.

`int adjtimex (struct timex *`*timex*`)`
: 'sys/timex.h' (GNU): Section 10.4.2 [High-Resolution Calendar], page 283.

`unsigned int alarm (unsigned int` *seconds*`)`
: 'unistd.h' (POSIX.1): Section 10.5 [Setting an Alarm], page 310.

`void * alloca (size_t` *size*`);`
: 'stdlib.h' (GNU, BSD): Section 3.2.5 [Automatic Storage with Variable Size], page 71.

`error_t argp_err_exit_status`
: 'argp.h' (GNU): Section 14.3.2 [Argp Global Variables], page 390.

`void argp_error (const struct argp_state *`*state*`, const char *`*fmt*`, ...)`
: 'argp.h' (GNU): Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397.

`int ARGP_ERR_UNKNOWN`
: 'argp.h' (GNU): Section 14.3.5 [Argp Parser Functions], page 394.

`void argp_failure (const struct argp_state *`*state*`, int` *status*`, int` *errnum*`, const char *`*fmt*`, ...)`
: 'argp.h' (GNU): Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397.

`ARGP_KEY_SUCCESS`
: 'argp.h' (GNU): Section 14.3.5.1 [Special Keys for Argp Parser Functions], page 395.

`ARGP_LONG_ONLY`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`ARGP_NO_ARGS`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`ARGP_NO_ERRS`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`ARGP_NO_EXIT`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`ARGP_NO_HELP`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`error_t argp_parse (const struct argp *`*argp*`, int` *argc*`, char **`*argv*`, unsigned` *flags*`, int *`*arg_index*`, void *`*input*`)`
: 'argp.h' (GNU): Section 14.3 [Parsing Program Options with Argp], page 389.

`ARGP_PARSE_ARGV0`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`const char * argp_program_bug_address`
: 'argp.h' (GNU): Section 14.3.2 [Argp Global Variables], page 390.

`const char * argp_program_version`
: 'argp.h' (GNU): Section 14.3.2 [Argp Global Variables], page 390.

`argp_program_version_hook`
: 'argp.h' (GNU): Section 14.3.2 [Argp Global Variables], page 390.

`ARGP_SILENT`
: 'argp.h' (GNU): Section 14.3.7 [Flags for argp_parse], page 401.

`void argp_state_help (const struct argp_state *`*state*`, FILE *`*stream*`, unsigned` *flags*`)`
: 'argp.h' (GNU): Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397.

`void argp_usage (const struct argp_state *`*state*`)`
: 'argp.h' (GNU): Section 14.3.5.2 [Functions for Use in Argp Parsers], page 397.

`error_t argz_add (char **`*argz*`, size_t *`*argz_len*`, const char *`*str*`)`
: 'argz.h' (GNU): Section 5.12.1 [Argz Functions], page 127.

`error_t argz_add_sep (char **`*argz*`, size_t *`*argz_len*`, const char *`*str*`, int` *delim*`)`
: 'argz.h' (GNU): Section 5.12.1 [Argz Functions], page 127.

`error_t argz_append (char **`*argz*`, size_t *`*argz_len*`, const char *`*buf*`, size_t` *buf_len*`)`
: 'argz.h' (GNU): Section 5.12.1 [Argz Functions], page 127.

`size_t argz_count (const char *`*argz*`, size_t` *arg_len*`)`
: 'argz.h' (GNU): Section 5.12.1 [Argz Functions], page 127.

`error_t argz_create (char *const` *argv*`[], char **`*argz*`, size_t *`*argz_len*`)`
: 'argz.h' (GNU): Section 5.12.1 [Argz Functions], page 127.

`long double atan2l (long double y, long double x)`
'`math.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`float atanf (float x)`
'`math.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`double atanh (double x)`
'`math.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`float atanhf (float x)`
'`math.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double atanhl (long double x)`
'`math.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double atanl (long double x)`
'`math.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`int atexit (void (*function) (void))`
'`stdlib.h`' (ISO): Section 14.6.3 [Clean-Ups on Exit], page 426.

`double atof (const char *string)`
'`stdlib.h`' (ISO): Section 9.11.2 [Parsing of Floats], page 273.

`int atoi (const char *string)`
'`stdlib.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`long int atol (const char *string)`
'`stdlib.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`long long int atoll (const char *string)`
'`stdlib.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`int backtrace (void **buffer, int size)`
'`execinfo.h`' (GNU): Section 16.1 [Backtraces], page 435.

`char ** backtrace_symbols (void *const *buffer, int size)`
'`execinfo.h`' (GNU): Section 16.1 [Backtraces], page 435.

`void backtrace_symbols_fd (void *const *buffer, int size, int fd)`
'`execinfo.h`' (GNU): Section 16.1 [Backtraces], page 435.

`char * basename (char *path)`
'`libgen.h`' (XPG): Section 5.8 [Finding Tokens in a String], page 119.

`char * basename (const char *filename)`
'`string.h`' (GNU): Section 5.8 [Finding Tokens in a String], page 119.

`int bcmp (const void *a1, const void *a2, size_t size)`
'`string.h`' (BSD): Section 5.5 [String/Array Comparison], page 105.

`void bcopy (const void *from, void *to, size_t size)`
'`string.h`' (BSD): Section 5.4 [Copying and Concatenation], page 93.

`char * bindtextdomain (const char *domainname, const char *dirname)`
'`libintl.h`' (GNU): Section 11.2.1.2 [How to Determine Which Catalog to Use], page 328.

`char * bind_textdomain_codeset (const char *domainname, const char *codeset)`
'`libintl.h`' (GNU): Section 11.2.1.4 [How to Specify the Output Character Set That `gettext` Uses], page 335.

`complex double casin (complex double z)`
'`complex.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`complex float casinf (complex float z)`
'`complex.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`complex double casinh (complex double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex float casinhf (complex float z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double casinhl (complex long double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double casinl (complex long double z)`
'`complex.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`complex double catan (complex double z)`
'`complex.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`complex float catanf (complex float z)`
'`complex.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`complex double catanh (complex double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex float catanhf (complex float z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double catanhl (complex long double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double catanl (complex long double z)`
'`complex.h`' (ISO): Section 8.3 [Inverse Trigonometric Functions], page 206.

`nl_catd catopen (const char *cat_name, int flag)`
'`nl_types.h`' (X/Open): Section 11.1.1 [The `catgets` Function Family], page 316.

`double cbrt (double x)`
'`math.h`' (BSD): Section 8.4 [Exponentiation and Logarithms], page 207.

`float cbrtf (float x)`
'`math.h`' (BSD): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double cbrtl (long double x)`
'`math.h`' (BSD): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex double ccos (complex double z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex float ccosf (complex float z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex double ccosh (complex double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex float ccoshf (complex float z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex double clog (complex double z)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex double clog10 (complex double z)`
'`complex.h`' (GNU): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex float clog10f (complex float z)`
'`complex.h`' (GNU): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex long double clog10l (complex long double z)`
'`complex.h`' (GNU): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex float clogf (complex float z)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex long double clogl (complex long double z)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex double conj (complex double z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`complex float conjf (complex float z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`complex long double conjl (complex long double z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`cookie_close_function`
'`stdio.h`' (GNU): Section 17.21.3.2 [Custom Stream Hook Functions], page 513.

`cookie_io_functions_t`
'`stdio.h`' (GNU): Section 17.21.3.1 [Custom Streams and Cookies], page 512.

`cookie_read_function`
'`stdio.h`' (GNU): Section 17.21.3.2 [Custom Stream Hook Functions], page 513.

`cookie_seek_function`
'`stdio.h`' (GNU): Section 17.21.3.2 [Custom Stream Hook Functions], page 513.

`cookie_write_function`
'`stdio.h`' (GNU): Section 17.21.3.2 [Custom Stream Hook Functions], page 513.

`double copysign (double x, double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`float copysignf (float x, float y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`long double copysignl (long double x, long double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`double cos (double x)`
'`math.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`float cosf (float x)`
'`math.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`double cosh (double x)`
'`math.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`float coshf (float x)`
'`math.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double coshl (long double x)`
'`math.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double cosl (long double x)`
'`math.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex double cpow (complex double base, complex double power)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex float cpowf (complex float base, complex float power)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex long double cpowl (complex long double base, complex long double power)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex double cproj (complex double z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`complex float cprojf (complex float z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`complex long double cprojl (complex long double z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`double creal (complex double z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`float crealf (complex float z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`long double creall (complex long double z)`
'`complex.h`' (ISO): Section 9.10 [Projections, Conjugates and Decomposing of Complex Numbers], page 267.

`complex double csin (complex double z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex float csinf (complex float z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex double csinh (complex double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex float csinhf (complex float z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double csinhl (complex long double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double csinl (complex long double z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex double csqrt (complex double z)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex float csqrtf (complex float z)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex long double csqrtl (complex long double z)`
'`complex.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`complex double ctan (complex double z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex float ctanf (complex float z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`complex double ctanh (complex double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex float ctanhf (complex float z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double ctanhl (complex long double z)`
'`complex.h`' (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`complex long double ctanl (complex long double z)`
'`complex.h`' (ISO): Section 8.2 [Trigonometric Functions], page 204.

`char * ctime (const time_t *time)`
'`time.h`' (ISO): Section 10.4.5 [Formatting Calendar Time], page 291.

`char * ctime_r (const time_t *time, char *buffer)`
'`time.h`' (POSIX.1c): Section 10.4.5 [Formatting Calendar Time], page 291.

`int daylight`
'`time.h`' (SVID): Section 10.4.8 [Functions and Variables for Time Zones], page 308.

`char * dcgettext (const char *domainname, const char *msgid, int category)`
'`libintl.h`' (GNU): Section 11.2.1.1 [What Has to Be Done to Translate a Message?], page 326.

`char * dcngettext (const char *domain, const char *msgid1, const char *msgid2, unsigned long int n, int category)`
'`libintl.h`' (GNU): Section 11.2.1.3 [Additional Functions for More Complicated Situations], page 330.

`char * dgettext (const char *domainname, const char *msgid)`
'`libintl.h`' (GNU): Section 11.2.1.1 [What Has to Be Done to Translate a Message?], page 326.

`double difftime (time_t time1, time_t time0)`
'`time.h`' (ISO): Section 10.2 [Elapsed Time], page 277.

`char * dirname (char *path)`
'`libgen.h`' (XPG): Section 5.8 [Finding Tokens in a String], page 119.

`int EBADE`
: ‘`errno.h`’ (Undocumented: Invalid exchange): Section 2.2 [Error Codes], page 18.

`int EBADF`
: ‘`errno.h`’ (POSIX.1: Bad file descriptor): Section 2.2 [Error Codes], page 18.

`int EBADFD`
: ‘`errno.h`’ (Undocumented: File descriptor in bad state): Section 2.2 [Error Codes], page 18.

`int EBADMSG`
: ‘`errno.h`’ (XOPEN: Bad message): Section 2.2 [Error Codes], page 18.

`int EBADR`
: ‘`errno.h`’ (Undocumented: Invalid request descriptor): Section 2.2 [Error Codes], page 18.

`int EBADRPC`
: ‘`errno.h`’ (BSD: RPC struct is bad): Section 2.2 [Error Codes], page 18.

`int EBADRQC`
: ‘`errno.h`’ (Undocumented: Invalid request code): Section 2.2 [Error Codes], page 18.

`int EBADSLT`
: ‘`errno.h`’ (Undocumented: Invalid slot): Section 2.2 [Error Codes], page 18.

`int EBFONT`
: ‘`errno.h`’ (Undocumented: Bad font file format): Section 2.2 [Error Codes], page 18.

`int EBUSY`
: ‘`errno.h`’ (POSIX.1: Device or resource busy): Section 2.2 [Error Codes], page 18.

`int ECANCELED`
: ‘`errno.h`’ (POSIX.1: Operation canceled): Section 2.2 [Error Codes], page 18.

`int ECHILD`
: ‘`errno.h`’ (POSIX.1: No child processes): Section 2.2 [Error Codes], page 18.

`int ECHRNG`
: ‘`errno.h`’ (: Channel number out of range): Section 2.2 [Error Codes], page 18.

`int ECOMM`
: ‘`errno.h`’ (Undocumented: Communication error on send): Section 2.2 [Error Codes], page 18.

`int ECONNABORTED`
: ‘`errno.h`’ (BSD: Software caused connection abort): Section 2.2 [Error Codes], page 18.

`int ECONNREFUSED`
: ‘`errno.h`’ (BSD: Connection refused): Section 2.2 [Error Codes], page 18.

`int ECONNRESET`
: ‘`errno.h`’ (BSD: Connection reset by peer): Section 2.2 [Error Codes], page 18.

`char * ecvt (double value, int ndigit, int *decpt, int *neg)`
'`stdlib.h`' (SVID, Unix98): Section 9.12 [Old-fashioned System V Number-to-String Functions], page 275.

`char * ecvt_r (double value, int ndigit, int *decpt, int *neg, char *buf, size_t len)`
'`stdlib.h`' (GNU): Section 9.12 [Old-fashioned System V Number-to-String Functions], page 275.

`int ED`
'`errno.h`' (GNU: Undocumented): Section 2.2 [Error Codes], page 18.

`int EDEADLK`
'`errno.h`' (POSIX.1: Resource deadlock avoided): Section 2.2 [Error Codes], page 18.

`int EDEADLOCK`
'`errno.h`' (Undocumented: File-locking deadlock error): Section 2.2 [Error Codes], page 18.

`int EDESTADDRREQ`
'`errno.h`' (BSD: Destination address required): Section 2.2 [Error Codes], page 18.

`int EDIED`
'`errno.h`' (GNU: Translator died): Section 2.2 [Error Codes], page 18.

`int EDOM`
'`errno.h`' (ISO: Numerical argument out of domain): Section 2.2 [Error Codes], page 18.

`int EDOTDOT`
'`errno.h`' (Undocumented: RFS specific error): Section 2.2 [Error Codes], page 18.

`int EDQUOT`
'`errno.h`' (BSD: Disk quota exceeded): Section 2.2 [Error Codes], page 18.

`int EEXIST`
'`errno.h`' (POSIX.1: File exists): Section 2.2 [Error Codes], page 18.

`int EFAULT`
'`errno.h`' (POSIX.1: Bad address): Section 2.2 [Error Codes], page 18.

`int EFBIG`
'`errno.h`' (POSIX.1: File too large): Section 2.2 [Error Codes], page 18.

`int EFTYPE`
'`errno.h`' (BSD: Inappropriate file type or format): Section 2.2 [Error Codes], page 18.

`int EGRATUITOUS`
'`errno.h`' (GNU: Gratuitous error): Section 2.2 [Error Codes], page 18.

`int EGREGIOUS`
'`errno.h`' (GNU: You really blew it this time): Section 2.2 [Error Codes], page 18.

`int EHOSTDOWN`
'`errno.h`' (BSD: Host is down): Section 2.2 [Error Codes], page 18.

`int EHOSTUNREACH`
: ‘`errno.h`’ (BSD: No route to host): Section 2.2 [Error Codes], page 18.

`int EIDRM`
: ‘`errno.h`’ (XOPEN: Identifier removed): Section 2.2 [Error Codes], page 18.

`int EIEIO`
: ‘`errno.h`’ (GNU: Computer bought the farm): Section 2.2 [Error Codes], page 18.

`int EILSEQ`
: ‘`errno.h`’ (ISO: Invalid or incomplete multibyte or wide character): Section 2.2 [Error Codes], page 18.

`int EINPROGRESS`
: ‘`errno.h`’ (BSD: Operation now in progress): Section 2.2 [Error Codes], page 18.

`int EINTR`
: ‘`errno.h`’ (POSIX.1: Interrupted system call): Section 2.2 [Error Codes], page 18.

`int EINVAL`
: ‘`errno.h`’ (POSIX.1: Invalid argument): Section 2.2 [Error Codes], page 18.

`int EIO`
: ‘`errno.h`’ (POSIX.1: Input/output error): Section 2.2 [Error Codes], page 18.

`int EISCONN`
: ‘`errno.h`’ (BSD: Transport endpoint is already connected): Section 2.2 [Error Codes], page 18.

`int EISDIR`
: ‘`errno.h`’ (POSIX.1: Is a directory): Section 2.2 [Error Codes], page 18.

`int EISNAM`
: ‘`errno.h`’ (Undocumented: Is a named type file): Section 2.2 [Error Codes], page 18.

`int EL2HLT`
: ‘`errno.h`’ (Obsolete: Level 2 halted): Section 2.2 [Error Codes], page 18.

`int EL2NSYNC`
: ‘`errno.h`’ (Obsolete: Level 2 not synchronized): Section 2.2 [Error Codes], page 18.

`int EL3HLT`
: ‘`errno.h`’ (Obsolete: Level 3 halted): Section 2.2 [Error Codes], page 18.

`int EL3RST`
: ‘`errno.h`’ (Obsolete: Level 3 reset): Section 2.2 [Error Codes], page 18.

`int ELIBACC`
: ‘`errno.h`’ (Undocumented: Can not access a needed shared library): Section 2.2 [Error Codes], page 18.

`int ELIBBAD`
: ‘`errno.h`’ (Undocumented: Accessing a corrupted shared library): Section 2.2 [Error Codes], page 18.

`int ELIBEXEC`
: ‘`errno.h`’ (Undocumented: Cannot exec a shared library directly): Section 2.2 [Error Codes], page 18.

`int ELIBMAX`
: ‘`errno.h`’ (Undocumented: Attempting to link in too many shared libraries): Section 2.2 [Error Codes], page 18.

`int ELIBSCN`
: ‘`errno.h`’ (Undocumented: .lib section in a.out corrupted): Section 2.2 [Error Codes], page 18.

`int ELNRNG`
: ‘`errno.h`’ (Undocumented: Link number out of range): Section 2.2 [Error Codes], page 18.

`int ELOOP`
: ‘`errno.h`’ (BSD: Too many levels of symbolic links): Section 2.2 [Error Codes], page 18.

`int EMEDIUMTYPE`
: ‘`errno.h`’ (Undocumented: Wrong medium type): Section 2.2 [Error Codes], page 18.

`int EMFILE`
: ‘`errno.h`’ (POSIX.1: Too many open files): Section 2.2 [Error Codes], page 18.

`int EMLINK`
: ‘`errno.h`’ (POSIX.1: Too many links): Section 2.2 [Error Codes], page 18.

`int EMSGSIZE`
: ‘`errno.h`’ (BSD: Message too long): Section 2.2 [Error Codes], page 18.

`int EMULTIHOP`
: ‘`errno.h`’ (XOPEN: Multihop attempted): Section 2.2 [Error Codes], page 18.

`int ENAMETOOLONG`
: ‘`errno.h`’ (POSIX.1: File name too long): Section 2.2 [Error Codes], page 18.

`int ENAVAIL`
: ‘`errno.h`’ (Undocumented: No XENIX semaphores available): Section 2.2 [Error Codes], page 18.

`int ENEEDAUTH`
: ‘`errno.h`’ (BSD: Need authenticator): Section 2.2 [Error Codes], page 18.

`int ENETDOWN`
: ‘`errno.h`’ (BSD: Network is down): Section 2.2 [Error Codes], page 18.

`int ENETRESET`
: ‘`errno.h`’ (BSD: Network dropped connection on reset): Section 2.2 [Error Codes], page 18.

`int ENETUNREACH`
: ‘`errno.h`’ (BSD: Network is unreachable): Section 2.2 [Error Codes], page 18.

`int ENFILE`
: ‘`errno.h`’ (POSIX.1: Too many open files in system): Section 2.2 [Error Codes], page 18.

`int ENOANO`
: ‘`errno.h`’ (Undocumented: No anode): Section 2.2 [Error Codes], page 18.

`int ENOBUFS`
: ‘`errno.h`’ (BSD: No buffer space available): Section 2.2 [Error Codes], page 18.

`int ENOCSI`
: ‘`errno.h`’ (Undocumented: No CSI structure available): Section 2.2 [Error Codes], page 18.

`int ENODATA`
: ‘`errno.h`’ (XOPEN: No data available): Section 2.2 [Error Codes], page 18.

`int ENODEV`
: ‘`errno.h`’ (POSIX.1: No such device): Section 2.2 [Error Codes], page 18.

`int ENOENT`
: ‘`errno.h`’ (POSIX.1: No such file or directory): Section 2.2 [Error Codes], page 18.

`int ENOEXEC`
: ‘`errno.h`’ (POSIX.1: Exec format error): Section 2.2 [Error Codes], page 18.

`int ENOLCK`
: ‘`errno.h`’ (POSIX.1: No locks available): Section 2.2 [Error Codes], page 18.

`int ENOLINK`
: ‘`errno.h`’ (XOPEN: Link has been severed): Section 2.2 [Error Codes], page 18.

`int ENOMEDIUM`
: ‘`errno.h`’ (Undocumented: No medium found): Section 2.2 [Error Codes], page 18.

`int ENOMEM`
: ‘`errno.h`’ (POSIX.1: Cannot allocate memory): Section 2.2 [Error Codes], page 18.

`int ENOMSG`
: ‘`errno.h`’ (XOPEN: No message of desired type): Section 2.2 [Error Codes], page 18.

`int ENONET`
: ‘`errno.h`’ (Undocumented: Machine is not on the network): Section 2.2 [Error Codes], page 18.

`int ENOPKG`
: ‘`errno.h`’ (Undocumented: Package not installed): Section 2.2 [Error Codes], page 18.

`int ENOPROTOOPT`
: ‘`errno.h`’ (BSD: Protocol not available): Section 2.2 [Error Codes], page 18.

`int ENOSPC`
: ‘`errno.h`’ (POSIX.1: No space left on device): Section 2.2 [Error Codes], page 18.

`int ENOSR`
: ‘`errno.h`’ (XOPEN: Out of streams resources): Section 2.2 [Error Codes], page 18.

`int ENOSTR`
: ‘`errno.h`’ (XOPEN: Device not a stream): Section 2.2 [Error Codes], page 18.

`int ENOSYS`
: ‘`errno.h`’ (POSIX.1: Function not implemented): Section 2.2 [Error Codes], page 18.

`int ENOTBLK`
'`errno.h`' (BSD: Block device required): Section 2.2 [Error Codes], page 18.

`int ENOTCONN`
'`errno.h`' (BSD: Transport endpoint is not connected): Section 2.2 [Error Codes], page 18.

`int ENOTDIR`
'`errno.h`' (POSIX.1: Not a directory): Section 2.2 [Error Codes], page 18.

`int ENOTEMPTY`
'`errno.h`' (POSIX.1: Directory not empty): Section 2.2 [Error Codes], page 18.

`int ENOTNAM`
'`errno.h`' (Undocumented: Not a XENIX named type file): Section 2.2 [Error Codes], page 18.

`int ENOTSOCK`
'`errno.h`' (BSD: Socket operation on nonsocket): Section 2.2 [Error Codes], page 18.

`int ENOTSUP`
'`errno.h`' (POSIX.1: Not supported): Section 2.2 [Error Codes], page 18.

`int ENOTTY`
'`errno.h`' (POSIX.1: Inappropriate ioctl for device): Section 2.2 [Error Codes], page 18.

`int ENOTUNIQ`
'`errno.h`' (Undocumented: Name not unique on network): Section 2.2 [Error Codes], page 18.

`char ** environ`
'`unistd.h`' (POSIX.1): Section 14.4.1 [Environment Access], page 419.

`error_t envz_add (char **`*envz*`, size_t *`*envz_len*`, const char *`*name*`, const char *`*value*`)`
'`envz.h`' (GNU): Section 5.12.2 [Envz Functions], page 130.

`char * envz_entry (const char *`*envz*`, size_t` *envz_len*`, const char *`*name*`)`
'`envz.h`' (GNU): Section 5.12.2 [Envz Functions], page 130.

`char * envz_get (const char *`*envz*`, size_t` *envz_len*`, const char *`*name*`)`
'`envz.h`' (GNU): Section 5.12.2 [Envz Functions], page 130.

`error_t envz_merge (char **`*envz*`, size_t *`*envz_len*`, const char *`*envz2*`, size_t` *envz2_len*`, int` *override*`)`
'`envz.h`' (GNU): Section 5.12.2 [Envz Functions], page 130.

`void envz_strip (char **`*envz*`, size_t *`*envz_len*`)`
'`envz.h`' (GNU): Section 5.12.2 [Envz Functions], page 130.

`int ENXIO`
'`errno.h`' (POSIX.1: No such device or address): Section 2.2 [Error Codes], page 18.

`int EOF`
'`stdio.h`' (ISO): Section 17.15 [End-of-File and Errors], page 497.

`int EOPNOTSUPP`
: ‘`errno.h`’ (BSD: Operation not supported): Section 2.2 [Error Codes], page 18.

`int EOVERFLOW`
: ‘`errno.h`’ (XOPEN: Value too large for defined data type): Section 2.2 [Error Codes], page 18.

`int EPERM`
: ‘`errno.h`’ (POSIX.1: Operation not permitted): Section 2.2 [Error Codes], page 18.

`int EPFNOSUPPORT`
: ‘`errno.h`’ (BSD: Protocol family not supported): Section 2.2 [Error Codes], page 18.

`int EPIPE`
: ‘`errno.h`’ (POSIX.1: Broken pipe): Section 2.2 [Error Codes], page 18.

`int EPROCLIM`
: ‘`errno.h`’ (BSD: Too many processes): Section 2.2 [Error Codes], page 18.

`int EPROCUNAVAIL`
: ‘`errno.h`’ (BSD: RPC bad procedure for program): Section 2.2 [Error Codes], page 18.

`int EPROGMISMATCH`
: ‘`errno.h`’ (BSD: RPC program version wrong): Section 2.2 [Error Codes], page 18.

`int EPROGUNAVAIL`
: ‘`errno.h`’ (BSD: RPC program not available): Section 2.2 [Error Codes], page 18.

`int EPROTO`
: ‘`errno.h`’ (XOPEN: Protocol error): Section 2.2 [Error Codes], page 18.

`int EPROTONOSUPPORT`
: ‘`errno.h`’ (BSD: Protocol not supported): Section 2.2 [Error Codes], page 18.

`int EPROTOTYPE`
: ‘`errno.h`’ (BSD: Protocol wrong type for socket): Section 2.2 [Error Codes], page 18.

`double erand48 (unsigned short int` *xsubi*`[3])`
: ‘`stdlib.h`’ (SVID): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int erand48_r (unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, double *`*result*`)`
: ‘`stdlib.h`’ (GNU): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int ERANGE`
: ‘`errno.h`’ (ISO: Numerical result out of range): Section 2.2 [Error Codes], page 18.

`int EREMCHG`
: ‘`errno.h`’ (Undocumented: Remote address changed): Section 2.2 [Error Codes], page 18.

`int EREMOTE`
: ‘`errno.h`’ (BSD: Object is remote): Section 2.2 [Error Codes], page 18.

`int ESOCKTNOSUPPORT`
: ‘`errno.h`’ (BSD: Socket type not supported): Section 2.2 [Error Codes], page 18.

`int ESPIPE`
: ‘`errno.h`’ (POSIX.1: Illegal seek): Section 2.2 [Error Codes], page 18.

`int ESRCH`
: ‘`errno.h`’ (POSIX.1: No such process): Section 2.2 [Error Codes], page 18.

`int ESRMNT`
: ‘`errno.h`’ (Undocumented: Srmount error): Section 2.2 [Error Codes], page 18.

`int ESTALE`
: ‘`errno.h`’ (BSD: Stale NFS file handle): Section 2.2 [Error Codes], page 18.

`int ESTRPIPE`
: ‘`errno.h`’ (Undocumented: Streams pipe error): Section 2.2 [Error Codes], page 18.

`int ETIME`
: ‘`errno.h`’ (XOPEN: Timer expired): Section 2.2 [Error Codes], page 18.

`int ETIMEDOUT`
: ‘`errno.h`’ (BSD: Connection timed out): Section 2.2 [Error Codes], page 18.

`int ETOOMANYREFS`
: ‘`errno.h`’ (BSD: Too many references: cannot splice): Section 2.2 [Error Codes], page 18.

`int ETXTBSY`
: ‘`errno.h`’ (BSD: Text file busy): Section 2.2 [Error Codes], page 18.

`int EUCLEAN`
: ‘`errno.h`’ (Undocumented: Structure needs cleaning): Section 2.2 [Error Codes], page 18.

`int EUNATCH`
: ‘`errno.h`’ (Undocumented: Protocol driver not attached): Section 2.2 [Error Codes], page 18.

`int EUSERS`
: ‘`errno.h`’ (BSD: Too many users): Section 2.2 [Error Codes], page 18.

`int EWOULDBLOCK`
: ‘`errno.h`’ (BSD: Operation would block): Section 2.2 [Error Codes], page 18.

`int EXDEV`
: ‘`errno.h`’ (POSIX.1: Invalid cross-device link): Section 2.2 [Error Codes], page 18.

`int EXFULL`
: ‘`errno.h`’ (Undocumented: Exchange full): Section 2.2 [Error Codes], page 18.

`void exit (int` *status*`)`
: ‘`stdlib.h`’ (ISO): Section 14.6.1 [Normal Termination], page 425.

`void _Exit (int` *status*`)`
: ‘`stdlib.h`’ (ISO): Section 14.6.5 [Termination Internals], page 428.

`wint_t fgetwc (FILE *stream)`
'`wchar.h`' (ISO): Section 17.8 [Character Input], page 453.

`wint_t fgetwc_unlocked (FILE *stream)`
'`wchar.h`' (GNU): Section 17.8 [Character Input], page 453.

`wchar_t * fgetws (wchar_t *ws, int count, FILE *stream)`
'`wchar.h`' (ISO): Section 17.9 [Line-Oriented Input], page 455.

`wchar_t * fgetws_unlocked (wchar_t *ws, int count, FILE *stream)`
'`wchar.h`' (GNU): Section 17.9 [Line-Oriented Input], page 455.

`FILE`
'`stdio.h`' (ISO): Section 17.1 [Streams], page 439.

`int finite (double x)`
'`math.h`' (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int finitef (float x)`
'`math.h`' (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int finitel (long double x)`
'`math.h`' (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int __flbf (FILE *stream)`
'`stdio_ext.h`' (GNU): Section 17.20.3 [Controlling Which Kind of Buffering], page 506.

`void flockfile (FILE *stream)`
'`stdio.h`' (POSIX): Section 17.5 [Streams and Threads], page 445.

`double floor (double x)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`float floorf (float x)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`long double floorl (long double x)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`void _flushlbf (void)`
'`stdio_ext.h`' (GNU): Section 17.20.2 [Flushing Buffers], page 505.

`double fma (double x, double y, double z)`
'`math.h`' (ISO): Section 9.8.7 [Miscellaneous FP Arithmetic Functions], page 265.

`float fmaf (float x, float y, float z)`
'`math.h`' (ISO): Section 9.8.7 [Miscellaneous FP Arithmetic Functions], page 265.

`long double fmal (long double x, long double y, long double z)`
'`math.h`' (ISO): Section 9.8.7 [Miscellaneous FP Arithmetic Functions], page 265.

`double fmax (double x, double y)`
'`math.h`' (ISO): Section 9.8.7 [Miscellaneous FP Arithmetic Functions], page 265.

`float fmaxf (float x, float y)`
'`math.h`' (ISO): Section 9.8.7 [Miscellaneous FP Arithmetic Functions], page 265.

`int FOPEN_MAX`
'`stdio.h`' (ISO): Section 17.3 [Opening Streams], page 440.

`int fpclassify (`*float-type x*`)`
'`math.h`' (ISO): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`size_t __fpending (FILE *`*stream*`) The __fpending`
'`stdio_ext.h`' (GNU): Section 17.20.3 [Controlling Which Kind of Buffering], page 506.

`int FP_ILOGB0`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`int FP_ILOGBNAN`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`fpos64_t`
'`stdio.h`' (Unix98): Section 17.19 [Portable File-Position Functions], page 502.

`fpos_t`
'`stdio.h`' (ISO): Section 17.19 [Portable File-Position Functions], page 502.

`int fprintf (FILE *`*stream*`, const char *`*template*`, ...)`
'`stdio.h`' (ISO): Section 17.12.7 [Formatted Output Functions], page 470.

`void __fpurge (FILE *`*stream*`)`
'`stdio_ext.h`' (GNU): Section 17.20.2 [Flushing Buffers], page 505.

`int fputc (int `*c*`, FILE *`*stream*`)`
'`stdio.h`' (ISO): Section 17.7 [Simple Output by Characters or Lines], page 450.

`int fputc_unlocked (int `*c*`, FILE *`*stream*`)`
'`stdio.h`' (POSIX): Section 17.7 [Simple Output by Characters or Lines], page 450.

`int fputs (const char *`*s*`, FILE *`*stream*`)`
'`stdio.h`' (ISO): Section 17.7 [Simple Output by Characters or Lines], page 450.

`int fputs_unlocked (const char *`*s*`, FILE *`*stream*`)`
'`stdio.h`' (GNU): Section 17.7 [Simple Output by Characters or Lines], page 450.

`wint_t fputwc (wchar_t `*wc*`, FILE *`*stream*`)`
'`wchar.h`' (ISO): Section 17.7 [Simple Output by Characters or Lines], page 450.

`wint_t fputwc_unlocked (wint_t `*wc*`, FILE *`*stream*`)`
'`wchar.h`' (POSIX): Section 17.7 [Simple Output by Characters or Lines], page 450.

`int fputws (const wchar_t *`*ws*`, FILE *`*stream*`)`
'`wchar.h`' (ISO): Section 17.7 [Simple Output by Characters or Lines], page 450.

`int fputws_unlocked (const wchar_t *`*ws*`, FILE *`*stream*`)`
'`wchar.h`' (GNU): Section 17.7 [Simple Output by Characters or Lines], page 450.

`size_t fread (void *`*data*`, size_t `*size*`, size_t `*count*`, FILE *`*stream*`)`
'`stdio.h`' (ISO): Section 17.11 [Block Input/Output], page 459.

`int __freadable (FILE *`*stream*`)`
'`stdio_ext.h`' (GNU): Section 17.3 [Opening Streams], page 440.

`void funlockfile (FILE *stream)`
'`stdio.h`' (POSIX): Section 17.5 [Streams and Threads], page 445.

`int fwide (FILE *stream, int mode)`
'`wchar.h`' (ISO): Section 17.6 [Streams in Internationalized Applications], page 448.

`int fwprintf (FILE *stream, const wchar_t *template, ...)`
'`wchar.h`' (ISO): Section 17.12.7 [Formatted Output Functions], page 470.

`int __fwritable (FILE *stream)`
'`stdio_ext.h`' (GNU): Section 17.3 [Opening Streams], page 440.

`size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)`
'`stdio.h`' (ISO): Section 17.11 [Block Input/Output], page 459.

`size_t fwrite_unlocked (const void *data, size_t size, size_t count, FILE *stream)`
'`stdio.h`' (GNU): Section 17.11 [Block Input/Output], page 459.

`int __fwriting (FILE *stream)`
'`stdio_ext.h`' (GNU): Section 17.3 [Opening Streams], page 440.

`int fwscanf (FILE *stream, const wchar_t *template, ...)`
'`wchar.h`' (ISO): Section 17.14.8 [Formatted Input Functions], page 495.

`double gamma (double x)`
'`math.h`' (SVID): Section 8.6 [Special Functions], page 214.

`float gammaf (float x)`
'`math.h`' (SVID): Section 8.6 [Special Functions], page 214.

`long double gammal (long double x)`
'`math.h`' (SVID): Section 8.6 [Special Functions], page 214.

`void (*__gconv_end_fct) (struct gconv_step *)`
'`gconv.h`' (GNU): Section 6.5.4 [The `iconv` Implementation in the GNU C Library], page 165.

`int (*__gconv_fct) (struct __gconv_step *, struct __gconv_step_data *, const char **, const char *, size_t *, int)`
'`gconv.h`' (GNU): Section 6.5.4 [The `iconv` Implementation in the GNU C Library], page 165.

`int (*__gconv_init_fct) (struct __gconv_step *)`
'`gconv.h`' (GNU): Section 6.5.4 [The `iconv` Implementation in the GNU C Library], page 165.

`char * gcvt (double value, int ndigit, char *buf)`
'`stdlib.h`' (SVID, Unix98): Section 9.12 [Old-fashioned System V Number-to-String Functions], page 275.

`int getc (FILE *stream)`
'`stdio.h`' (ISO): Section 17.8 [Character Input], page 453.

`int getchar (void)`
'`stdio.h`' (ISO): Section 17.8 [Character Input], page 453.

`int getchar_unlocked (void)`
'`stdio.h`' (POSIX): Section 17.8 [Character Input], page 453.

`wint_t getwchar_unlocked (void)`
'`wchar.h`' (GNU): Section 17.8 [Character Input], page 453.

`wint_t getwc_unlocked (FILE *`*stream*`)`
'`wchar.h`' (GNU): Section 17.8 [Character Input], page 453.

`int glob (const char *`*pattern*`, int` *flags*`, int (*`*errfunc*`) (const char *`*filename*`, int` *error-code*`), glob_t *`*vector-ptr*`)`
'`glob.h`' (POSIX.2): Section 13.2.1 [Calling `glob`], page 357.

`int glob64 (const char *`*pattern*`, int` *flags*`, int (*`*errfunc*`) (const char *`*filename*`, int` *error-code*`), glob64_t *`*vector-ptr*`)`
'`glob.h`' (GNU): Section 13.2.1 [Calling `glob`], page 357.

`glob64_t`
'`glob.h`' (GNU): Section 13.2.1 [Calling `glob`], page 357.

`GLOB_ABORTED`
'`glob.h`' (POSIX.2): Section 13.2.1 [Calling `glob`], page 357.

`GLOB_ALTDIRFUNC`
'`glob.h`' (GNU): Section 13.2.3 [More Flags for Globbing], page 362.

`GLOB_APPEND`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`GLOB_BRACE`
'`glob.h`' (GNU): Section 13.2.3 [More Flags for Globbing], page 362.

`GLOB_DOOFFS`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`GLOB_ERR`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`void globfree (glob_t *`*pglob*`)`
'`glob.h`' (POSIX.2): Section 13.2.3 [More Flags for Globbing], page 362.

`void globfree64 (glob64_t *`*pglob*`)`
'`glob.h`' (GNU): Section 13.2.3 [More Flags for Globbing], page 362.

`GLOB_MAGCHAR`
'`glob.h`' (GNU): Section 13.2.3 [More Flags for Globbing], page 362.

`GLOB_MARK`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`GLOB_NOCHECK`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`GLOB_NOESCAPE`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`GLOB_NOMAGIC`
'`glob.h`' (GNU): Section 13.2.3 [More Flags for Globbing], page 362.

`GLOB_NOMATCH`
'`glob.h`' (POSIX.2): Section 13.2.1 [Calling `glob`], page 357.

`GLOB_NOSORT`
'`glob.h`' (POSIX.2): Section 13.2.2 [Flags for Globbing], page 361.

`float hypotf (float x, float y)`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double hypotl (long double x, long double y)`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`size_t iconv (iconv_t cd, char **inbuf, size_t *inbytesleft, char **outbuf, size_t *outbytesleft)`
'`iconv.h`' (XPG2): Section 6.5.1 [Generic Character-Set Conversion Interface], page 157.

`int iconv_close (iconv_t cd)`
'`iconv.h`' (XPG2): Section 6.5.1 [Generic Character-Set Conversion Interface], page 157.

`iconv_t iconv_open (const char *tocode, const char *fromcode)`
'`iconv.h`' (XPG2): Section 6.5.1 [Generic Character-Set Conversion Interface], page 157.

`iconv_t`
'`iconv.h`' (XPG2): Section 6.5.1 [Generic Character-Set Conversion Interface], page 157.

`int ilogb (double x)`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`int ilogbf (float x)`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`int ilogbl (long double x)`
'`math.h`' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`intmax_t imaxabs (intmax_t number)`
'`inttypes.h`' (ISO): Section 9.8.1 [Absolute Value], page 258.

`imaxdiv_t imaxdiv (intmax_t numerator, intmax_t denominator)`
'`inttypes.h`' (ISO): Section 9.2 [Integer Division], page 244.

`imaxdiv_t`
'`inttypes.h`' (ISO): Section 9.2 [Integer Division], page 244.

`char * index (const char *string, int c)`
'`string.h`' (BSD): Section 5.7 [Search Functions], page 114.

`float INFINITY`
'`math.h`' (ISO): Section 9.5.2 [Infinity and NaN], page 250.

`void * initstate (unsigned int seed, void *state, size_t size)`
'`stdlib.h`' (BSD): Section 8.8.2 [BSD Random-Number Functions], page 235.

`int initstate_r (unsigned int seed, char *restrict statebuf, size_t statelen, struct random_data *restrict buf)`
'`stdlib.h`' (GNU): Section 8.8.2 [BSD Random-Number Functions], page 235.

`int _IOFBF`
'`stdio.h`' (ISO): Section 17.20.3 [Controlling Which Kind of Buffering], page 506.

`int _IOLBF`
'`stdio.h`' (ISO): Section 17.20.3 [Controlling Which Kind of Buffering], page 506.

`int _IONBF`
: ‘`stdio.h`’ (ISO): Section 17.20.3 [Controlling Which Kind of Buffering], page 506.

`int isalnum (int` *c*`)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isalpha (int` *c*`)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isascii (int` *c*`)`
: ‘`ctype.h`’ (SVID, BSD): Section 4.1 [Classification of Characters], page 79.

`int isblank (int` *c*`)`
: ‘`ctype.h`’ (GNU): Section 4.1 [Classification of Characters], page 79.

`int iscntrl (int` *c*`)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isdigit (int` *c*`)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isfinite (`*float-type x*`)`
: ‘`math.h`’ (ISO): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isgraph (int` *c*`)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isgreater (`*real-floating x*`, `*real-floating y*`)`
: ‘`math.h`’ (ISO): Section 9.8.6 [Floating-Point Comparison Functions], page 264.

`int isgreaterequal (`*real-floating x*`, `*real-floating y*`)`
: ‘`math.h`’ (ISO): Section 9.8.6 [Floating-Point Comparison Functions], page 264.

`int isinf (double` *x*`)`
: ‘`math.h`’ (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isinff (float` *x*`)`
: ‘`math.h`’ (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isinfl (long double` *x*`)`
: ‘`math.h`’ (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isless (`*real-floating x*`, `*real-floating y*`)`
: ‘`math.h`’ (ISO): Section 9.8.6 [Floating-Point Comparison Functions], page 264.

`int islessequal (`*real-floating x*`, `*real-floating y*`)`
: ‘`math.h`’ (ISO): Section 9.8.6 [Floating-Point Comparison Functions], page 264.

`int islessgreater (`*real-floating x*`, `*real-floating y*`)`
: ‘`math.h`’ (ISO): Section 9.8.6 [Floating-Point Comparison Functions], page 264.

`int islower (int` *c*`)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isnan (double` *x*`)`
: ‘`math.h`’ (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isnan (`*float-type x*`)`
: ‘`math.h`’ (ISO): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isnanf (float x)`
: ‘`math.h`’ (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isnanl (long double x)`
: ‘`math.h`’ (BSD): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`int isnormal (`*float-type x*`)`
: ‘`math.h`’ (ISO): Section 9.4 [Floating-Point Number Classification Functions], page 247.

`_ISOC99_SOURCE`
: (GNU): Section 1.3.4 [Feature-Test Macros], page 8.

`int isprint (int c)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int ispunct (int c)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isspace (int c)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int isunordered (`*real-floating x*`, `*real-floating y*`)`
: ‘`math.h`’ (ISO): Section 9.8.6 [Floating-Point Comparison Functions], page 264.

`int isupper (int c)`
: ‘`ctype.h`’ (ISO): Section 4.1 [Classification of Characters], page 79.

`int iswalnum (wint_t wc)`
: ‘`wctype.h`’ (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int iswalpha (wint_t wc)`
: ‘`wctype.h`’ (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int iswblank (wint_t wc)`
: ‘`wctype.h`’ (GNU): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int iswcntrl (wint_t wc)`
: ‘`wctype.h`’ (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int iswctype (wint_t wc, wctype_t desc)`
: ‘`wctype.h`’ (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int iswdigit (wint_t wc)`
: ‘`wctype.h`’ (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int iswgraph (wint_t wc)`
: ‘`wctype.h`’ (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`long double jnl (int n, long double x)`
'`math.h`' (SVID): Section 8.6 [Special Functions], page 214.

`long int jrand48 (unsigned short int xsubi[3])`
'`stdlib.h`' (SVID): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int jrand48_r (unsigned short int xsubi[3], struct drand48_data *buffer, long int *result)`
'`stdlib.h`' (GNU): Section 8.8.3 [SVID Random-Number Functions], page 237.

`char * l64a (long int n)`
'`stdlib.h`' (XPG): Section 5.11 [Encode Binary Data], page 125.

`long int labs (long int number)`
'`stdlib.h`' (ISO): Section 9.8.1 [Absolute Value], page 258.

`LANG`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`LC_ALL`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`LC_COLLATE`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`LC_CTYPE`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`LC_MESSAGES`
'`locale.h`' (XOPEN): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`LC_MONETARY`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`LC_NUMERIC`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`void lcong48 (unsigned short int param[7])`
'`stdlib.h`' (SVID): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int lcong48_r (unsigned short int param[7], struct drand48_data *buffer)`
'`stdlib.h`' (GNU): Section 8.8.3 [SVID Random-Number Functions], page 237.

`LC_TIME`
'`locale.h`' (ISO): Section 7.3 [Categories of Activities That Locales Affect], page 182.

`double ldexp (double value, int exponent)`
'`math.h`' (ISO): Section 9.8.2 [Normalization Functions], page 259.

`float ldexpf (float value, int exponent)`
'`math.h`' (ISO): Section 9.8.2 [Normalization Functions], page 259.

`struct lconv * localeconv (void)`
: 'locale.h' (ISO): Section 7.6.1 [localeconv: "It is portable, but ...´], page 186.

`struct tm * localtime (const time_t *time)`
: 'time.h' (ISO): Section 10.4.3 [Broken-Down Time], page 285.

`struct tm * localtime_r (const time_t *time, struct tm *resultp)`
: 'time.h' (POSIX.1c): Section 10.4.3 [Broken-Down Time], page 285.

`double log (double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`double log10 (double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`float log10f (float x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double log10l (long double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`double log1p (double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`float log1pf (float x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double log1pl (long double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`double log2 (double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`float log2f (float x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double log2l (long double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`double logb (double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`float logbf (float x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double logbl (long double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`float logf (float x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long double logl (long double x)`
: 'math.h' (ISO): Section 8.4 [Exponentiation and Logarithms], page 207.

`long int lrand48 (void)`
: 'stdlib.h' (SVID): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int lrand48_r (struct drand48_data *buffer, double *result)`
: 'stdlib.h' (GNU): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int mbsinit (const mbstate_t *ps)`
'wchar.h' (ISO): Section 6.3.2 [Representing the State of the Conversion], page 139.

`size_t mbsnrtowcs (wchar_t *restrict dst, const char **restrict src, size_t nmc, size_t len, mbstate_t *restrict ps)`
'wchar.h' (GNU): Section 6.3.4 [Converting Multibyte- and Wide-Character Strings], page 147.

`size_t mbsrtowcs (wchar_t *restrict dst, const char **restrict src, size_t len, mbstate_t *restrict ps)`
'wchar.h' (ISO): Section 6.3.4 [Converting Multibyte- and Wide-Character Strings], page 147.

`mbstate_t`
'wchar.h' (ISO): Section 6.3.2 [Representing the State of the Conversion], page 139.

`size_t mbstowcs (wchar_t *wstring, const char *string, size_t size)`
'stdlib.h' (ISO): Section 6.4.2 [Nonreentrant Conversion of Strings], page 154.

`int mbtowc (wchar_t *restrict result, const char *restrict string, size_t size)`
'stdlib.h' (ISO): Section 6.4.1 [Nonreentrant Conversion of Single Characters], page 153.

`int mcheck (void (*abortfn) (enum mcheck_status status))`
'mcheck.h' (GNU): Section 3.2.2.9 [Heap Consistency Checking], page 48.

`void * memalign (size_t boundary, size_t size)`
'malloc.h' (BSD): Section 3.2.2.7 [Allocating Aligned Memory Blocks], page 47.

`__memalign_hook`
'malloc.h' (GNU): Section 3.2.2.10 [Memory Allocation Hooks], page 50.

`void * memccpy (void *restrict to, const void *restrict from, int c, size_t size)`
'string.h' (SVID): Section 5.4 [Copying and Concatenation], page 93.

`void * memchr (const void *block, int c, size_t size)`
'string.h' (ISO): Section 5.7 [Search Functions], page 114.

`int memcmp (const void *a1, const void *a2, size_t size)`
'string.h' (ISO): Section 5.5 [String/Array Comparison], page 105.

`void * memcpy (void *restrict to, const void *restrict from, size_t size)`
'string.h' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`void * memfrob (void *mem, size_t length)`
'string.h' (GNU): Section 5.10 [Trivial Encryption], page 124.

`void * memmem (const void *haystack, size_t haystack-len, const void *needle, size_t needle-len)`
'string.h' (GNU): Section 5.7 [Search Functions], page 114.

`void * memmove (void *to, const void *from, size_t size)`
'string.h' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`void * mempcpy (void *restrict to, const void *restrict from, size_t size)`
'string.h' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`long double nanl (const char *tagp)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`int nanosleep (const struct timespec *requested_time, struct timespec *remaining)`
'`time.h`' (POSIX.1): Section 10.6 [Sleeping], page 312.

`double nearbyint (double x)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`float nearbyintf (float x)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`long double nearbyintl (long double x)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`double nextafter (double x, double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`float nextafterf (float x, float y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`long double nextafterl (long double x, long double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`double nexttoward (double x, long double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`float nexttowardf (float x, long double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`long double nexttowardl (long double x, long double y)`
'`math.h`' (ISO): Section 9.8.5 [Setting and Modifying Single Bits of FP Values], page 263.

`char * ngettext (const char *msgid1, const char *msgid2, unsigned long int n)`
'`libintl.h`' (GNU): Section 11.2.1.3 [Additional Functions for More Complicated Situations], page 330.

`char * nl_langinfo (nl_item item)`
'`langinfo.h`' (XOPEN): Section 7.6.2 [Pinpoint Access to Locale Data], page 191.

`long int nrand48 (unsigned short int xsubi[3])`
'`stdlib.h`' (SVID): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int nrand48_r (unsigned short int xsubi[3], struct drand48_data *buffer, long int *result)`
'`stdlib.h`' (GNU): Section 8.8.3 [SVID Random-Number Functions], page 237.

`int ntp_adjtime (struct timex *tptr)`
'`sys/timex.h`' (GNU): Section 10.4.4 [High-Accuracy Clock], page 288.

`REG_BADBR`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_BADPAT`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_BADRPT`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`int regcomp (regex_t *`*compiled*`, const char *`*pattern*`, int` *cflags*`)`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_EBRACE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_EBRACK`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_ECOLLATE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_ECTYPE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_EESCAPE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_EPAREN`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_ERANGE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`size_t regerror (int` *errcode*`, regex_t *`*compiled*`, char *`*buffer*`, size_t` *length*`)`
: ‘`regex.h`’ (POSIX.2): Section 13.3.6 [POSIX Regexp Matching Clean-Up], page 369.

`REG_ESPACE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.3 [Matching a Compiled POSIX Regular Expression], page 367.

`REG_ESPACE`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`REG_ESUBREG`
: ‘`regex.h`’ (POSIX.2): Section 13.3.1 [POSIX Regular Expression Compilation], page 365.

`long double remainderl (long double` *numerator*`, long double` *denominator*`)`
'`math.h`' (BSD): Section 9.8.4 [Remainder Functions], page 262.

`void rewind (FILE *`*stream*`)`
'`stdio.h`' (ISO): Section 17.18 [File Positioning], page 500.

`char * rindex (const char *`*string*`, int` *c*`)`
'`string.h`' (BSD): Section 5.7 [Search Functions], page 114.

`double rint (double` *x*`)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`float rintf (float` *x*`)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`long double rintl (long double` *x*`)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`double round (double` *x*`)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`float roundf (float` *x*`)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`long double roundl (long double` *x*`)`
'`math.h`' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`int rpmatch (const char *`*response*`)`
'`stdlib.h`' (stdlib.h): Section 7.8 [Yes-or-No Questions], page 200.

`int sbrk (ptrdiff_t` *delta*`)`
'`unistd.h`' (BSD): Section 3.3 [Resizing the Data Segment], page 74.

`double scalb (double` *value*`, int` *exponent*`)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`float scalbf (float` *value*`, int` *exponent*`)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long double scalbl (long double` *value*`, int` *exponent*`)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long long int scalbln (double` *x*`, long int n)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long long int scalblnf (float` *x*`, long int n)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long long int scalblnl (long double` *x*`, long int n)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long long int scalbn (double` *x*`, int n)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long long int scalbnf (float` *x*`, int n)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`long long int scalbnl (long double` *x*`, int n)`
'`math.h`' (BSD): Section 9.8.2 [Normalization Functions], page 259.

`int scanf (const char *`*template*`, ...)`
'`stdio.h`' (ISO): Section 17.14.8 [Formatted Input Functions], page 495.

`FILE * stderr`
'`stdio.h`' (ISO): Section 17.2 [Standard Streams], page 439.

`FILE * stdin`
'`stdio.h`' (ISO): Section 17.2 [Standard Streams], page 439.

`FILE * stdout`
'`stdio.h`' (ISO): Section 17.2 [Standard Streams], page 439.

`int stime (time_t *`*newtime*`)`
'`time.h`' (SVID, XPG): Section 10.4.1 [Simple Calendar Time], page 282.

`char * stpcpy (char *restrict` *to*`, const char *restrict` *from*`)`
'`string.h`' (Unknown origin): Section 5.4 [Copying and Concatenation], page 93.

`char * stpncpy (char *restrict` *to*`, const char *restrict` *from*`, size_t` *size*`)`
'`string.h`' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`int strcasecmp (const char *`*s1*`, const char *`*s2*`)`
'`string.h`' (BSD): Section 5.5 [String/Array Comparison], page 105.

`char * strcasestr (const char *`*haystack*`, const char *`*needle*`)`
'`string.h`' (GNU): Section 5.7 [Search Functions], page 114.

`char * strcat (char *restrict` *to*`, const char *restrict` *from*`)`
'`string.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`char * strchr (const char *`*string*`, int` *c*`)`
'`string.h`' (ISO): Section 5.7 [Search Functions], page 114.

`char * strchrnul (const char *`*string*`, int` *c*`)`
'`string.h`' (GNU): Section 5.7 [Search Functions], page 114.

`int strcmp (const char *`*s1*`, const char *`*s2*`)`
'`string.h`' (ISO): Section 5.5 [String/Array Comparison], page 105.

`int strcoll (const char *`*s1*`, const char *`*s2*`)`
'`string.h`' (ISO): Section 5.6 [Collation Functions], page 109.

`char * strcpy (char *restrict` *to*`, const char *restrict` *from*`)`
'`string.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`size_t strcspn (const char *`*string*`, const char *`*stopset*`)`
'`string.h`' (ISO): Section 5.7 [Search Functions], page 114.

`char * strdup (const char *`*s*`)`
'`string.h`' (SVID): Section 5.4 [Copying and Concatenation], page 93.

`char * strdupa (const char *`*s*`)`
'`string.h`' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`char * strerror (int` *errnum*`)`
'`string.h`' (ISO): Section 2.3 [Error Messages], page 32.

`char * strerror_r (int` *errnum*`, char *`*buf*`, size_t` *n*`)`
'`string.h`' (GNU): Section 2.3 [Error Messages], page 32.

`char * strfry (char *`*string*`)`
'`string.h`' (GNU): Section 5.9 [`strfry`], page 124.

`size_t strftime (char *`*s*`, size_t` *size*`, const char *`*template*`, const struct tm *`*brokentime*`)`
'`time.h`' (ISO): Section 10.4.5 [Formatting Calendar Time], page 291.

`size_t strlen (const char *s)`
'`string.h`' (ISO): Section 5.3 [String Length], page 91.

`int strncasecmp (const char *s1, const char *s2, size_t n)`
'`string.h`' (BSD): Section 5.5 [String/Array Comparison], page 105.

`char * strncat (char *restrict to, const char *restrict from, size_t size)`
'`string.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`int strncmp (const char *s1, const char *s2, size_t size)`
'`string.h`' (ISO): Section 5.5 [String/Array Comparison], page 105.

`char * strncpy (char *restrict to, const char *restrict from, size_t size)`
'`string.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`char * strndup (const char *s, size_t size)`
'`string.h`' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`char * strndupa (const char *s, size_t size)`
'`string.h`' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`size_t strnlen (const char *s, size_t maxlen)`
'`string.h`' (GNU): Section 5.3 [String Length], page 91.

`char * strpbrk (const char *string, const char *stopset)`
'`string.h`' (ISO): Section 5.7 [Search Functions], page 114.

`char * strptime (const char *s, const char *fmt, struct tm *tp)`
'`time.h`' (XPG4): Section 10.4.6.1 [Interpret String According to Given Format], page 297.

`char * strrchr (const char *string, int c)`
'`string.h`' (ISO): Section 5.7 [Search Functions], page 114.

`char * strsep (char **string_ptr, const char *delimiter)`
'`string.h`' (BSD): Section 5.8 [Finding Tokens in a String], page 119.

`size_t strspn (const char *string, const char *skipset)`
'`string.h`' (ISO): Section 5.7 [Search Functions], page 114.

`char * strstr (const char *haystack, const char *needle)`
'`string.h`' (ISO): Section 5.7 [Search Functions], page 114.

`double strtod (const char *restrict string, char **restrict tailptr)`
'`stdlib.h`' (ISO): Section 9.11.2 [Parsing of Floats], page 273.

`float strtof (const char *string, char **tailptr)`
'`stdlib.h`' (ISO): Section 9.11.2 [Parsing of Floats], page 273.

`intmax_t strtoimax (const char *restrict string, char **restrict tailptr, int base)`
'`inttypes.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`char * strtok (char *restrict newstring, const char *restrict delimiters)`
'`string.h`' (ISO): Section 5.8 [Finding Tokens in a String], page 119.

`char * strtok_r (char *newstring, const char *delimiters, char **save_ptr)`
'`string.h`' (POSIX): Section 5.8 [Finding Tokens in a String], page 119.

`long int strtol (const char *restrict string, char **restrict tailptr, int base)`
'`stdlib.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`struct option`
: ‘`getopt.h`’ (GNU): Section 14.2.3 [Parsing Long Options with `getopt_long`], page 385.

`struct printf_info`
: ‘`printf.h`’ (GNU): Section 17.13.2 [Conversion Specifier Options], page 481.

`struct random_data`
: ‘`stdlib.h`’ (GNU): Section 8.8.2 [BSD Random-Number Functions], page 235.

`struct timespec`
: ‘`sys/time.h`’ (POSIX.1): Section 10.2 [Elapsed Time], page 277.

`struct timeval`
: ‘`sys/time.h`’ (BSD): Section 10.2 [Elapsed Time], page 277.

`struct timezone`
: ‘`sys/time.h`’ (BSD): Section 10.4.2 [High-Resolution Calendar], page 283.

`struct tm`
: ‘`time.h`’ (ISO): Section 10.4.3 [Broken-Down Time], page 285.

`struct tms`
: ‘`sys/times.h`’ (POSIX.1): Section 10.3.2 [Processor Time Inquiry], page 281.

`int strverscmp (const char *`*s1*`, const char *`*s2*`)`
: ‘`string.h`’ (GNU): Section 5.5 [String/Array Comparison], page 105.

`size_t strxfrm (char *restrict` *to*`, const char *restrict` *from*`, size_t` *size*`)`
: ‘`string.h`’ (ISO): Section 5.6 [Collation Functions], page 109.

`_SVID_SOURCE`
: (GNU): Section 1.3.4 [Feature-Test Macros], page 8.

`int swprintf (wchar_t *`*s*`, size_t` *size*`, const wchar_t *`*template*`, ...)`
: ‘`wchar.h`’ (GNU): Section 17.12.7 [Formatted Output Functions], page 470.

`int swscanf (const wchar_t *`*ws*`, const char *`*template*`, ...)`
: ‘`wchar.h`’ (ISO): Section 17.14.8 [Formatted Input Functions], page 495.

`long int syscall (long int` *sysno*`, ...)`
: ‘`unistd.h`’ (Undocumented): Section 14.5 [System Calls], page 423.

`double tan (double` *x*`)`
: ‘`math.h`’ (ISO): Section 8.2 [Trigonometric Functions], page 204.

`float tanf (float` *x*`)`
: ‘`math.h`’ (ISO): Section 8.2 [Trigonometric Functions], page 204.

`double tanh (double` *x*`)`
: ‘`math.h`’ (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`float tanhf (float` *x*`)`
: ‘`math.h`’ (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double tanhl (long double` *x*`)`
: ‘`math.h`’ (ISO): Section 8.5 [Hyperbolic Functions], page 212.

`long double tanl (long double` *x*`)`
: ‘`math.h`’ (ISO): Section 8.2 [Trigonometric Functions], page 204.

`wint_t towupper (wint_t wc)`
: 'wctype.h' (ISO): Section 4.5 [Mapping of Wide Characters], page 87.

`double trunc (double x)`
: 'math.h' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`float truncf (float x)`
: 'math.h' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`long double truncl (long double x)`
: 'math.h' (ISO): Section 9.8.3 [Rounding Functions], page 260.

`void * tsearch (const void *key, void **rootp, comparison_fn_t compar)`
: 'search.h' (SVID): Section 12.6 [The tsearch Function], page 351.

`void twalk (const void *root, __action_fn_t action)`
: 'search.h' (SVID): Section 12.6 [The tsearch Function], page 351.

`char * tzname [2]`
: 'time.h' (POSIX.1): Section 10.4.8 [Functions and Variables for Time Zones], page 308.

`void tzset (void)`
: 'time.h' (POSIX.1): Section 10.4.8 [Functions and Variables for Time Zones], page 308.

`int ungetc (int c, FILE *stream)`
: 'stdio.h' (ISO): Section 17.10.2 [Using ungetc to Do Unreading], page 458.

`wint_t ungetwc (wint_t wc, FILE *stream)`
: 'wchar.h' (ISO): Section 17.10.2 [Using ungetc to Do Unreading], page 458.

`int unsetenv (const char *name)`
: 'stdlib.h' (BSD): Section 14.4.1 [Environment Access], page 419.

`void * valloc (size_t size)`
: 'malloc.h', 'stdlib.h' (BSD): Section 3.2.2.7 [Allocating Aligned Memory Blocks], page 47.

`int vasprintf (char **ptr, const char *template, va_list ap)`
: 'stdio.h' (GNU): Section 17.12.9 [Variable Arguments Output Functions], page 474.

`void verr (int status, const char *format, va_list)`
: 'err.h' (BSD): Section 2.3 [Error Messages], page 32.

`void verrx (int status, const char *format, va_list)`
: 'err.h' (BSD): Section 2.3 [Error Messages], page 32.

`int vfprintf (FILE *stream, const char *template, va_list ap)`
: 'stdio.h' (ISO): Section 17.12.9 [Variable Arguments Output Functions], page 474.

`int vfscanf (FILE *stream, const char *template, va_list ap)`
: 'stdio.h' (ISO): Section 17.14.9 [Variable Arguments Input Functions], page 496.

`int vfwprintf (FILE *stream, const wchar_t *template, va_list ap)`
: 'wchar.h' (ISO): Section 17.12.9 [Variable Arguments Output Functions], page 474.

`wchar_t * wcpncpy (wchar_t *restrict wto, const wchar_t *restrict wfrom, size_t size)`
'`wchar.h`' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`size_t wcrtomb (char *restrict s, wchar_t wc, mbstate_t *restrict ps)`
'`wchar.h`' (ISO): Section 6.3.3 [Converting Single Characters], page 140.

`int wcscasecmp (const wchar_t *ws1, const wchar_T *ws2)`
'`wchar.h`' (GNU): Section 5.5 [String/Array Comparison], page 105.

`wchar_t * wcscat (wchar_t *restrict wto, const wchar_t *restrict wfrom)`
'`wchar.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`wchar_t * wcschr (const wchar_t *wstring, int wc)`
'`wchar.h`' (ISO): Section 5.7 [Search Functions], page 114.

`wchar_t * wcschrnul (const wchar_t *wstring, wchar_t wc)`
'`wchar.h`' (GNU): Section 5.7 [Search Functions], page 114.

`int wcscmp (const wchar_t *ws1, const wchar_t *ws2)`
'`wchar.h`' (ISO): Section 5.5 [String/Array Comparison], page 105.

`int wcscoll (const wchar_t *ws1, const wchar_t *ws2)`
'`wchar.h`' (ISO): Section 5.6 [Collation Functions], page 109.

`wchar_t * wcscpy (wchar_t *restrict wto, const wchar_t *restrict wfrom)`
'`wchar.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`size_t wcscspn (const wchar_t *wstring, const wchar_t *stopset)`
'`wchar.h`' (ISO): Section 5.7 [Search Functions], page 114.

`wchar_t * wcsdup (const wchar_t *ws)`
'`wchar.h`' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`size_t wcsftime (wchar_t *s, size_t size, const wchar_t *template, const struct tm *brokentime)`
'`time.h`' (ISO/Amend1): Section 10.4.5 [Formatting Calendar Time], page 291.

`size_t wcslen (const wchar_t *ws)`
'`wchar.h`' (ISO): Section 5.3 [String Length], page 91.

`int wcsncasecmp (const wchar_t *ws1, const wchar_t *s2, size_t n)`
'`wchar.h`' (GNU): Section 5.5 [String/Array Comparison], page 105.

`wchar_t * wcsncat (wchar_t *restrict wto, const wchar_t *restrict wfrom, size_t size)`
'`wchar.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`int wcsncmp (const wchar_t *ws1, const wchar_t *ws2, size_t size)`
'`wchar.h`' (ISO): Section 5.5 [String/Array Comparison], page 105.

`wchar_t * wcsncpy (wchar_t *restrict wto, const wchar_t *restrict wfrom, size_t size)`
'`wchar.h`' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`size_t wcsnlen (const wchar_t *ws, size_t maxlen)`
'`wchar.h`' (GNU): Section 5.3 [String Length], page 91.

`size_t wcsnrtombs (char *restrict dst, const wchar_t **restrict src, size_t nwc, size_t len, mbstate_t *restrict ps)`
'`wchar.h`' (GNU): Section 6.3.4 [Converting Multibyte- and Wide-Character Strings], page 147.

`wchar_t * wcspbrk (const wchar_t *wstring, const wchar_t *stopset)`
'`wchar.h`' (ISO): Section 5.7 [Search Functions], page 114.

`wchar_t * wcsrchr (const wchar_t *wstring, wchar_t c)`
'`wchar.h`' (ISO): Section 5.7 [Search Functions], page 114.

`size_t wcsrtombs (char *restrict dst, const wchar_t **restrict src, size_t len, mbstate_t *restrict ps)`
'`wchar.h`' (ISO): Section 6.3.4 [Converting Multibyte- and Wide-Character Strings], page 147.

`size_t wcsspn (const wchar_t *wstring, const wchar_t *skipset)`
'`wchar.h`' (ISO): Section 5.7 [Search Functions], page 114.

`wchar_t * wcsstr (const wchar_t *haystack, const wchar_t *needle)`
'`wchar.h`' (ISO): Section 5.7 [Search Functions], page 114.

`double wcstod (const wchar_t *restrict string, wchar_t **restrict tailptr)`
'`wchar.h`' (ISO): Section 9.11.2 [Parsing of Floats], page 273.

`float wcstof (const wchar_t *string, wchar_t **tailptr)`
'`stdlib.h`' (ISO): Section 9.11.2 [Parsing of Floats], page 273.

`intmax_t wcstoimax (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`wchar_t * wcstok (wchar_t *newstring, const char *delimiters)`
'`wchar.h`' (ISO): Section 5.8 [Finding Tokens in a String], page 119.

`long int wcstol (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`long double wcstold (const wchar_t *string, wchar_t **tailptr)`
'`stdlib.h`' (ISO): Section 9.11.2 [Parsing of Floats], page 273.

`long long int wcstoll (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`size_t wcstombs (char *string, const wchar_t *wstring, size_t size)`
'`stdlib.h`' (ISO): Section 6.4.2 [Nonreentrant Conversion of Strings], page 154.

`long long int wcstoq (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (GNU): Section 9.11.1 [Parsing of Integers], page 268.

`unsigned long int wcstoul (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`unsigned long long int wcstoull (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`uintmax_t wcstoumax (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'`wchar.h`' (ISO): Section 9.11.1 [Parsing of Integers], page 268.

`unsigned long long int wcstouq (const wchar_t *restrict string, wchar_t **restrict tailptr, int base)`
'wchar.h' (GNU): Section 9.11.1 [Parsing of Integers], page 268.

`wchar_t * wcswcs (const wchar_t *haystack, const wchar_t *needle)`
'wchar.h' (XPG): Section 5.7 [Search Functions], page 114.

`size_t wcsxfrm (wchar_t *restrict wto, const wchar_t *wfrom, size_t size)`
'wchar.h' (ISO): Section 5.6 [Collation Functions], page 109.

`int wctob (wint_t c)`
'wchar.h' (ISO): Section 6.3.3 [Converting Single Characters], page 140.

`int wctomb (char *string, wchar_t wchar)`
'stdlib.h' (ISO): Section 6.4.1 [Nonreentrant Conversion of Single Characters], page 153.

`wctrans_t wctrans (const char *property)`
'wctype.h' (ISO): Section 4.5 [Mapping of Wide Characters], page 87.

`wctrans_t`
'wctype.h' (ISO): Section 4.5 [Mapping of Wide Characters], page 87.

`wctype_t wctype (const char *property)`
'wctype.h' (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`wctype_t`
'wctype.h' (ISO): Section 4.3 [Character Class Determination for Wide Characters], page 82.

`int WEOF`
'wchar.h' (ISO): Section 17.15 [End-of-File and Errors], page 497.

`wint_t WEOF`
'wchar.h' (ISO): Section 6.1 [Introduction to Extended Characters], page 133.

`wint_t`
'wchar.h' (ISO): Section 6.1 [Introduction to Extended Characters], page 133.

`wchar_t * wmemchr (const wchar_t *block, wchar_t wc, size_t size)`
'wchar.h' (ISO): Section 5.7 [Search Functions], page 114.

`int wmemcmp (const wchar_t *a1, const wchar_t *a2, size_t size)`
'wcjar.h' (ISO): Section 5.5 [String/Array Comparison], page 105.

`wchar_t * wmemcpy (wchar_t *restrict wto, const wchar_t *restruct wfrom, size_t size)`
'wchar.h' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`wchar_t * wmemmove (wchar *wto, const wchar_t *wfrom, size_t size)`
'wchar.h' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`wchar_t * wmempcpy (wchar_t *restrict wto, const wchar_t *restrict wfrom, size_t size)`
'wchar.h' (GNU): Section 5.4 [Copying and Concatenation], page 93.

`wchar_t * wmemset (wchar_t *block, wchar_t wc, size_t size)`
'wchar.h' (ISO): Section 5.4 [Copying and Concatenation], page 93.

`double y1 (double x)`
: 'math.h' (SVID): Section 8.6 [Special Functions], page 214.

`float y1f (float x)`
: 'math.h' (SVID): Section 8.6 [Special Functions], page 214.

`long double y1l (long double x)`
: 'math.h' (SVID): Section 8.6 [Special Functions], page 214.

`double yn (int n, double x)`
: 'math.h' (SVID): Section 8.6 [Special Functions], page 214.

`float ynf (int n, float x)`
: 'math.h' (SVID): Section 8.6 [Special Functions], page 214.

`long double ynl (int n, long double x)`
: 'math.h' (SVID): Section 8.6 [Special Functions], page 214.

Appendix B Contributors to the GNU C Library

The GNU C Library was written originally by Roland McGrath, and is currently maintained by Ulrich Drepper. Some parts of the library were contributed or worked on by other people.

- The `getopt` function and related code were written by Richard Stallman, David J. MacKenzie and Roland McGrath.
- The merge sort function `qsort` was written by Michael J. Haertel.
- The quick sort function used as a fallback by `qsort` was written by Douglas C. Schmidt.
- The memory allocation functions `malloc`, `realloc` and `free` and related code were written by Michael J. Haertel, Wolfram Gloger and Doug Lea.
- Fast implementations of many of the string functions (`memcpy`, `strlen`, etc.) were written by Torbjörn Granlund.
- The '`tar.h`' header file was written by David J. MacKenzie.
- The port to the MIPS DECStation running Ultrix 4 (`mips-dec-ultrix4`) was contributed by Brendan Kehoe and Ian Lance Taylor.
- The DES encryption function `crypt` and related functions were contributed by Michael Glad.
- The `ftw` and `nftw` functions were contributed by Ulrich Drepper.
- The start-up code to support SunOS shared libraries was contributed by Tom Quinn.
- The `mktime` function was contributed by Paul Eggert.
- The port to the Sequent Symmetry running Dynix version 3 (`i386-sequent-bsd`) was contributed by Jason Merrill.
- The time zone support code is derived from the public-domain time zone package by Arthur David Olson and his many contributors.
- The port to the DEC Alpha running OSF/1 (`alpha-dec-osf1`) was contributed by Brendan Kehoe, using some code written by Roland McGrath.
- The port to SGI machines running Irix 4 (`mips-sgi-irix4`) was contributed by Tom Quinn.
- The port of the Mach and Hurd code to the MIPS architecture (`mips-`*anything*`-gnu`) was contributed by Kazumoto Kojima.
- The floating-point printing function used by `printf` and friends, and the floating-point reading function used by `scanf`, `strtod` and friends were written by Ulrich Drepper. The multiprecision integer functions used in those functions are taken from GNU MP, which was contributed by Torbjörn Granlund.
- The internationalization support in the library, and the support programs `locale` and `localedef`, were written by Ulrich Drepper. Ulrich Drepper adapted the support code for message catalogs ('`libintl.h`', etc.) from

the GNU `gettext` package, which he also wrote. He also contributed the `catgets` support and the entire suite of multibyte- and wide-character support functions (‘`wctype.h`’, ‘`wchar.h`’, etc.).

- The implementations of the ‘`nsswitch.conf`’ mechanism and the files and DNS backends for it were designed and written by Ulrich Drepper and Roland McGrath, based on a backend interface defined by Peter Eriksson.
- The port to Linux i386/ELF (`i386-`*anything*`-linux`) was contributed by Ulrich Drepper, based in large part on work done in Hongjiu Lu’s Linux version of the GNU C Library.
- The port to Linux/m68k (`m68k-`*anything*`-linux`) was contributed by Andreas Schwab.
- The ports to Linux/ARM (`arm-`*ANYTHING*`-linuxaout`) and ARM standalone (`arm-`*ANYTHING*`-none`), as well as parts of the IPv6 support code, were contributed by Philip Blundell.
- Richard Henderson contributed the ELF dynamic linking code and other support for the Alpha processor.
- David Mosberger-Tang contributed the port to Linux/Alpha (`alpha-`*anything*`-linux`).
- The port to Linux on PowerPC (`powerpc-`*anything*`-linux`) was contributed by Geoffrey Keating.
- Miles Bader wrote the argp argument-parsing package, and the argz/envz interfaces.
- Stephen R. van den Berg contributed a highly-optimized `strstr` function.
- Ulrich Drepper contributed the `hsearch` and `drand48` families of functions; reentrant ‘`..._r`’ versions of the `random` family; System V shared memory and IPC support code; and several highly-optimized string functions for i*x*86 processors.
- The math functions are taken from `fdlibm-5.1` by Sun Microsystems, as modified by J.T. Conklin, Ian Lance Taylor, Ulrich Drepper, Andreas Schwab and Roland McGrath.
- The `libio` library used to implement `stdio` functions on some platforms was written by Per Bothner and modified by Ulrich Drepper.
- Eric Youngdale and Ulrich Drepper implemented versioning of objects on the symbol level.
- Thorsten Kukuk provided an implementation for NIS (YP) and NIS+, securelevel 0, 1 and 2.
- Andreas Jaeger provided a test suite for the math library.
- Mark Kettenis implemented the utmpx interface and an utmp daemon.
- Ulrich Drepper added character conversion functions (`iconv`).
- Thorsten Kukuk provided an implementation for a caching daemon for NSS (nscd).

- Tim Waugh provided an implementation of the POSIX.2 wordexp function family.
- Mark Kettenis provided a Hesiod NSS module.
- The Internet-related code (most of the '`inet`' subdirectory) and several other miscellaneous functions and header files have been included from 4.4 BSD with little or no modification. The copying permission notice for this code can be found in the file '`LICENSES`' in the source distribution.
- The random-number generation functions `random`, `srandom`, `setstate` and `initstate`, which are also the basis for the `rand` and `srand` functions, were written by Earl T. Cohen for the University of California at Berkeley and are copyrighted by the Regents of the University of California. They have undergone minor changes to fit into the GNU C Library and to fit the ISO C standard, but the functional code is Berkeley's.
- The DNS resolver code is taken directly from BIND 4.9.5, which includes copyrighted code from UC Berkeley and from Digital Equipment Corporation. See the file '`LICENSES`' for the text of the DEC license.
- The code to support Sun RPC is taken verbatim from Sun's RPCSRC-4.0 distribution; see the file '`LICENSES`' for the text of the license.
- Some of the support code for Mach is taken from Mach 3.0 by CMU; the file if_ppp.h is also copyright by CMU, but under a different license; see the file '`LICENSES`' for the text of the licenses.
- Many of the IA64 math functions are taken from a collection of "Highly Optimized Mathematical Functions for Itanium" that Intel makes available under a free license; see the file '`LICENSES`' for details.
- The `getaddrinfo` and `getnameinfo` functions and supporting code were written by Craig Metz; see the file '`LICENSES`' for details on their licensing.
- Many of the IEEE 64-bit double precision math functions (in the '`sysdeps/ieee754/dbl-64`' subdirectory) come from the IBM Accurate Mathematical Library, contributed by IBM.

Appendix C Free Software Needs Free Documentation

The biggest deficiency in the free software community today is not in the software—it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a major gap. We have many such gaps today.

Consider Perl, for instance. The tutorial manuals that people normally use are nonfree. How did this come about? Because the authors of those manuals published them with restrictive terms—no copying, no modification, source files not available—which exclude them from the free software world.

That wasn't the first time this sort of thing happened, and it was far from the last. Many times we have heard a GNU user eagerly describe a manual that he is writing, his intended contribution to the community, only to learn that he had ruined everything by signing a publication contract to make it nonfree.

Free documentation, like free software, is a matter of freedom, not price. The problem with the nonfree manual is not that publishers charge a price for printed copies—that in itself is fine. (The Free Software Foundation sells printed copies of manuals, too.) The problem is the restrictions on the use of the manual. Free manuals are available in source code form, and give you permission to copy and modify. Nonfree manuals do not allow this.

The criteria of freedom for a free manual are roughly the same as for free software. Redistribution (including the normal kinds of commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, both on-line and on paper.

Permission for modification of the technical content is crucial too. When people modify the software, adding or changing features, if they are conscientious they will change the manual too—so they can provide accurate and clear documentation for the modified program. A manual that leaves you no choice but to write a new manual to document a changed version of the program is not really available to our community.

Some kinds of limits on the way modification is handled are acceptable. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified. Even entire sections that may not be deleted or changed are acceptable, as long as they deal with nontechnical topics (like this one). These kinds of restrictions are acceptable because they don't obstruct the community's normal use of the manual.

However, it must be possible to modify all the *technical* content of the manual, and then distribute the result in all the usual media, through all the usual channels. Otherwise, the restrictions obstruct the use of the manual, it is not free, and we need another manual to replace it.

Please spread the word about this issue. Our community continues to lose manuals to proprietary publishing. If we spread the word that free software needs free reference manuals and free tutorials, perhaps the next person who wants to contribute by writing documentation will realize, before it is too late, that only free manuals contribute to the free software community.

If you are writing documentation, please insist on publishing it under the GNU Free Documentation License or another free documentation license. Remember that this decision requires your approval—you don't have to let the publisher decide. Some commercial publishers will use a free license if you insist, but they will not propose the option; it is up to you to raise the issue and say firmly that this is what you want. If the publisher you are dealing with refuses, please try other publishers. If you're not sure whether a proposed license is free, write to licensing@gnu.org.

You can encourage commercial publishers to sell more free, copylefted manuals and tutorials by buying them, and particularly by buying copies from the publishers that paid for their writing or for major improvements. Meanwhile, try to avoid buying nonfree documentation at all. Check the distribution terms of a manual before you buy it, and insist that whoever seeks your business must respect your freedom. Check the history of the book, and try reward the publishers that have paid or pay the authors to work on it.

The Free Software Foundation maintains a list of free documentation published by other publishers, at http://www.fsf.org/doc/other-free-books.html.

Appendix D GNU Lesser General Public License

Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

D.0.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into nonfree programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the *Lesser* General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing nonfree programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, nonfree programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used nonfree libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in nonfree programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in nonfree programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and

a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

D.0.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms

of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small in-line functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

   a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete

machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.
10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.
11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder

who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF

THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

D.0.3 How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the library's name and an idea of what it does.*
> Copyright (C) *year* *name of author*
>
> This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.
>
> This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
>
> You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.
>
> *signature of Ty Coon*, 1 April 1990
> Ty Coon, President of Vice

That's all there is to it!

Appendix E GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title

equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

   C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

   D. Preserve all the copyright notices of the Document.

   E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one

of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http:// www.gnu.org/ copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

E.0.1 ADDENDUM: How to Use This License for Your Documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year* *your name*.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being *list their titles*, with the
> Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.
> A copy of the license is included in the section entitled ``GNU
> Free Documentation License''.

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Type Index

Function and Macro Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Variable and Constant Macro Index

(

-

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W

Y

Program and File Index

U

W

Z