Потоци (*Rev: 1.3*)

Любомир Чорбаджиев[1]
lchorbadjiev@elsys-bg.org

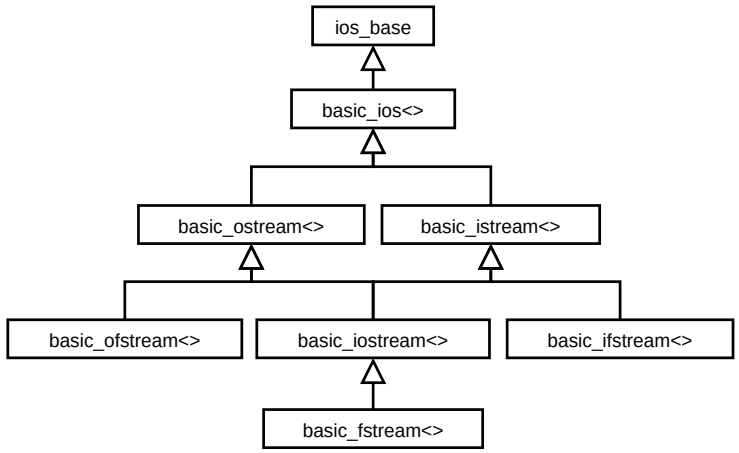
12 април 2006 г.

1. Йерархия на потоците за вход/изход

Въведение

- Входно/изходните потоци в C++ са обектно ориентирани.
- Входно/изходните операции в C++ са **строго типизирани**. Това позволява при обработването на данни с различен тип да се използва C++ механизмът за предефиниране на оператори и функции.
- Входно/изходните операции са **разширяеми**. Лесно се реализират входно/изходни операции за типове, дефинирани от потребителя.

Йерархия на потоците за вход/изход



Йерархия на потоците за вход/изход

- Потоците за вход/изход са организирани като йерархия от шаблонни класове.
- Потоците, които се използват обичайно са: `ostream`, `istream`, `ofstream`, `ifstream`. Тези потоци са 8 битови – т.е. последователността от символи, които се четат/пишат са от типа **char**.
- Дефинициите на тези потоци са следните:

```
typedef basic_ostream<char> ostream;
typedef basic_istream<char> istream;
typedef basic_ofstream<char> ofstream;
typedef basic_ifstream<char> ifstream;
typedef basic_fstream<char> fstream;
```

Йерархия на потоците за вход/изход

- В езика C++ освен типа **char** (8 битов символ) е дефиниран и типа **wchar_t** (16 битов символ);
- Потоците, които работят с типа **wchar_t** са:

```
typedef basic_ostream<wchar_t> wostream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_fstream<wchar_t> wfstream;
```

Работа с потоците за вход/изход

- Всички потоци за вход/изход са дефинирани в пространството от имена `std`.
- Основните потоци са дефинирани в заглавния файл `<iostream>`. В него са дефинирани обектите:
  - `cin` — обект от класа `istream`, който е свързан със стандартния вход.
  - `cout` — обект от класа `ostream`, който е свързан със стандартния изход.
  - `cerr` — обект от класа `ostream`, който е свързан със стандартната грешка. Този поток е **небуфериран**.

2. Писане в поток

Писане в поток

– За писане в изходен поток се използва операторът **operator**<<. В класа `basic_ostream` този оператор е дефиниран за всички примитивни типове:

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ostream:
        virtual public basic_ios<Ch, Tr> {
public:
  //...
  basic_ostream& operator<<(short n);
  basic_ostream& operator<<(int n);
  basic_ostream& operator<<(long n);
  basic_ostream& operator<<(unsigned short n);
  //...
};
```

Писане в поток

– Операторът **operator**<< връща псевдоним, насочен към използвания изходен поток `ostream`. Това дава възможност операторът за изход да се прилага каскадно. Изразът:

```
cout << "x=" << x;
```

е еквивалентен на:

```
(cout.operator<<("x=")).operator<<(x);
```

Операции за изход, дефинирани от потребителя

– Да разгледаме клас `point`, определен от потребителя:

```
class point {
  double x_, y_;
public:
  point(double x, double y)
    : x_(x), y_(y)
  {}
  double get_x(void) {return x_;}
  double get_y(void) {return y_;}
  void set_x(double x) {x_=x;}
  void set_y(double y) {y_=y;}
  //...
};
```


Операции за изход, дефинирани от потребителя

– Операторът **operator**<< за този тип може да бъде дефиниран по следния начин:

```
ostream& operator<<(ostream& out,
                    const point& p) {
  out << '('
      << p.get_x() << ','
      << p.get_y() << ')';
  return out;
}
```

3. Четене от поток

Четене от поток

– За четене от входен поток се използва операторът **operator**>>. В класа `basic_istream` този оператор е дефиниран за всички примитивни типове:

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_istream:
        virtual public basic_ios<Ch, Tr> {
public:
  //...
  basic_istream& operator>>(short& n);
  basic_istream& operator>>(int& n);
  basic_istream& operator>>(long& n);
  basic_istream& operator>>(unsigned short& n);
  //...
};
```

Четене от поток

– Операторът **operator>>** пропуска символите разделители (’␣’, ’\n’, ’\r’, ’\t’, ’\f’, ’\v’).

4. Състояние на потока

Състояние на потока

– Най-разпространената грешка при използване на потоци за вход `istream`: това, което е в потока се различава от това, което очакваме да бъде в потока. Например: искаме да прочетем променлива от типа **int**, а в потока има букви.

– За да се предпазим от този тип грешки е необходимо преди да се използват прочетените от потока данни да се провери **състоянието** на потока.

Състояние на потока

– Всеки поток `istream` и `ostream` има свързано с него **състояние**. Състоянието на потока е дефинирано в базовия клас `basic_ios<>`.

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ios: public ios_base {
public:
  //...
  bool good(void) const;
  bool eof(void) const;
  bool fail(void) const;
  bool bad(void) const;
  //...
};
```

Състояние на потока

– `good()` – предходните операции са изпълнени успешно;

– `eof()` – вижда се краят на файла;

– `fail()` – следващата операция няма да се изпълни успешно;

– `bad()` – потокът е повреден.

Състояние на потока

– Състоянието на потока представлява набор от флагове, които са дефинирани в базовия клас `ios_base`:

```
class ios_base {
public:
  //...
  typedef ... iostate;
  static const iostate
      badbit, // потокът е развален
      eofbit, // вижда се краят на файла
      failbit,// следващата операция няма да се изпълни
      goodbit;// потокът е наред
  //...
};
```

Състояние на потока

– В класа `basic_ios<>` са дефинирани следните методи за манипулиране на състоянието на потока:

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ios: public ios_base {
public:
  ...
  // връща флаговете на състоянието на потока
  iostate rdstate(void) const;
  // установява флаговете на състоянието
  void clear(iostate f=goodbit);
  // добавя f към флаговете на състоянието
  void setstate(iostate f) {
    clear(rdstate()|f);
  }
};
```

Състояние на потока

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ios: public ios_base {
public:
```

```
4     ...
5     operator void* () const;
6   };
```

Операторът, дефиниран в ред 5, е оператор за преобразуване към **void***. Този оператор връща стойност, различна от `NULL`, ако потокът е наред. Този оператор за преобразуване позволява потоците да участват в условни оператори и оператори за цикъл. Например:

```
while(cin){
  cin >> i;
}
```

Операции за вход, дефинирани от потребителя

Нека формата за въвеждане на променливи от типа `point` е `(x,y)`, където `x` и `y` са числа с плаваща точка. Тогава дефиницията на оператор за четене на обекти от класа `point` може да бъде направена по следния начин:

```
1   istream& operator>>(istream& in, point& p) {
2     double x,y;
3     char c;
4     in >> c;
5     if(c!='(') {
6       in.clear(ios_base::badbit);
7       return in;
8     }
9     in >> x >> c;
```

Операции за вход, дефинирани от потребителя

```
10    if(c!=',') {
11      in.clear(ios_base::badbit);
12      return in;
13    }
14    in >> y >> c;
15    if(c!=')') {
16      in.clear(ios_base::badbit);
17      return in;
18    }
19    if(in.good()){
```



```
20      p.set_x(x);
21      p.set_y(y);
22    }
23    return in;
24  };
```

5. Четене на символни низове

Четене на символи

- Операторът **operator>>** е предназначен за форматиран вход — т.е. за четене на обекти от някакъв очакван тип, в някакъв очакван формат.
- Когато предварително не се знае какви типове ще има във входния поток, то от входния поток обикновено се четат символни низове. За тази цел се използва семейството от методи `get()`, дефинирани в `basic_istream<>`.

Четене на символи

```
char c;
cin.get(c);

char buf[100];
cin.get(buf,100);
cin.get(buf,100,'\n');

cin.getline(buf,100);
cin.getline(buf,100,'\n');
```

Четене на символи

- Функциите `in.get(buf,n)` и `in.get(buf,n,term)` прочитат не повече от $n-1$ символа. Тези методи винаги поставят 0 след прочетените в буфера символи.
- Ако при четене с `get` е срещнат завършващият символ, то той остава в потока. Следващият фрагмент е пример за "хитър" безкраен цикъл:

```
char buf[256];
while(cin) {
  cin.get(buf,256);
  cout << buf;
}
```

- Функцията `getline()` се държи аналогично на `get()`, но прочита от `istream` срещнатия завършващ символ.

Четене на символи

- `in.ignore(max,term)` – пропуска следващите символи докато не срещне символа `term` или не прочете `max` символа.
- `in.putback(ch)` – превръща `ch` в следващия непрочетен символ от потока `in`.
- `in.peek()` – връща следващия символ от потока `in`, но не го изтрива от потока.

6. Изключения генерирани от входно/изходните операции

Изключения

- Да се проверява за грешки след всяка входно/изходна операция е неудобно. Поради това е предвидена възможност да "помолите" потока да генерира изключения, когато се променя състоянието му.
- В базовия клас `basic_ios<>` са дефинирани две функции:
  - **void** `exceptions(iostate st)` – установява състоянията, при които трябва да се генерира изключение.
  - `iostate exceptions()` **const** – връща набора от флагове на състоянието, при които се генерира изключение.
- Основната роля на генерирането на изключения при вход/изход е да се обработват малко вероятни изключителни ситуации. Изключенията могат да се използват и за контролиране на входно-изходните операции.

Пример: изключения


```
#include <iostream>
#include <list>

using namespace std;

int
main(int argc,char* argv[]) {
  list<int> result;
  ios_base::iostate oldstate=cin.exceptions();
  cin.exceptions(ios_base::eofbit
                 |ios_base::badbit
                 |ios_base::failbit);
```

Пример: изключения

```
  try {
    for(;;){
      int i;
      cin >> i;
      result.push_back(i);
    }
  } catch(const ios_base::failure& e) {
    cout<<"ios_base::failure␣catched..."<<endl;
  }

  return 0;
}
```

7. Форматиране при изходни операции

Форматиране

- Форматирането на входно/изходните операции се контролира чрез класовете `basic_ios` и `ios_base`.
- За управление на форматирането на входно/изходните операции се използва набор от флагове, определени в `ios_base`.
- Част от флаговете, определящи състоянието на формата са представени в следния фрагмент:

Форматиране

```
class ios_base {
public:
 typedef implementation_dependent fmtflags;
 static const fmtflags
  skipws,     // пропуска разделителите при четене
  boolalpha,  // типа boolean се представят
              // като true и false
  // целочислени типове
  dec,        // десетична система
  hex,        // шестнадесетична система
  oct,        // осмично система
  showbase,   // поставя префикс,
              // обозначаващ системата
```

Форматиране

```
  // числа с плаваща запетая
  scientific,// представяне във вида: d.dddddeddd
  fixed,     // представяне във вида: ddddd.dd
  showpoint, // незначеща нула пред десетичната точка
  showpos,   // явен знак '+' пред положителните числа
  ...;
};
```

Състояние на формата

skipws	пропускане на разделителните символи
boolalpha	представяне на типа boolean
dec	целочислени типове – в каква бройна
hex	система да се извеждат
oct	
showbase	представяне на бройната система
scientific	как се извеждат типовете с плаваща
fixed	точка
showpoint	
showpos	

Форматиране

– За манипулиране на състоянието на формата, в класа `ios_base` са дефинирани следните методи:

```
class ios_base {
public:
  ...
  fmtflags flags() const;
  fmtflags flags(fmtflags f);
  fmtflags setf(fmtflags f){
    return flags(flags()|f);
  }
```

Форматиране

```
  fmtflags setf(fmtflags f, fmtflags mask) {
    return flags((flags()&~mask)|(f&mask));
  }
  void unset(fmtflags mask) {
    flags(flags()&~mask);
  }
};
```

Форматиране

– Стандартната схема за работа с флаговете за форматиране е следната:
  – запомняме състоянието на формата;
  – променяме състоянието на формата и използваме потока;
  – възстановяваме предишното състояние на потока.

```
void foo(void) {
  ios_base::fmtflags old_flags=cout.flags();
  cout.setf(ios_base::oct);
  ...
  cout.flags(old_flags);
};
```

Извеждане на цели числа

- Добавянето на флагове чрез метода `setf()` или чрез побитово "ИЛИ" (`|`) е удобно, само когато дадена характеристика на потока се управлява от един бит.
- Тази схема е неудобна в случаи като определяне на бройната система. В такива случаи състоянието на формата не се определя от един бит.
- Решението на този проблем, което се използва в `<iostream>`, е да се предостави версия на `setf()` с втори "псевдо-аргумент":

```
cout.setf(ios_base::oct, ios_base::basefield);
cout.setf(ios_base::dec, ios_base::basefield);
cout.setf(ios_base::hex, ios_base::basefield);
```

Извеждане на цели числа: пример

```
#include <iostream>
using namespace std;

int main(int argc,char* argv[]) {
  cout.setf(ios_base::showbase);
  cout.setf(ios_base::oct, ios_base::basefield);
  cout<< 1234 << '␣' << 1234 << endl;
  cout.setf(ios_base::dec, ios_base::basefield);
  cout<< 1234 << '␣' << 1234 << endl;
  cout.setf(ios_base::hex, ios_base::basefield);
  cout<< 1234 << '␣' << 1234 << endl;

  return 0;
};
```

Извеждане на цели числа: пример

```
02322 02322
1234 1234
0x4d2 0x4d2
```


Извеждане на числа с плаваща точка

- Извеждането на числа с плаваща запетая се определя от **формата** и **точността.**
- Форматите, които се използват за извеждане на числа с плаваща запетая са:
  - *Универсален* — позволява на потока сам да реши в какъв вид да се представи извежданото число. По подразбиране потоците използват този формат.
  - *Научен* — представя числото като десетична дроб с една цифра преди десетичната точка и показател на степента.
  - *Фиксиран* — точността определя максималният брой цифри след десетичната точка.
- По подразбиране точността е 6 цифри.

Извеждане на числа с плаваща точка: пример

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
  cout<<1234.56789<<'␣'<<1234.5678901<< endl;
  cout.setf(ios_base::scientific,
            ios_base::floatfield);
  cout<<1234.56789<<'␣'<<1234.5678901<<endl;
  cout.setf(ios_base::fixed,
            ios_base::floatfield);
  cout<<1234.56789<<'␣'<<1234.5678901<<endl;
  cout.setf(static_cast<ios_base::fmtflags>(0),
            ios_base::floatfield);
  cout<<1234.56789<<'␣'<<1234.5678901<<endl;
  return 0;
}
```

Извеждане на числа с плаваща точка: пример

```
1234.57 1234.57
1.234568e+03 1.234568e+03
1234.567890 1234.567890
1234.57 1234.57
```

Извеждане на числа с плаваща точка

- За промяна на точността на работа с числа с плаваща запетая се използват следните методи:

```
class ios_base {
public:
  ...
  unsigned precision() const;
  unsigned precision(unsigned n);
  ...
};
```

- Използването на `precision()` влияе на всички входно/изходни операции с потока и действа до следващото използване на метода.

Извеждане на числа с плаваща точка: пример

```
#include <iostream>
using namespace std;
int main(int argc,char* argv[]) {
  cout.precision(12);
  cout<<123456789<<'␣'<<1234.12345<<'␣'
      <<1234.123456789<<endl;
  cout.precision(9);
  cout<<123456789<<'␣'<<1234.12345<<'␣'
      <<1234.123456789<<endl;
  cout.precision(4);
  cout<<123456789<<'␣'<<1234.12345<<'␣'
      <<1234.123456789<<endl;
  return 0;
}
```

Извеждане на числа с плаваща точка: пример

```
123456789 1234.12345 1234.12345679
123456789 1234.12345 1234.12346
123456789 1234 1234
```

Полета за изход


```
#include <iostream>
using namespace std;
int main(int argc,char* argv[]) {
  cout << '(';
  cout.width(5);
  cout << 12 << ')' << endl;;
  cout << '(';
  cout.width(5); cout.fill('#');
  cout << 12 << ')' << endl;;
  cout << '(';
  cout.width(5); cout.fill('#');
  cout << "a" << ')' << endl;;
  cout << '(';
```

Полета за изход

```
  cout.width(0); cout.fill('#');
  cout << "a" << ')' << endl;;
  return 0;
}
```

```
(   12)
(###12)
(####a)
(a)
```

8. Манипулатори

Манипулатори

- Да се променя състоянието на потока посредством флаговете на формата е неудобно.
- Стандартната библиотека предоставя набор от функции и обекти за манипулиране на състоянието на потока — **манипулатори**.

Манипулатори

- Основният начин за използване на манипулатори може да се види от следния пример:

```
cout<<boolalpha<<true<<' '
    <<noboolalpha<<true;
```

което извежда: `true 1`.

– Използват се и манипулатори с аргументи:

```
cout << setprecision(10)
     << 1234.12345678 << endl;
```

което извежда: `1234.123457`. Манипулаторите с аргументи са дефинирани в `<iomanip>`.

`boolalpha`
`showbase`
`showpoint`
`showpos`
`skipws`
	`noboolalpha`
	`noshowbase`
	`noshowpoint`
	`noshowpos`
	`noskipws`
		представяне на типа boolean
		представяне на бройната система
		
		пропускане на разделителните символи

`dec`
`hex`
`oct`
		целочислени типове – в каква бройна система да се извеждат

`fixed` `scientific`		как се извеждат типовете с плаваща точка
`setprecision(n)`		точност на извеждането
`setw(int n)` `setfill(int c)`		ширина на полето за извеждане символ за запълване на полето

9. Файлови потоци

Файлови потоци

– Потоците за работа с файлове са дефинирани в `<fstream>`.

– Потокът за писане във файл е `basic_ofstream`.

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ofstream: public basic_ostream<Ch,Tr> {
public:
  explicit basic_ofstream(const char* p,
                  openmode m=out|trunc);
  bool is_open() const;
  void open(const char* p,
            openmode m=out|trunc);
  void close();
  ...
};
```


Файлови потоци

```
class ios_base {
public:
  typedef implementation_dependent1 openmode;
  static const openmode app,
    ate, binary, in, out, trunc;
  ...
};
```

`in`	отваряне за четене
`out`	отваряне за запис
`app`	запис на данните в края на файла; предизвиква отваряне на файла в режим `out`
`ate`	запис на данните в края на файла
`trunc`	унищожава предишното съдържание на файла

Файлови потоци

```
#include <iostream>
#include <fstream>
#include <cstdlib>
void error(const char* p, const char* p2="") {
  std::cerr << p << ' ' << p2 << std::endl;
  std::exit(1);
}

int main(int argc,char* argv[]) {
  if(argc!=3)
    error("bad number of arguments...");
  std::ifstream from(argv[1]);
  if(!from)
    error("bad input file", argv[1]);
  std::ofstream to(argv[2]);
  if(!to)
    error("bad output file:", argv[2]);
  char ch;
  while(from.get(ch))
    to.put(ch);
  if(!from.eof() || !to)
    error("something strange...");
```

```
22    return 0;
23  }
```