Кратък обзор на езика за програмиране C++

Любомир Чорбаджиев
Технологическо училище "Електронни системи"
Технически университет, София
lchorbadjiev@elsys-bg.org
Revision : 1.34

$Date: 2006/02/19 08:10:10 $

Съдържание

Въведение

1. История на създаването на езика

2. Променливи и аритметика

2.1. Вградени типове данни

Езикът C++ наследява дефинираните в C базови типове — **char**, **int**, **float**, **double**. Към тях е добавен нов тип **bool** — булев тип, чиито стойности могат да бъдат **true** или **false**.

Допълнително в езика има на разположение модификатори, също наследени от C, чрез които могат да променят свойствата на някои от типовете. Такива са **short** и **long**, които се използват за промяна на размера на типа. Основно тези модификатори се прилагат към цели (**int**):

```
1 short int index;
2 long  int counter;
```

Ключовата дума **int** може да бъде пропусната и горната дефиниция може да се запише по следния напълно еквивалентен начин:

```
1 short index;
2 long  counter;
```

Модификаторът **long** може да се прилага и към числа с плаваща запетая **double**, т.е. може да се използва типът **long double** — число с плаваща запетая с двойна точност.

Модификаторите **signed** и **unsigned** могат да се прилагат към целите числа (**int**) и към символния тип (**char**). Те определят дали дефинираната променлива ще бъде с или без знак. Например следният фрагмент

```
1 signed char schar;
2 unsigned char uchar;
```

дефинира две променливи от символен тип. Това означава, че стойностите и на двете променливи са еднобайтови (8 бита). Диапазона на стойностите, които може да приема `schar` (символна променлива със знак) е от `-128` до `127`, а за променливата `uchar` (символна променлива без знак) — от `0` до `255`.

2.2. Променливи

Начинът по който се дефинират променливи в C++ е аналогичен на дефинирането на променливи в C. Следният фрагмент

```
int counter;
double sum;
```

дефинира две променливи – променливата `counter` от типа **int** и променливата `sum` от типа **double**.

Всяка променлива може да бъде инициализирана при нейното дефиниране:

```
int i=0;
double eps=1e-6;
```

В `C` променливите, които се използват в дадена функция, трябва задължително да бъдат дефинирани в началото на функцията. В `C++` такова изискване няма. Повече от това — стилът на програмиране на `C++` препоръчва:

- Променливите да не се дефинират преди да е възникнала необходимост от тях.
- Всяка променлива да се инициализира при дефинирането ѝ.

2.3. Константи

Към дефиницията на всяка променлива може да се прилага модификаторът **const**, който показва, че стойността на променливата няма да се променя:

```
const double e =2.7182818284590452354;
const double pi=3.14159265358979323846;
const char[] message="warning:␣";
```

Това е предпочитаният в `C++` начин за дефиниране на константи.

2.4. Аритметични, релационни и логически оператори

Аритметичните оператори в `C++` са същите, като в `C`:

Оператор	Описание
+	бинарен оператор за събиране и унарен оператор "плюс".
-	бинарен оператор за изваждане и унарен оператор "минус".
*	бинарен оператор за умножение.
/	бинарен оператор за деление.
%	бинарен оператор за намиране на остатъка от целочислено деление.

Релационните оператори, наричани още оператори за отношение са:

Оператор	Описание
<	бинарен оператор “по-малко”.
<=	бинарен оператор “по-малко или равно”.
>	бинарен оператор “по-голямо”.
>=	бинарен оператор “по-голямо или равно”.
==	бинарен оператор “равно”.
!=	бинарен оператор “неравно (различно)”.

Логическите оператори са:

Оператор	Описание
&&	бинарен оператор “логическо И”.
||	бинарен оператор “логическо ИЛИ”.
!	бинарен оператор “логическо НЕ”.

2.5. Преобразуване на типове

2.5.1. Неявно преобразуване на типовете

В езика C има набор от преобразувания между типовете на езика, които се извършват автоматично. Да разгледаме следния фрагмент:

```
1 int a=1;
2 double f;
3 f=a;
```

В ред 3 се извършва неявно преобразуване на целочислената променлива a към стойност от типа **double**.

Целочислените типове и типовете с плаваща запетая могат да свободно да бъдат смесвани в аритметични оператори и оператори за присвояване. Правилата за неявно преобразуване на вградените типовете в C и в C++ са сходни. Общото правило е, че когато е възможно, автоматичното преобразуване на типове се извършва без загуба на точност.

За съжаление обаче в C и в C++ автоматично се извършват и преобразувания на типове, които очевидно водят до загуба на стойността на променливата. Например:

```
1 void function(double d) {
2   char ch=d;
3 }
```

В ред 2 се извършва неявно преобразуване на **double** към **char**, което в повечето случаи би довело до недефинирани резултати.

Такива опасни неявни преобразувания на типове обикновено трябва да се избягват. Повечето съвременни компилатори са в състояние да дават предупреждения, когато срещнат подобно опасно преобразуване на типове.

2.5.2. Явно преобразуване на типа

Има редица ситуации, при които се налага изрично една променлива да бъде преобразувана към друг тип. Да разгледаме следния фрагмент:

```
1  int a=2;
2  int b=3;
3  double d=b/a;
```

Делението, което се извършва в ред 3 е целочислено, резултатът от което се присвоява на променливата `d`. Следователно стойността, която получава променливата `d` е `1`.

Ако искаме делението да бъде не целочислено, а делене на числа с плаваща запетая, то е необходимо явно да преобразуваме някой от операндите, участващи в делението:

```
1  int a=2;
2  int b=3;
3  double d=static_cast<double>(b)/a;
```

Тогава резултатът от деленето в ред 3 е `d=1.5`.

В C++ за явно преобразуване се използва оператора **static_cast**. Синтаксисът му е следният:

```
static_cast<тип>(аргумент)
```

където *тип* е типът, към който трябва да се преобразува аргументът, а *аргумент* е променливата която трябва да бъде преобразувана. Например:

```
1 char a='a';
2 double d=2.0;
3 d=static_cast<double>(a);
4 a=static_cast<char>(d);
```

в ред 3 променливата `a` от тип **char** се преобразува към тип **double**. В ред 4 променливата `d` от тип **double** се преобразува към тип **char**.

3. Указатели и масиви. Препратки (*reference*)

3.1. Указатели

В `C` и `C++` има специален тип променливи, в които могат да се съхраняват адреси на области от паметта. Този тип променливи са *указателите*. Ако с `T` означим даден тип, то `T*` ще бъде типът на *указател към* `T`. С други думи — променливите от тип `T*` съдържат адрес на обект от типа `T`. Например:

```
1 T* ps;
2 int* pi;
```

В ред 1 е дефинирана променлива `ps`, която е указател към променливи от типа `T`. В ред 2 е дефинирана променлива `pi`, която е указател към променливи от типа `int`.

За да се получи адресът на дадена променлива се използва унарния оператор `&`. Например:

```
1 int a=42;
2 int* pa;
3 pa=&a;
```

В ред 2 е дефинирана променливата `pa`, която е от типа *указател към* `int`. В следващия ред 3 на `pa` е присвоена стойност, която е адресът на променливата `a`, т.е. указателят `pa` ще сочи към променливата `a`.

Основна операция, която се извършва върху указателите, е да се получи обектът, към който сочи указателят. За тази цел се използва унарният оператор `*`. Да разгледаме следния фрагмент:

```
1 int a=42;
2 int* pa=&a;
3 int b=*pa;
4 *pa+=42;
```

В ред 3 стойността на променливата, към която сочи `pa`, се присвоява на `b`, т.е. резултатът от операцията е `b=42`. В ред 4 стойността на променливата, към която сочи указателя `pa`, се увеличава с `42`, т.е. резултатът от операцията е, че стойността на променливата `a` се е увеличила с 42.

3.2. Препратки (references)

Препратката (reference) е алтернативно име на обекта. Поради това често препратките се наричат *псевдоними*. За даден тип `T`, типът на препратка към обекти от този тип се обозначава с `T&`. Например:

```
1 int i=1;
2 int& r=i;
3 r=2;
```

В ред 2 е дефинирана препратка към променливата `i`. Когато в ред 3 на препратката `r` се присвоява нова стойност, тази операция в действителност се извършва с променливата `i`, към която препраща `r`.

При дефиниране на препратка, тя задължително трябва да бъде инициализира с обект от съответния тип. Да разгледаме следния фрагмент:

```
1 int& r1;     //грешка!
2 int& r2=10; //грешка!
```

В ред 1 е допусната грешка, тъй като липсва инициализация на препратката `r`. В ред 2 инициализацията е неправилна, тъй като препратката не е инициализирана с променлива от типа **int**.

Веднъж дефинирана препратката не може да се пренасочи към друг обект. Точно поради тази причина при дефинирането ѝ е задължително тя да бъде инициализирана. В следващия фрагмент:

```
1 int i=1;
2 int& r=i;
3 int i2=2;
4 r=i2;
```

операцията, която се изпълнява в ред 4 не пренасочва препратката към `i2`, а присвоява нова стойност на `i`.

Всички операции, които се извършват върху препратката, в действителност се извършват върху обекта, към който е насочена препратката. Така например:

```
1 int i=0;
2 int& r=i;
3 r++;
4 int* p=&r;
```

В ред 3 операцията се извършва върху променливата `i` и тя се увеличава с единица. Взимането на адрес от препратка в ред 3 всъщност води до взимането на адреса на променливата `i`.

Препратките най-често се използват като формални аргументи на функции в случай, че функцията трябва да е в състояние да променя стойността на предадения обект. Например:

```
1 void plus2(int& v) {
```

```
2   v+=2;
3 }
4 ...
5 int x=1;
6 plus2(x);
```

Функцията, дефинирана в ред 1 има за аргумент препратка. Това означава, че при предаване на действителния аргумент ще се създаде препратка към него, и в тялото на функцията ще се работи с препратка към действителния аргумент. Следователно след изпълнението на ред 6, стойността на променливата `x` ще бъде 3.

3.3. Масиви

3.3.1. Дефиниране на масиви

Масивите са производни типове, които представляват колекция от стойности с еднакъв тип. За даден тип `T`, типът `T[size]` представлява типът *масив от* `size` *елемента от типа* `T`. Например:

```
1 T tarray[10];
2 int iarray[42];
```

В ред 1 е дефиниран масив от `10` елемента от типа `T`. В ред 2 е дефиниран масив от `42` елемента от типа `int`.

Изразът `size`, който определя размера на масива, трябва да бъде константен израз. Елементите на масива се индексират с числата от `0` до `size-1`. Например:

```
1 int iarr[42];
2 iarr[0]=1;
3 iarr[41]=42;
4 iarr[42]=43;   // грешка!
```

Индексът на първият елемент на масива `iarr` е нула (виж ред 2), индексът на последния 42-ри елемент на масива е `41` (виж ред 3). Използването на размера на масива като индекс е грешка (виж ред 4).

Многомерните масиви в `C` и `C++` се дефинират като масиви от масиви. Например, ако е необходим двумерен масив, то декларацията е следната:

```
1 int d2[5][10];
```

3.3.2. Инициализация на масив

Началните стойности на елементите на даден масив, могат да се присвоят като се използва списък от стойности.

```
1 int iarr[4]={0,1,2,3};
2 char carr[6]={'H','e','l','l','o','\0'};
```

Когато се използва инициализация на елементите на масива, то размерът на масива може да се пропусне. В такъв случай размерът на масива се определя от броя на елементите в инициализиращия списък. Например:

```
1 int v[]={1,2,3,4};
2 char ac[]={'a','b','c'};
```

Дефинираният в ред 1 масив има 4 елемента, а дефинираният в ред 2 — 3 елемента.

Обърнете внимание, че когато размерът на масива е указан явно, инициализирането на масива със списък, съдържащ повече елементи е грешка.

```
1 int v[2]={1,2,3}; // Грешка!
```

Ако в списъка за инициализация на масива броят на елементите е по-малък от размера на масива, то на неинициализираните елементи се присвоява стойност по подразбиране. Следната инициализация

```
1 int v[4]={1,2};
```

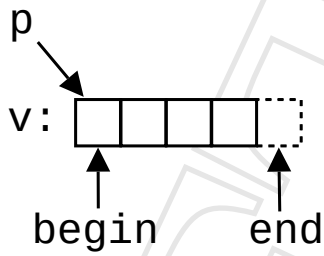
е еквивалентна на

```
1 int v[]={1,2,0,0};
```

3.3.3. Указатели и масиви

Указателите и масивите са тясно свързани. Името на масива може да се използва като указател, сочещ към първият елемент на масива. Нека разгледаме следния фрагмент:

```
1 int v[]={1,2,3,4};
2 int* p=v;
3 int* begin=&v[0];
4 int* end=&v[4];
```



Фиг. 1. Указатели и масиви

В ред 1 е дефиниран масивът `v`, който има четири елемента. Името на масива `v` може да се разглежда като указател, сочещ началото (първия елемент) на масива, т.е. дефинирания в ред 2 указател `p` сочи към първия елемент на масива `v`. Адресът на първия елемент на масива може да бъде получен и като се използва унарния оператор `&` по следния начин — `&v[0]` (виж ред 3).

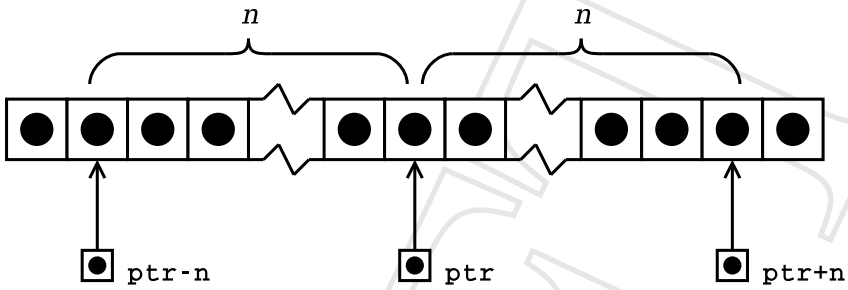
3.4. Аритметика с указатели

Когато целочислен израз се добави към или извади от указател, резултатът е указател от същия тип. Ако указателят сочи към елемент на масив и масивът е достатъчно голям, то резултатът ще бъде указател към елемент от същия масив, отместен със съответния брой елементи. С други думи, ако указателят `ptr` сочи към i-тия елемент на даден масив, а n е цяло число, то изразът `ptr+n` ще сочи към $(i + n)$-тия елемент на масива, а `ptr-n` ще сочи към $(i - n)$-тия елемент, в случай, че такива елементи съществуват.

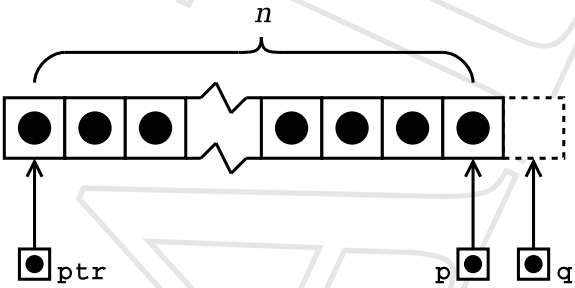
Дефиницията на езика гарантира, че стойността на указател, насочен с едно след последния елемент на масива е смислена. Ако указателят `p` сочи към последния елемент на даден масив (виж фиг. 3), то `(p+1)` е указател насочен с едно след последния елемент на масива. Ако указателят `q` сочи с едно след последния елемент на масива (виж отново фиг. 3), то `(q-1)` сочи към последния елемент на масива.

Трябва да се отбележи, че резултатът от операциите събиране и изваждане на указател с цяло число е добре дефиниран само когато и указателят и резултатът от операцията са насочени към елементи на един и същ масив или с едно след последния елемент на масива. Например:

```
1 int v[10];
```



Фиг. 2. Аритметика с указатели: събиране и изваждане на цяло число и указател

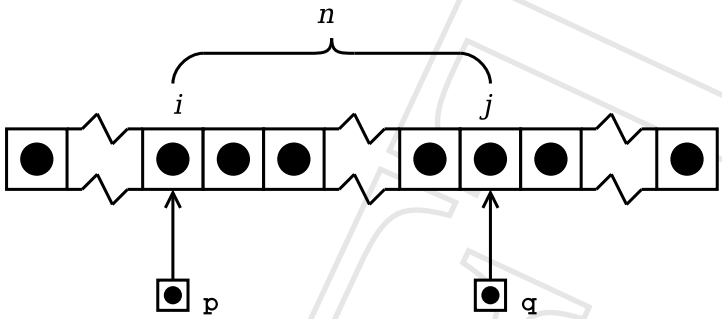


Фиг. 3. Аритметика с указатели: указател насочен с едно след последния елемент на масива

```
2 int* p1=v+11;
3  int* p2=v+12;
4 int* p3=v-1;
```

Операцията в ред 2 е коректна, тъй като резултатът ще бъде указател насочен с едно след последния елемент на масива `v`. Резултатът от операциите в ред 3 и 4 не е дефиниран.

Изваждането на един указател от друг указател е дефинирано само в случай, че двата указателя сочат към елементи на един и същ масив. Резултатът от изваждането на два указателя е равен на броя елементи на масива, разположени между указателите. Ако указателят `p` сочи към i-тия елемент от масива (виж фиг 4), а указателят `q` сочи към j-тия елемент, то разликата между двата указателя `(q-p)` ще бъде равна на $j - i = n$. Резултатът от изваждането на двата указателя е число със знак, т.е. резултатът от `(p-q)` е $i - j = -n$.



Фиг. 4. Аритметика с указатели: изваждане на указатели

Да разгледаме следния фрагмент:

```
1 int v1[10], v2[10];
2 int i1=&v1[5]-&v1[3];
3 int i2=&v1[5]-&v2[3];// резултатът не е дефиниран
```

В резултат на операцията в ред 2 променливата `i1` получава стойност 2. Операцията в ред 3 не е дефиниран, тъй като двата указателя не сочат към елементи на един и същ масив.

4. Функции

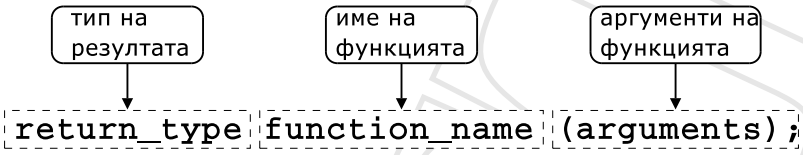
Всяка програма на `C` или `C++` има дефинирана поне една функция — `main`-функция. Всички програми, с изключение на най-тривиалните, дефинират допълнителни функции. Функциите служат за групиране на често използван код, като позволяват групирания код да се използват лесно и многократно.

4.1. Деклариране на функция

Преди да бъде използвана една функция, тя трябва да бъде декларирана. Декларацията казва на компилатора какво е името на функцията, какъв е типът на резултата, връщан от функцията и какви са параметрите на функцията. Има два начина да се декларира една функция:

- Да се дефинира цялата функция преди да бъде използвана.
- Да се дефинира прототипа на функцията, който дава на компилатора необходимата информация.

Дефинирането на прототипа на една функция има следния синтаксис:



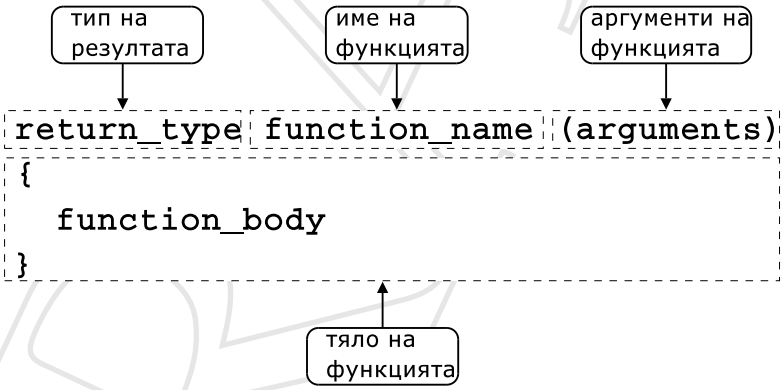
Следния пример съдържа прототипите на няколко функции:

```
1 double distance(double x1, double y1,
2                 double x2, double y2);
3 double area(double r);
```

В ред 1 е дефиниран прототипът на функция с име `distance`, която връща резултат от тип **double** и има четири параметъра от типа **double**. В ред 3 е дефиниран прототипът на функция с име `area`, която връща резултат от тип **double** и има един параметър от тип **double**.

4.2. Дефиниране на функции

Дефиницията на една функция има следния синтаксис:

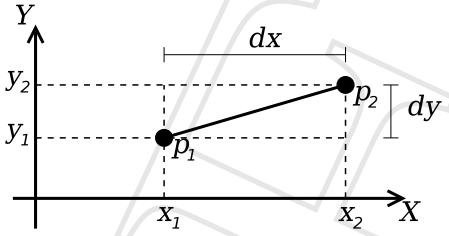


В следния фрагмент е дефинирана функция, която изчислява площта на окръжност, ако е даден радиусът на окръжността.

```
1 const double PI=3.141592653589793;
2
3 double area(double r) {
4   return PI*r*r;
5 }
```

Радиусът на окръжността се предава като параметър на функцията. В тялото на функцията се извършват изчисленията и площта на окръжността се връща като резултат от функцията.

Да разгледаме още един пример. Нека са дадени две точки в равнината — p_1 и p_2. Нека в дадена декартова координатна система координатите



Фиг. 5. Разстояние между две точки в равнината

на точката p_1 са (x_1, y_1), а координатите на точката p_2 са (x_2, y_2). Тогава, разстоянието между двете точки може да се изчисли като се използва теоремата на Питагор (вж. фиг. 5):

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

В следващия фрагмент е дефинирана функция, която изчислява разстоянието между две точки в равнината.

```
1 #include <cmath>
2 using namespace std;
3
4 double distance(double x1, double y1,
5                 double x2, double y2) {
6   double dx=x2-x1;
7   double dy=y2-y1;
8   return sqrt(dx*dx+dy*dy);
9 }
```

Като параметри на функцията се предават координатите на точките в равнината. Като резултат от функцията се връща разстоянието между точките. За намиране на разстоянието в ред 8 се използва функцията `sqrt`, която изчислява корен квадратен. Тази функция е декларирана в заглавния файл `<cmath>`, който е част от стандартната библиотека[1].

[1]Всички типове и функции, дефинирани в стандартната библиотека се намират в пространството от имена `std`. За да се включи това пространство от имена се използва

4.3. Предаване на аргументи

В C параметрите на функцията се предават по стойност. С други думи — при извикване на дадена функция, параметрите, които се предават се копират и в тялото на функцията се използват копията. Нека разгледаме следния пример:

```
1 void plus2(int x) {
2   x+=2;
3 }
4 int main() {
5   int counter=0;
6   plus2(counter);
7   ...
8 }
```

Тъй като параметрите на функциите се предават по стойност, операцията в ред 2 се извършва върху копие на предадената стойност. Това означава, че извикването на функцията `plus2` в ред 6 няма да доведе до промяна на стойността на променливата `counter`.

Ако е необходимо функцията да променя стойността на аргументите си, то може да се използват указатели, т.е. като параметри на функцията се предават не самите променливи, а указатели към тях. Да разгледаме следния фрагмент:

```
1 void plus2(int* px) {
2   *px+=2;
3 }
4 int main() {
5   int counter=0;
6   plus2(&counter);
7   ...
8 }
```

Съгласно дефиницията на функцията `plus2` в ред 1 като параметър се предава указател към **int**. В тялото на функцията, в ред 2, операцията се извършва върху обекта, към който сочи предаденият указател. Това означава, че когато в ред 6 се извика функцията `plus2` и като параметър се предаде указател към променливата `counter`, при изпълнение на функцията ще се промени стойността на променливата към която сочи указателят, т.е. променливата `counter`.

конструкцията **using namespace** `std;` (вж. ред 2). Повече за пространството от имена виж в раздел 7

В езика C този подход е единственият, който позволява една функция да променя параметрите си. В C++ има една допълнителна възможност — аргументите на функцията да се предават не по стойност, а като препратки. Да разгледаме следния фрагмент:

```
1 void plus2(int& x) {
2   x+=2;
3 }
4 int main() {
5   int counter=0;
6   plus2(counter);
7   ...
8 }
```

При дефиницията на функцията `plus2` в ред 1 е указано, че параметърът `x` на функцията се предава като препратка. Това означава, че когато в ред 6 функцията се извиква с действителен аргумент променливата `counter`, в тялото на функцията се предава препратка към променливата `counter`. Операциите в тялото на функцията се извършват върху препратката към действителния аргумент, което означава, че при промяна на формалния аргумент в ред 2, се променя предадения действителен аргумент на функцията — променливата `counter`.

4.4. Предефиниране (*overloading*) на функции

В C++ е допустимо в една и съща програма да се използват няколко функции, които имат различни аргументи, но едно и също име. Когато се използва едно и също име за дефиниране на няколко функции се говори за *предефиниране* на функции[2].

Нека разгледаме следната ситуация. Трябва да се дефинира функция, която събира две числа и връща резултата. Тази функция трябва да може да се използва, както с аргументи от типа **int**, така и с аргументи от типа **double**. Да разгледаме следния фрагмент:

```
1 int add(int x, int y) {
2   return x+y;
3 }
4 double add(double x, double y) {
```

[2] В литературата на български език няма единна терминология за обозначаване на това свойство на C++. Други често използвани термини за обозначаване на предефинирането на функции (function overloading) са: *функции с много имена*, *препокриване на функции*. Все пак, като че ли най-често се използва *предефиниране на функции*.

```
5     return x+y;
6   }
7   int main() {
8     int a=1,b=2;
9     double x=1.0,y=2.0;
10
11    int si=add(a,b);
12    double sd=add(x,y);
13    return 0;
14  }
```

При използването на функция с име `add` компилаторът преценява коя точно дефиниция на функцията трябва да използва като търси съвпадение на типовете на действителните аргументи с типовете от дефиницията на функцията. Тъй като в ред 11 функцията `add` се извиква с цели аргументи, компилаторът ще използва дефиницията от ред 1. В ред 12 ще се използва дефиницията от ред 4, тъй като аргументите са от типа **double**.

4.5. Аргументи по подразбиране

При дефиниране на функции в `C++` на параметрите на функцията могат да се задават стойности по подразбиране. Нека разгледаме следната функция:

```
1   void increment(int& count, int step) {
2     count+=step;
3   }
4   int main() {
5     int c=10;
6     increment(c,1);
7     increment(c,10);
8     //...
9     return 0;
10  }
```

Функцията `increment` има два аргумента. Първият аргумент се предава като препратка. При извикване на функцията стойността на първият аргумент се увеличава със стойността, предадена като втори аргумент. В представения пример, при извикване на функцията в ред 6 променливата `c` се увеличава с единица и става равна на 11; в ред 7 се увеличава с десет и става равна на 21.

Да приемем, че най-често тази функция ще се вика с втори аргумент равен на 1. Тогава дефиницията на функцията може да се промени по следния начин:

```
1  void increment(int& count, int step=1) {
2    count+=step;
3  }
4  int main() {
5    int c=10;
6    increment(c);
7    increment(c,10);
8    //...
9    return 0;
10 }
```

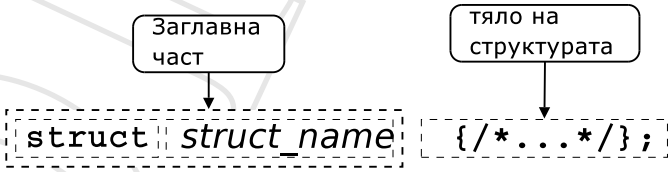
Обърнете внимание, че в ред 1 на втория аргумент на функцията е зададена стойност по подразбиране, равна на 1. Това позволява функцията да се вика като се пропускат аргументите, които имат стойност по подразбиране — виж ред 6. В такъв случай стойността на пропуснатия аргумент ще бъде равна на стойността по подразбиране.

5. Структури

5.1. Дефиниране на структури

Структурата представлява съвкупност от една или повече променливи, които могат да от различни типове. Дефиницията на структура има следния синтаксис:

- Заглавна част, която се състои от ключовата дума **struct** последвана от името на структурата.
- Тяло, в което се описват членовете на структурата. Тялото на дефиницията е оградено от фигурни скоби и *задължително* трябва да бъде последвано от точка и запетая ‘;’.



В следващия фрагмент е дефинирана структурата `person`:

```
struct person {
  char* name;
  int age;
};
```

Идеята на тази структура е да съдържа данни за хора, необходими за работата на програма, която има нужда от името и възрастта на човека. Поради това структурата е дефинирана с два члена — член `name` от типа `char*`, който съдържа името и член `age` от типа `age`, който държи възрастта.

След дефинирането името на структурата се превръща в име на тип и може да се използва за дефиниране на променливи. В следващия фрагмент е дефинирана променлива `person` от типа на дефинираната по-горе структура `person`[3]:

```
person somebody;
```

При дефинирането на променлива от типа на дефинирана структура, тя може да се инициализира. За целта след дефиницията на променливата трябва да се постави списък с инициализатори, които съответстват на членовете на структурата. Например, ако искаме да инициализираме променлива от типа `person`, то списъкът от инициализатори трябва да съдържа първо стойност за първия член на структурата, т.е. за името, а след това стойност за втория член на структурата, т.е. за възрастта:

```
person anybody={"pesho",18};
```

В представения пример членът `name` на променливата `anybody` получава начална стойност `"pesho"`, а членът `age` — стойност `18`.

Достъпът до елементите на структурата се осъществява с използването на оператора `".`— оператор за достъп до член. Този оператор свързва името на структурата с името на члена. Например:

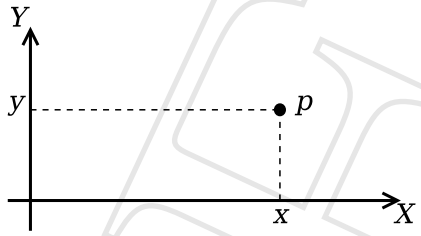
```
person somebody;
somebody.name="ivan";
somebody.age=16;
```

[3]Обърнете внимание, че в `C` за да се дефинира променлива която е от типа на дефинираната структура трябва да се използва конструкцията **`struct person somebody;`** Тази конструкция е валидна и `C++`.

5.2. Примери за използване на структури

5.2.1. Точка в равнината

Нека дефинираме структура, която изобразява точка в равнината, като използва нейните декартови координати (вж. фиг. 6). Структурата

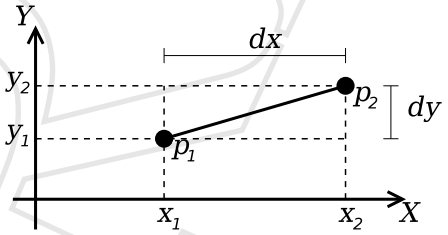


Фиг. 6. Декартови координати на точка в равнината

може да бъде дефинирана по следния начин:

```
1 struct point {
2   double x;
3   double y;
4 };
```

Нека сега дефинираме няколко функции, които използват тази структура. Първата функция, която ще разгледаме е функция, която изчислява разстоянието между две точки в равнината като използва Питагоровата теорема (вж. фиг. 7). Дефиницията на функцията е представена



Фиг. 7. Разстояние между две точки в равнината

в следния фрагмент[4]:

[4]Сравнете така дефинираната функция `distance` с дефиницията на същата функция в раздел 4.2.

```
#include <cmath>
using namespace std;
double distance(point p1, point p2) {
  double dx=p1.x-p2.x;
  double dy=p1.y-p2.y;
  return sqrt(dx*dx+dy*dy);
}
```

Нека разгледаме още две операции, които често се изпълняват върху точки в равнината — операциите събиране и изваждане. Ако имаме две точки p_1 и p_2 в равнината с декартови координати (x_1, y_1) и (x_2, y_2) съответно, то сума на точките $p_1 + p_2$ се нарича точка от равнината $P = (X, Y)$, чиито координати се изчисляват по следния начин:

$$X = x_1 + x_2;\ Y = y_1 + y_2. \tag{1}$$

Функцията, реализираща тази операция, може да се дефинира по следния начин:

```
point add(point p1, point p2) {
  point result={p1.x+p2.x, p1.y+p2.y};
  return result;
}
```

Аналогично се дефинира изваждането на точки от равнината. Разликата на две точки в равнината $p_1 - p_2$ е отново точка $P = (X, Y)$, чиито координати се изчисляват по следния начин:

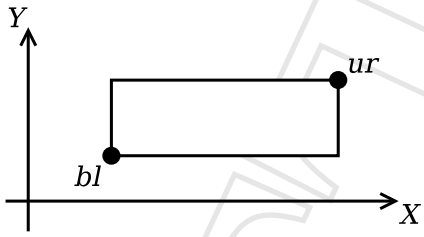
$$X = x_1 - x_2;\ Y = y_1 - y_2. \tag{2}$$

Дефиницията на функцията, която реализира операцията изваждане на точки от равнината, е представена в следния фрагмент:

```
point sub(point p1, point p2) {
  point result={p1.x-p2.x, p1.y-p2.y};
  return result;
}
```

5.2.2. Правоъгълник в равнината

Нека разгледаме правоъгълник в равнината, чиито страни са успоредни на координатните оси. За определянето на такъв правоъгълник е напълно достатъчно да се укажат два срещуположни негови върха —



Фиг. 8. Правоъгълник със страни успоредни на координатните оси

например долния ляв (bl — bottom left) и горния десен (ur — upper right). Дефиницията на структурата `rect`, която описва такива правоъгълници, е представена в следния фрагмент:

```
5  struct rect {
6    point bl;
7    point ur;
8  };
```

Обърнете внимание, че като членове на структурата `rect` се използват променливи, чийто тип е дефинираната по-горе структура `point`.

Нека разгледаме няколко функции, които работят със структурите `rect` и `point`. Първо, нека дефинираме функция `cononical_rect`, целта на която е да върне правоъгълник, в който членовете има коректни стойности — x-координатата на долния ляв ъгъл трябва да е по-малка от x-координатата на горния десен ъгъл; y-координатата на долния ляв ъгъл трябва да е по-малка от y-координата на горния десен ъгъл. Функцията `cononical_rect` обезпечава спазването на това изискване.

```
10 inline double min(double x, double y) {
11   return x<y?x:y;
12 }
13 inline double max(double x, double y) {
14   return x>y?x:y;
15 }
16 rect canonical_rect(rect r) {
17   point bl={min(r.bl.x,r.ur.x),min(r.bl.y,r.ur.y)};
18   point ur={max(r.bl.x,r.ur.x),max(r.bl.y,r.ur.y)};
19   rect result={bl,ur};
20   return result;
21 }
```

Обърнете внимание, че функциите `min` и `max` са дефинирани като **inline** функции.

Нека разгледаме още една функция, която работи със структурите `rect` и `point` — функцията `contains`. Тази функция получава като аргументи правоъгълник `r` и точка `p` и проверява дали точката `p` се намира в правоъгълника `r`. Ако точката се намира в правоъгълника, функцията връща стойност **true**, иначе — връща **false**.

```
23 bool contains(rect r, point p) {
24   return p.x>=r.bl.x && p.x<=r.ur.x
25     && p.y>=r.bl.y && p.y <= r.ur.y;
26 }
```

6. Класове

В езика C++ има няколко начина за дефиниране на типове от потребителя. Едната възможност е да се използват разгледаните вече структури **struct**. Другата възможност е да се използват класове.

Механизмът на класовете в C++ разполага с изключително богати възможности, което позволява дефинираните от потребителя типове да бъдат точно толкова мощни и изразителни, колкото и вградените в езика типове.

6.1. Дефиниция на клас

Дефиницията на клас в езика C++ се състои от две части — *заглавна част* и *тяло*. *Заглавната част на класа* се състои от ключовата дума **class**, последвана от името на класа. *Тялото на класа* се разполага след заглавната част и е затворено във фигурни скоби.

Дефиницията на класа трябва да бъде последвана от точка и запетая или от списък от декларации. Например:

```
class Point {/*...*/};
class Rectangle {/*...*/} r1, r2;
```

В тялото на класа се дефинира списъкът от членове на класа и нивото на достъп до тях. Класовете имат два вида членове: *член-променливи* и *член-функции*.

6.2. Член-променливи

Дефинирането на член-променливите на класа е аналогично на дефинирането на променливи. За да стане една променлива член на класа, то тя трябва да бъде дефинирана в тялото на класа.

Например, променливите `x_` и `y_`, дефинирани в ред 2 и 3, стават член-променливи на класа `Point`, защото са дефинирани в неговото тяло.

```
1 class Point {
2   double x_;
3   double y_;
4 };
```

Променливите `bl_` и `ur_` от следващия фрагмент са член-променливи на класа `Rectangle`.

```
1 class Rectangle {
2   Point bl_, ur_;
3 };
```

Член-променливите не могат да бъдат инициализирани при тяхното дефиниране.[5] Следователно, следният фрагмент съдържа грешка в ред 2:

```
1 class Foo {
2   int bar_=42;
3 };
```

Причината за тази разлика е, че при дефинирането на член-променлива не се заделя памет. Заделянето на памет и инициализирането на член-променливите се извършва едва при създаването на обект от дадения клас. Поради това за инициализиране на член-променливите на обектите от даден клас се грижи специализирана член-функция — *конструктор* — която се вика автоматично при създаването на всеки обект.

[5] Това правило има изключение. Ако член-променливата е статична и константна, то тя може да бъде инициализирана още при нейното дефиниране.

6.3. Член-функции

Член-функциите реализират множеството от операции, които могат да се извършват върху обектите от даден клас. Идеологията на обектно-ориентираното програмиране налага правилото, че всички операции върху член-променливите трябва да се извършват от член-функциите на класа.

За да стане една функция член на класа, тя трябва да бъде декларирана в тялото на класа. Например функцията `set_x()`, декларирана в следващия фрагмент, е член-функция на класа `Point`.

```
class Point {
//...
  void set_x(double x);
};
```

Член-функциите могат да се дефинират в тялото на класа.

```
class Point {
  double x_, y_;
public:
  void set_x(double x){x_=x;}
};
```

6.4. Модификатори за достъп до членовете на класа

Капсулирането (скриването на информацията) е механизъм който предпазва вътрешното представяне на данните. В обектно-ориентирането програмиране капсулирането е основна техника, която се използва за разпределяне на отговорностите между различните части на програмата.

Класовете в C++ имат силно развит механизъм за скриване на информацията. В основата му са спецификаторите за достъп — **public**, **private** и **protected**. Чрез спецификаторите за достъп тялото на класа се разделя на секции — *публична* (**public**), *скрита* (**private**) и *защитена* (**protected**). В зависимост от това в коя секция е дефиниран даден член на класа, той се превръща съответно в *публичен*, *скрит* или *защитен*.

Публичните членове на класа са достъпни от всички точки на програмата. *Скритите членове* на класа са достъпни само в член-функциите на класа и в *приятелите* на класа. *Защитените членове* се държат като публични за членовете на производните класове и като скрити за всички останали точки на програмата.

Като пример нека разгледаме класа `Point`, дефиниран по следния начин:

```
1 class Point {
2   double x_, y_;
3 public:
4   void set_x(double x){x_=x;}
5 };
6 Point p1, p2;
7 p1.set_x(10.0);
8 p2.x_=10.0;// грешка
```

В ред 7 променяме състоянието на обекта `p1` като се обръщаме към публичен член на класа — член-функцията `set_x()`. Като резултат стойността на скритата член-променлива `p1.x_` става 10. Ако се опитаме обаче директно да променим стойността на скрита член-променлива (вж. ред 8), то компилаторът ще даде съобщение за грешка.

6.5. Обекти

Дефиницията на класа може да се разглежда като шаблон, по който се създават обекти. Дефинирането на клас създава нов тип в областта на видимост, в която е направена дефиницията. За да се дефинира обект от даден клас, трябва да се дефинира променлива от съответния тип.

```
Point p;

Point* ptr=&p;
Point& ref=p;
```

При дефиниране на променлива от типа на даден клас се създава обект (екземпляр, инстанция) от класа. Всеки обект притежава собствено копие на член-променливите на класа. При създаването на обект се създават и екземпляри на всички член-променливи на класа, които стават "собственост" на създадения обект.

За разлика от член-променливите, всички обекти си поделят само едно копие на член-функциите на класа. Независимо от броя на обектите в програмата има само едно копие на член-функциите на класа.

Нека като пример разгледаме класа `Point` дефиниран по следния начин:

```
1 class Point {
2   double x_, y_;
```

```
3 public:
4   void set_x(double x) { x_=x;}
5   double get_x(void) {return x_;}
6 };
```

Нека са дефинирани два обекта `p1` и `p2` от типа `Point`. Всеки от тези обекти притежава собствено копие от нестатичните член-променливи `x_` и `y_`. Когато методът `get_x()` се извиква чрез обекта `p1`, то използваната в метода член-променлива `x_` принадлежи на обекта `p1` (вж. ред 2).Когато методът `get_x()` се извиква чрез обекта `p2`, то използваната в метода член-променлива `x_` принадлежи на обекта `p2` (вж. ред 3).

```
1 Point p1, p2;
2 p1.set_x(10);
3 p2.set_x(20);
4 p1.get_x();
5 p2.get_x();
```

Като се използва методът `set_x()` се модифицира стойността на член-променливата `x_` за съответния обект — в ред 2 член-променливата `x_` на обекта `p1` получава стойност `10`, а в ред 3 член-променливата `x_` на обекта `p2` получава стойност `20`.

6.6. Структури и класове

В езика `C++` структурите и класовете са тясно свързани. Съгласно определението структурата е клас, за който по подразбиране всички членове са публични. Това означава, че следните две дефиниции са еквивалентни:

```
class s {
public:
  //...
};
```

```
struct s {
  //...
};
```

Във всяко друго отношение структурите се държат като класове – за тях е възможно да се дефинират член-функции, конструктори и деструктори. Например, следните дефиниции са еквивалентни:

```
class Foo1 {
  int bar_;
public:
  Foo1(int bar);
  int get_bar(void);
};
```

```
struct Foo2 {
private:
  int bar_;
public:
  Foo2(int bar);
  int get_bar(void);
};
```

6.7. Конструктори

Член-променливите не могат да се инициализират при тяхната дефиниция. Причината за това е, че при дефинирането на класа не се създават екземпляри на член-променливите му. Това се случва едва когато се създаде обект от дефинирания клас. За всеки създаден обект се създават екземпляри на всички член-променливи на класа, които са свързани със създадения обект. Поради това инициализирането на член-променливите трябва да се извърши при създаване на обекти.

За инициализиране на член-променливите на обектите от даден клас се използва специализирана член-функция, която се нарича *конструктор*. При създаването на всеки обект се вика конструктор, който инициализира член-променливите на обекта. Извикването на конструктора се извършва автоматично при създаването на обект.

Името на конструктора съвпада с името на самия клас. Например, в следващия фрагмент е дефиниран клас `Point`. Конструкторът на този клас е деклариран в ред 4:

```
1 class Point {
2   double x_,y_;
3 public:
4   Point(double x, double y); // конструктор
5   //...
6 };
```

Ако конструкторът има аргументи, то те трябва да се предадат при създаването на обекта. Например:

```
1 Point p1 = Point(1.0,1.0);
2 Point p2(2.0,2.0);
3 Point p3; // грешка
4 Point p4(4.0); // грешка
```

Има възможност за един клас да се дефинират няколко конструктора, които се различават по аргументите, които им се предават.[6]

Конструктор, който се извиква без аргументи се нарича *конструктор по подразбиране*. Например, в следващия фрагмент в ред 3 е деклариран конструктор, който се извиква с два аргумента, а в ред 4 — конструктор по подразбиране:

```
1 class Point {
2 public:
3   Point(double x, double y);
4   Point(void);
5 };
```

Кой конструктор ще се извика в даден конкретен случай се решава според аргументите, които са предадени при извикването на конструктора. Например, в следващия фрагмент, в ред 1 ще извика конструкторът на `Point`, който приема два аргумента, а в ред 2 ще се извика конструкторът по подразбиране.

```
1 Point p1(1.0,1.0);
2 Point p2;
```

6.8. Примери за използване на класове

6.8.1. Точка от равнината

Нека дефинираме клас `Point`, който моделира точка в равнината така, както е описано в раздел 5.2.1. Класът `Point` ще има две член-променливи `x_` и `y_` от типа **double**, които са декартовите координати на точката в равнината. Член-променливите са дефинирани в скритата част на класа. За да имаме достъп до стойностите на тези променливи дефинираме две двойки методи:

- `get_x` и `get_y`, които връщат текущите стойности на x и y-координатата на точката съответно;
- `set_x` и `set_y`, които задават нови стойности на координатите на точката.

```
4 class Point {
5   double x_, y_;
```

[6] В C++ е допустимо да се дефинират няколко функции с едно и също име, които се различават по броя и типа на аргументите. За да реши коя точно функция трябва се извика, компилаторът използва списъка на предадените аргументи (вж. раздел 4.4).

```
6  public:
7    double get_x() {return x_;}
8    double get_y() {return y_;}
9    void set_x(double x) {x_=x;}
10   void set_y(double y) {y_=y;}
```

Дефинираме и конструктор за класа `Point`, който приема два аргумента — декартовите координати на създаваната точка.

```
12   Point(double x=0.0, double y=0.0) {
13     x_=x;
14     y_=y;
15   }
```

Обърнете внимание, че и двата аргумента на конструктора имат стойности по подразбиране. Това позволява конструкторът да се вика без да му се предават аргументи, т.е. този конструктор може да работи и като конструктор по подразбиране.

Като член-функции на класа реализираме и операциите събиране и изваждане на точки от равнината. Член-функцията `add` получава като аргумент точка от равнината и я добавя към обекта, чрез който е извикана.

```
17   void add(Point other) {
18     x_+=other.x_;
19     y_+=other.y_;
20   }
```

Аналогично работи функцията `sub`, която намира разликата между две точки.

```
22   void sub(Point other) {
23     x_-=other.x_;
24     y_-=other.y_;
25   }
```

Като пример за използването на обекти от класа `Point` нека разгледаме функцията `add`:

```
34 Point add(Point p1, Point p2) {
35   Point result(p1.get_x(),p1.get_y());
36   result.add(p2);
37   return result;
38 }
```

Тази функция приема като аргументи две променливи от типа `Point`, изчислява тяхната сума и връща резултата. В ред 35 се създава временна променлива `result` от типа `Point`, като се инициализира така, че нейните координати да са равни на координатите на първия аргумент `p1`. След това, в ред 36, към точката `result` се добавя точката `p2`. Следователно променливата `result` съдържа резултата от събирането на двете точки, предадени като аргументи на функцията — `p1` и `p2`.

Целият код на примерната програма е представен в следващия фрагмент:

```
1  #include <cmath>
2  using namespace std;
3
4  class Point {
5    double x_, y_;
6  public:
7    double get_x() {return x_;}
8    double get_y() {return y_;}
9    void set_x(double x) {x_=x;}
10   void set_y(double y) {y_=y;}
11
12   Point(double x=0.0, double y=0.0) {
13     x_=x;
14     y_=y;
15   }
16
17   void add(Point other) {
18     x_+=other.x_;
19     y_+=other.y_;
20   }
21
22   void sub(Point other) {
23     x_-=other.x_;
24     y_-=other.y_;
25   }
26
27   double distance(Point other) {
28     double dx=x_-other.x_;
29     double dy=y_-other.y_;
30     return sqrt(dx*dx+dy*dy);
```

```
31    }
32  };
33
34  Point add(Point p1, Point p2) {
35    Point result(p1.get_x(),p1.get_y());
36    result.add(p2);
37    return result;
38  }
39
40  Point sub(Point p1, Point p2) {
41    Point result(p1.get_x(), p2.get_y());
42    result.sub(p2);
43    return result;
44  }
45
46  double distance(Point p1, Point p2) {
47    return p1.distance(p2);
48  }
```

6.8.2. Стек

Нека като пример разгледаме дефинирането на клас `Stack`, моделиращ абстрактния тип *стек*. Основните операции, които могат да се извършват с един стек са:

- добавяне на нов елемент в стека — `push()`;
- изваждане на последния добавен елемент от стека — `pop()`.

Поради тази причина, често стекът се нарича FILO (First In, Last Out) — първи влязъл, последен излязъл.

Възможните начини за реализиране на стек са няколко. Най-простият вариант е да се използва масив, като ако е необходимо стекът да е динамичен,[7] може да се използва динамичен масив. Друг вариант за реализирането на стека е да се използва списък — например двусвързан списък. Независимо от начина на реализация операциите които могат да се извършват със стека са едни и същи — `push()` и `pop()`.

Като пример ще разгледаме възможно най-простата реализация на класа `Stack`, която използва масив с фиксиран размер. Предложената реализация на класа `Stack` има две член-променливи: член-променливата

[7] Броят на елементите, които могат да се поберат в масива да не е фиксиран по време на компилацията на програмата.

`data_`, която е масивът в който се съхраняват елементите на стека, и член-променлива `top_`, която е индексът на следващия свободен елемент на масива.

```
const int STACK_SIZE=10;
class Stack {
  int data_[STACK_SIZE];
  int top_;
public:
  Stack() {
    top_=0;
  }
```

Основните операции със стека са операциите `push` и `pop`. Операцията `push` добавя нов елемент в стека, като го поставя в масива `data_` на мястото сочено от индекса `top_`. След това индексът `top_` се увеличава с единица, за да сочи към следващия свободен елемент. Обърнете внимание, че ако стекът е пълен, т.е. `top_>=STACK_SIZE`, то операцията `push` не прави нищо.

```
  void push(int val) {
    if(top_<STACK_SIZE) {
      data_[top_++]=val;
    }
  }
```

Операцията `pop` връща стойността на следващия елемент от стека и го изтрива. За тази цел първо се намалява стойността на `top_` с единица и след това се връща стойността на елемента от масива с индекс новата стойност на `top_`. Обърнете внимание, че ако стекът е празен, операцията `pop` връща нула без да дава съобщение за грешка.

```
  int pop(void) {
    if(top_>0) {
      return data_[--top_];
    }
    return 0;
  }
```

Към реализацията на стека са добавени две допълнителни операции, които проверяват дали стекът е празен `is_empty()` или е пълен — `is_full()`:

```
  bool is_empty() {
    return top_==0;
```

```
22    }
23    bool is_full() {
24      return top_==STACK_SIZE;
25    }
26  };
```

Като пример за използване на класа `Stack` нека разгледаме програма, която обръща стрингове:

```
28  int main(int arch, char* argv[]) {
29    char* msg="Hello!";
30    char buff[10];
31    Stack st;
32    for(char* p=msg;*p!='\0';p++)
33      st.push(*p);
34    char* p=buff;
35    while(!st.is_empty())
36      *p++=st.pop();
37    *p='\0';
38    return 0;
39  }
```

В ред 29 е дефиниран стрингът `msg="Hello!"`. Целта на програмата е да се получи нов стринг, в който буквите са подредени в обратен ред. Новият стринг ще се конструира в буфера `buff`. За обръщането на реда на символите се използва стекът `Stack st`, дефиниран в ред 31. Цикълът в редовете 32–33 обхожда всички букви от стринга `msg` и ги слага в стека, като използва операцията `st.push()` — вж. ред 33. След изпълнението на този цикъл, всички символи от оригиналния стринг `msg` са вкарани в стека. Следващия цикъл 34–36 изважда всички букви от стека като използва операцията `st.pop()` и ги подрежда в буфера `buff`. Тъй като стекът е LIFO (Last In First Out) структура, първият символ, който ще извадим от стека е последният символ, който е поставен в стека, т.е. последният символ от стринга `msg`, и т.н. Накрая трябва да добавим терминираща `'\0'`, за да могат символите в буфера `buff` да се интерпретират като стринг.

7. Пространство от имена (*namespace*)

Пространствата от имена (*namespaces*) са въведени в `C++` като поддръжка на така нареченото модулно програмиране. По същество прост-

ранствата от имена позволяват изграждането на дървовидна структура от имена на идентификаторите в една C++ програма като по този начин намаляват риска от конфликт на имената.

Да си представим за момент, че файловата структура на някакъв компютър няма добре познатата йерархична структура от директории, поддиректории и файлове. В такъв случай всички файлове биха се съхранявали в едно общо глобално хранилище, което се вижда постоянно от всеки потребител и приложение. Като следствие от подобна ситуация биха възникнали ред трудности при преглеждане, търсене или копиране на файлове. Освен това лесно може да си представим за многобройни конфликти на имената на файловете, когато различни потребители искат да използват едно и също име за два различни файла.

Подобно е и положението при една програма, когато нейният размер започне да нараства — конфликтите на имена започват да стават често явление. Точно за решаването на този проблем в C++ са въведени пространствата от имена. Пространствата от имена са аналогични на директориите в горния пример. Те могат да бъдат влагани едно в друго и да образуват йерархични структури от имена, подобни на файловата система. Такава йерархична структура от имена може лесно да предпази кода на една програма от конфликти на имената. Повечето от компонентите на стандартната C++ библиотека например, са групирани в пространството от имена `std`. То от своя страна е разделено на подпространства от имена.

7.1. Дефиниране на пространство от имена

За дефиниране на именувано пространство от имена се използва ключовата дума **namespace**. Например:

```
1 namespace elsys {
2   class School {
3     ...
4   };
5   class Student {
6     ...
7   };
8 };
```

В този фрагмент са декларирани два класа `School` и `Student` като членове на пространството от имена `elsys`.

Към едно пространство от имена винаги може да се добавят нови имена. Например, в следващия фрагмент към пространството от имена `elsys` се добавя класа `Theacher`:

```
namespace elsys {
  class Theacher {
    ...
  };
};
```

7.2. Използване на пространства от имена

Когато една променлива или тип са дефинирани в рамките на дадено пространство от имена, то за използване на съответния идентификатор има няколко варианта:

- Името на идентификатора трябва да се квалифицира пълно. Например, за да дефинираме променлива от типа `Theacher`, дефиниран в пространството от имена `elsys`, трябва като име на типа да се използва `elsys::Theacher`:

  ```
  elsys::Theacher theacher;
  ```

- Името на идентификатора може да бъде включено в текущата област на видимост, като се използва **using**-дефиниция. Например, в следващия фрагмент в текущата област на видимост се включва идентификаторът `Theacher`, дефиниран в пространството от имена `elsys`:

  ```
  using elsys::Theacher;
  Theacher theacher;
  ```

  След включването на идентификатора `Theacher` в текущата област на видимост, той може свободно да бъде използван бе изрично да се указва към кое пространство от имена принадлежи той.

- Третият вариант е в текущата област на видимост да се включат всички идентификатори, дефинирани в рамките на дадено пространство от имена. За целта се използва **using**-декларация. Например, в следващия фрагмент в текущото пространство от имена се включват всички идентификатори, декларирани в пространството от имена `elsys`:

  ```
  using namespace elsys;
  ```

```
Theacher theacher;
Student student;
```

След използването на такава декларация, всички идентификатори от пространството от имена `elsys` могат да се използват свободно в текущата област на видимост.

7.3. Пространство от имена `std`

Повечето от типовете, променливите и функциите от стандартната `C++` библиотека са дефинирани в пространството от имена `std`. Поради това често срещана практика е, да се използва **using**-декларация за включване на идентификаторите от стандартното пространство от имена в текущата област на видимост. Например:

```
#include <cmath>
#include <cstdlib>
using namespace std;
```

В представения фрагмент се използват два заглавни файла, които са част от стандартната `C++` библиотека. В заглавния файл `<cmath>` са декларирани основните математически функции като `sqrt`, `cos`, `acos` и т.н. В заглавния файл `<cstdlib>` са декларирани функциите от `C`-файла `<stdlib.h>`, като например `exit`, `malloc`, `free` и т.н. Всички декларации и дефиниции в тези заглавни файлове са включени в пространството от имена `std`. За да могат да се използват е необходимо да се включат в текущата област на видимост, което се постига с използването на конструкцията **using namespace** `std`.

8. Входно изходни операции

Тъй като `C++` и `C` са родствени езици, в една програма на `C++` е напълно възможно да се използва стандартната `C`-библиотека за вход и изход. С други думи следната програма е валидна `C++` програма.

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4   printf("Hello␣world!\n");
5   return 0;
6 }
```

От друга страна обаче, езикът C++ има собствена библиотека за входно/изходни операции. В C++ входно/изходните операции са организирани като операции с потоци. При разработването на входно/изходната библиотека на C++ специално внимание е обърнато на удобството и лекотата на използване на библиотеката.

8.1. Стандартни потоци за вход и изход

Стандартните потоци за вход и изход са декларирани в заглавния файл <iostream>. Както всички останали идентификатори от стандартната библиотека, потоците за вход и изход също са дефинирани в пространството от имена std. Поради това за да могат да се използват те трябва да бъдат включени в текущата област на видимост като се използва **using**-директива или техните имена трябва да се указват пълно.

8.2. Стандартен поток за изход cout

Следващият фрагмент използва стандартния поток за изход cout:

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

int main(int argc, char* argv[]) {
  cout << "Hello world!" << endl;
  return 0;
}
```

При разработването на входно изходната библиотека като оператор за извеждане е избран операторът <<. В един израз могат да се комбинират няколко оператора за изход. Например:

```
cout << "Hello" << " " << "world!" << endl;
```

Освен това с един оператор за изход могат да се извеждат различни типове данни. За извеждане на край на реда се използва endl. Например:

```
cout << "The answer is " << 42 << endl;
```

Работа на потока е да види какви типове данни се извеждат и да се съобрази със спецификата на извеждане на всеки от тях.

8.3. Стандартен поток за вход cin

Стандартният поток за вход е cin. Операторът за четене от потока е >>. Потокът за вход може да обработва последователност от различни

по тип променливи. Например:

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

int main(int argc, char* argv[]) {
  int number1, number2;
  cin >> number1 >> number2;
  cout << "number1=" << number1 << endl
       << "number2=" << number2 << endl;
  return 0;
}
```

9. Обработка на изключения

9.1. Обработка на грешки

Да си представим, че по време на изпълнение дадена функция открива ненормална, грешна ситуация. Причината за възникването на такава ситуация може да бъде различна — неправилни входни данни, препълване на диска, изчерпване на наличната динамична памет, невъзможност да се отвори файл и т.н. По какъв начин функцията трябва да реагира на такава ситуация?

Подходите могат да бъдат различни. Обикновено всеки програмист използва собствен стил за обработка на грешки, което води до голямо разнообразие от програмистки практики. Един от подходите, който сравнително стандартно се използва в езика `C`, е функцията, открила ненормална ситуация да върне резултат, който сигнализира за наличието на грешка. Голяма част от функциите в стандартната `C` библиотека са организирани точно по този начин. Например функцията за отваряна на файлове:

```
FILE* fopen(const char* filename, const char* mode);
```

връща нулев указател `NULL` ако не успее да отвори файл със зададеното име. Функциите за писане във файл

```
int fputc(int c, FILE* file);
int fputs(const char* str, FILE* file);
```

връщат `EOF` когато не успеят да запишат данните във файла. По подобен начин работи и функцията за четене от файл:

```
int fgetc(FILE* file);
```

При възникване на грешка или достигане до края на файла тази функция връща `EOF`. В стандартната `C` библиотека могат да се намерят огромно количество подобни примери.

Имайки предвид тази практика нека разгледаме метода **void** `push()` на класа стек от раздел 6.8.2:

```
class Stack {
...
public:
...
  void push(int val) {
    if(top_<STACK_SIZE) {
      data_[top_++]=val;
    }
  }
...
};
```

Когато стекът е пълен, методът **void** `push()` по никакъв начин не сигнализира за наличие на грешка. Едно решение на този проблем е да преработим метода **void** `push()` така, че да връща стойност — например **int**, — като ако възникне грешка, методът връща ненулева стойност, а ако всичко е наред, връща нула:

```
  int push(int val) {
    if(top_<STACK_SIZE) {
      data_[top_++]=val;
      return 0;
    }
    return -1;// Грешка: стека е пълен
  }
```

Как обаче можем да се справим с подобен проблем във член-функцията **int** `pop()` (вж. раздел 6.8.2). Член-функцията е дефинирана по следния начин:

```
class Stack {
...
public:
...
  int pop(void) {
```

```
6       if(top_>0) {
7         return data_[--top_];
8       }
9       return 0;
10    }
11  ...
12  };
```

Проблемът с тази функция, е че ако стекът е празен, тя връща нула без по какъвто и да било начин да сигнализира за настъпилата грешка. Директното прилагане на развитата по-горе техника за сигнализиране за грешки е невъзможно, тъй като функцията **int** `pop()` връща резултат. При това всяка стойност на резултата е допустима, тъй като в стека могат да се съхраняват всякакви цели числа.

Един от вариантите за справяне с този проблем е функцията да се промени по следния начин:

```
1   int pop(int& val) {
2     if(top_>0) {
3       val=data_[--top_];
4       return 0;
5     }
6     return -1;// Грешка: стека е празен
7   }
```

Стойността на елемента от стека се връща като стойност на аргумента `val`, а резултатът от функцията се разглежда изцяло като код за грешка. Ако резултатът от функцията е нула, то функцията е работила правилно; ако резултатът е ненулев, то е настъпила грешка при изпълнение на функцията.

Разгледаният подход за обработка на грешки макар и често използван е тежък и тромав. Използването му води до това, че при всяко извикване на функция, програмистът трябва да изследва резултата от тази функция за възможни настъпили грешки. Това прави кода на програмата труден за разбиране и поддържане. Друг недостатък на разглеждания подход е, че при него няма стандарти, които да се спазват от всички. При настъпване на грешка може да връща ненулев резултат, друга да връща нула, а трета — отрицателно число. Това прави трудно еднотипното обработване на грешки.

9.2. Генериране и обработка на изключения

Механизмът за обработката на изключения в C++ предоставя стандартни, вградени в езика средства за реагиране на ненормални, грешни ситуации по време на изпълнение програмата. Механизмът на изключенията предоставя еднообразен синтаксис и стил за обработка на грешки в програмата. Той елиминира нуждата от изрични проверки за грешки и съсредоточава кода за обработка на грешки в отделни части на програмата.

Основните компоненти, които формират механизма за обработка на изключения в C++ са следните:

- При възникване на ненормална ситуация в програмата, програмистът сигнализира за настъпването ѝ чрез генерирането на изключение. Когато се генерира изключение нормалното изпълнение на програмата се прекратява докато изключението не бъде обработено. В C++ за генериране на изключение се използва ключовата дума **throw**. Например, в следния фрагмент в ред 8 се генерира изключение от типа `StackError` ако се опитаме да прочетем елемент от празен стек:

```
1  class StackError { ... };
2  class Stack {
3  ...
4  public:
5  ...
6    int pop(void) {
7      if(top_<=0)
8        throw StackError;
9      return data_[--top_];
10   }
11 ...
12 }
```

- Точка в програмата, където генерираното изключение се обработва. Най-често изключенията в програмата се генерират и обработват от различни функции. Намирането на секция за обработка на генерирано изключение често води до развиване на програмния стек. След като изключението бъде обработено изпълнението на програмата продължава нормално. Възстановяването на изпълнението на програмата обаче става не от точката на генериране на изключението, а от точката, където изключението е било обработено.

В C++ обработката на изключенията се изпълнява в **catch**-секции. Например, в следващата **catch**-секция се обработва изключението, генерирано в предходния фрагмент:

```
1 catch(StackError ex) {
2   log_error(ex);
3   exit(1);
4 }
```

- Всяка една **catch**-секция трябва да се асоциира с **try**-блок. В един **try**-блок се групират един или повече оператори, които могат да генерират изключения, с една или повече **catch**-секции. Например:

```
1  try {
2    // Използване на обекти от класа Stack
3    ...
4  } catch(StackError ex) {
5    // Обработка на грешка при използването на стека
6    ...
7  } catch(...) {
8    // Обработка на всички останали грешки
9    ...
10 }
```

Ако се генерира изключение от типа `StackError`, то ще бъде прихванато и обработено от **catch**-секцията в ред 4. Използването на секция **catch**`(...)`, където вместо тип на изключението се използва многоточие, означава, че тази **catch**-секция прихваща изключения от всякакъв тип. Следователно всички изключения, генерирани в **try**-блока, които не са от типа `StackError` ще бъдат прихванати и обработени от **catch**-секцията в ред 7.

9.3. Пример за използване на изключения

Като пример за използване на изключения нека разгледаме класа `Stack`, който беше въведен в раздел 6.8.2. За целта в методите `push()` и `pop()` е добавена обработка на грешки чрез генериране на изключения от типа `StackError`:

```
1 class StackError {};
2 const int STACK_SIZE=10;
3 class Stack {
4   int data_[STACK_SIZE];
```

```
5     int top_;
6   public:
7     Stack() {
8       top_=0;
9     }
10    void push(int val) {
11      if(top_>=STACK_SIZE)
12        throw StackError();
13      data_[top_++]=val;
14    }
15    int pop(void) {
16      if(top_<=0)
17        throw StackError();
18      return data_[--top_];
19    }
20    bool is_empty() {
21      return top_==0;
22    }
23    bool is_full() {
24      return top_==STACK_SIZE;
25    }
26  };
```

В главната функция на програмата от раздел 6.8.2 е добавен **try-catch**-блок за обработка на грешки, възникнали при работа на програмата:

```
28  #include <cstdlib>
29  #include <iostream>
30  using namespace std;
31  int main(int arch, char* argv[]) {
32    char* msg="Hello␣Cruel␣World!";
33    char buff[10];
34    try {
35      Stack st;
36      for(char* p=msg;*p!='\0';p++)
37        st.push(*p);
38      char* p=buff;
39      while(!st.is_empty())
40        *p++=st.pop();
41      *p='\0';
42    } catch(StackError ex) {
```

```
43      cerr << "StackError␣catched..." << endl;
44      exit(1);
45    } catch(...) {
46      cerr << "Unknown␣error␣catched..." << endl;
47      exit(1);
48    }
49    return 0;
50  }
```

10. Шаблони (*templates*) и родово (*generic*) програмиране

Литература

[1] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, 2nd edition, 1988.

[2] Andrew Koenig and Bjarne Stroustrup. C++: As close to C as possible— But no closer. *The C++ Report*, 1989.

[3] Stanley Lippman and Josee Lajoie. *C++ Primer*. Addison-Wesley, Reading (MA), USA, 3rd edition, 1997.

[4] X3 Secretariat. *Standard — The C Language. X3J11/90013. ISO Standard ISO/IEC 9899*. Computer and Business Equipment Manufacturers Associataion, Washington, DC (USA), 1990.

[5] X3 Secretariat. *International Standard — The C++ Language. X3J16–14882*. Information Technology Council (NSITC), Washington, DC (USA), 1998.

[6] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading (MA), USA, 1994.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading (MA), USA, 3rd edition, 1997.

[8] Димитър Богданов. *Обектно ориентирано програмиране със C++*. Издателство “Техника”, София, 1994.

[9] Стенли Липман. *Езикът C++ в примери*. Колхида Трейд, София, 1993.

[10] Браян Керниган и Денис Ричи. *Програмният език C*. ЗеСТ Прес, София, 2004.

[11] Бьорн Строустроп. *Програмният език C++*. ИнфоДАР, София, 2001.

[12] Кай Хорстман. *Принципи на програмирането със C++*. Софтех, София, 2000.