Въведение в C++
(Rev: 1.1)

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

[1]Технологическо училище "Електронни системи"
Технически университет, София

19 септември 2007 г.

Съдържание

История на езика C++

- Езикът C++ е разработен от Bjarne Stroustrup. Работата по езика започва през 1979. Първият му вариант се появява през 1980 – "C with Classes".
- Името C++ се използва за първи път през 1983, а през 1984 се появява следваща версия на езика. В следващите години езикът продължава да се развива и разпространява.
- През 1998 г. е одобрен стандартът за езика C++ – ISO/IEC 14882 "Standard for the C++ Programming Language".

Връзка между C и C++

- Като базов език за C++ е избран езикът C.
- Основна цел при разработването на C++ е той да бъде съвместим със C. Всяка конструкция, която е допустима в C и C++, има еднакъв смисъл в двата езика.
- Пълна съвместимост между C и C++ няма. Налагането на пълна съвместимост между двата езика би довело до жертването на различни предимства, които C++ има.

Обща характеристика на езика

- C++ е език за програмиране с общо предназначение.
- C++ предоставя механизми за поддръжка на обектно-ориентиран стил на програмиране.
- C++ е създаден с цел да се добави поддръжка на обектно-ориентираното и обобщено програмиране към традиционния C.

Типове

- Всеки идентификатор в една C++ програма, трябва да има асоцииран с него *тип*.
- *Типът* определя какви операции са приложими към дадения идентификатор и как трябва да се интерпретират тези операции.
- В езика C++ са дефинирани набор от базови (фундаментални) типове и са предоставени средства за дефиниране на нови типове от потребителя.

Вградени типове

- Логически (булев) тип – **bool**
- Символни типове – **char**, **wchar_t**
- Целочислени типове – **int**
- Типове с плаваща запетая – **float**, **double**
- Изброим тип – дефиниран от потребителя с използването на **enum**
- Типът **void**

Освен тези типове, могат да се конструират и други:

- Указатели – например **int***
- Масиви – например **char**[]
- Препратки – например **double&**
- Структури от данни и класове

Примери

```
1  bool a=false;
2  bool b=true;
3  bool bb=a||b;
4
5  char ch='a';
6
7  int count=1;
8  unsigned int i=0;
9  long int li;
10
11 double x=0.0;
12 double y, z;
13 const double pi=3.14159265358;
```

Типът void

- По принцип типът **void** е фундаментален (базов) тип, но неговото използване е ограничено.
- Типът **void** може да се използва само като част от по-сложен тип. Обекти от тип **void** не съществуват.
- Допустимото използване на този тип е:
  - да укаже, че дадена функция не връща резултат – **void** `fun();`.
  - като базов за указател към обект от неизвестен тип – **void*** `pv;`.

Променливи

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

```
1 int counter;
2 double sum;
```

- Всяка променлива може да бъде инициализирана при нейното дефиниране:

```
1 int i=0;
2 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.

Константи

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

```
1 const double e =2.71828182845904523536;
2 const double pi=3.14159265358979323846;
3 const char[] message="warning: ";
```

Аритметичните оператори

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

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

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

Неявно преобразуване на типовете

- Целочислените типове и типовете с плаваща запетая могат 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.

Неявно преобразуване на типовете

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

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

- 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.

Явно преобразуване на типа

- Има редица ситуации, при които се налага изрично една променлива да бъде преобразувана към друг тип.

```
1   int a=2;
2   int b=3;
3   double d1=b/a;
4   double d2=static_cast<double>(b)/a;
```

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

Декларации

- Преди даден идентификатор да може да се използва в една програма на C++, той трябва да бъде деклариран.
- *Декларация* е термин, който се използва за всичко, което казва на компилатора какъв е смисълът на даден идентификатор.
- За да се използва даден идентификатор, компилаторът трябва да знае какво представлява този идентификатор — дали е име на променлива, на функция, на тип или на нещо друго. С други думи трябва да бъде указан *типът* на идентификатора.
- Поради това, във всеки файл с код трябва да се съдържа декларация на всички имена, които се използват.

Примери

```
1  char ch;
2  int count=1;
3  const double pi=3.14159265358979;
4  extern int error_number;
5  char* name="Bjarne␣Stroustrup";
6  char* season[]={"spring","summer","autumn","winter"};
7
8  struct Date {int d,m,y;};
9  int day(Date* p) {return p->d;}
10 double sqrt(double);
11
12 struct User;
13 enum Beer {Carlsberg, Tuborg, Beiks, Amstel};
```

Декларации и дефиниции

- Повечето от представените *декларации* са всъщност *дефиниции* – те определят някаква същност, която съответства на дадения идентификатор:
  - за променливата `ch` дефинираната същност представлява подходящото количество памет, необходима на тази променлива;
  - за структурата `Date`, дефинираната същност представлява нов тип;
  - за функцията **int** `day(Date* p)` дефинираната същност е алгоритъмът по който се изпълнява функцията;

Декларации и дефиниции

- От представените примерни *декларации*, само следните *не са дефиниции*:

```
extern int error_number;
double sqrt(double);
struct User;
```

- Променливата `error_number`, функцията `sqrt` и структурата `User` трябва да бъдат *дефинирани* някъде другаде в програмата.

Декларации и дефиниции

- В дадена програма на C++ може да *има няколко декларации* за даден идентификатор.
- В дадена програма на C++ за всеки идентификатор може да има *само една дефиниция*.

```
1 int count;
2 int count;// error
3
4 extern int error_number;
5 extern int error_number; // OK!
6
7 extern short error_number; // error!
```

Указатели

- За даден тип T, типът T* е *указател към T*. С други думи, променливите от тип T* съдържат адрес на обект от типа T.

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

- Основната операция, която се изпълнява върху указателите, е операцията *. Този оператор връща обекта, към който сочи указателят.

```
1 int a=42;
2 int* pa=&a;
3 int a1=*pa;
```

Препратки

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

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

- При дефиниране на препратка, тя задължително трябва да бъде инициализира с обект от съответния тип.

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

Препратки

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

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

Препратки

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

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

Препратки

- 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.

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

Дефиниране на масиви

- За даден тип `T`, типът `T[size]` е масив от `size` елемента от тип `T`. Елементите на масива се индексират (номерират) от `0` до `size-1`. Броят на елементите на масива трябва да бъде константен израз.

```
1 int b[3];
2 char* a[42];
```

- Многомерните масиви се дефинират като масиви от масиви.

```
1 int d2[10][10];
2 int d3[10][10][10];
```

Инициализация на масиви

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

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

- Когато масивът е деклариран без да е указан неговият размер, броят на елементите в масива може да се определи от размера на инициализиращия списък.
- Когато размерът на масива е указан явно, инициализирането на масива със списък, съдържащ повече елементи е грешка.

```
int v[2]={1,2,3}; // Грешка!
```

Инициализация на масиви

- 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

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

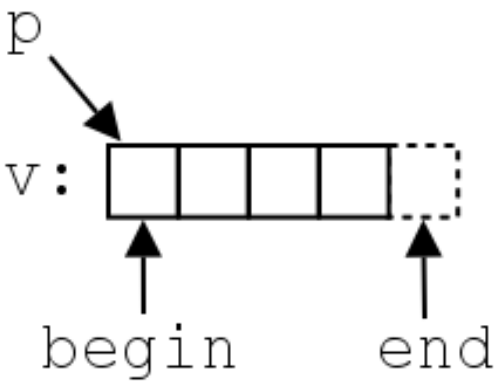
  е еквивалентна на

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

- За инициализирането на многомерни масиви се използва списък от списъци за инициализация.

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

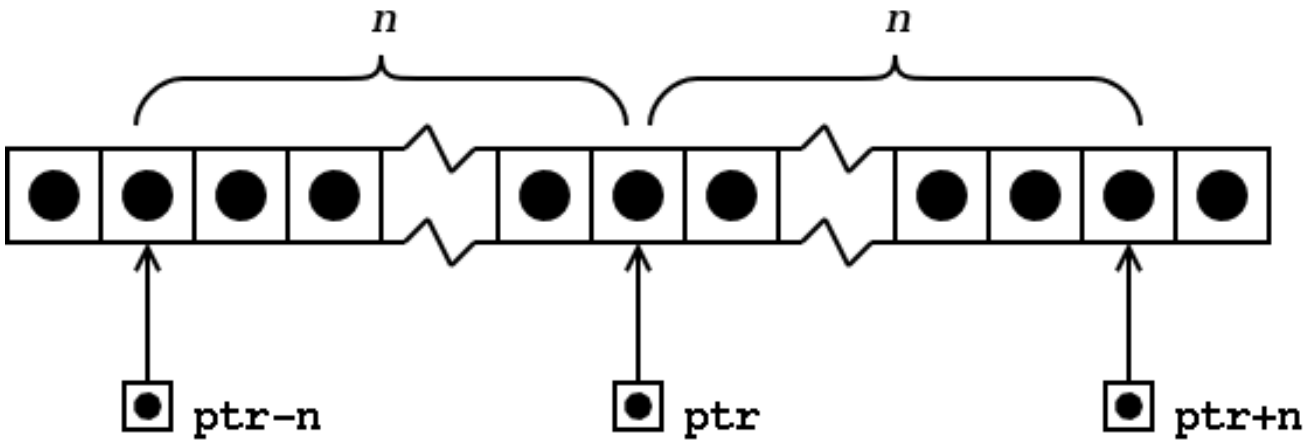
Указатели и масиви




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

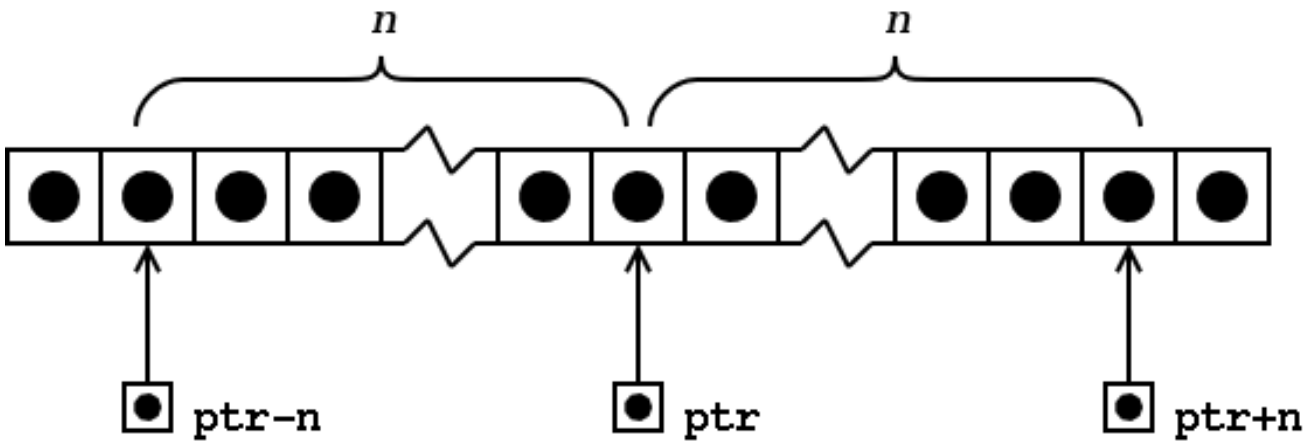
- Указателите и масивите са тясно свързани. Името на масива може да се използва като указател, сочещ към първия елемент на масива.
- Езикът гарантира, че стойността на указател, насочен с едно след последния елемент на масива, е смислена.
- Тъй като този указател не сочи към елемент от масива, той не бива да бъде използван за четене на стойност или записване на стойност.

Аритметика на указател и цяло число



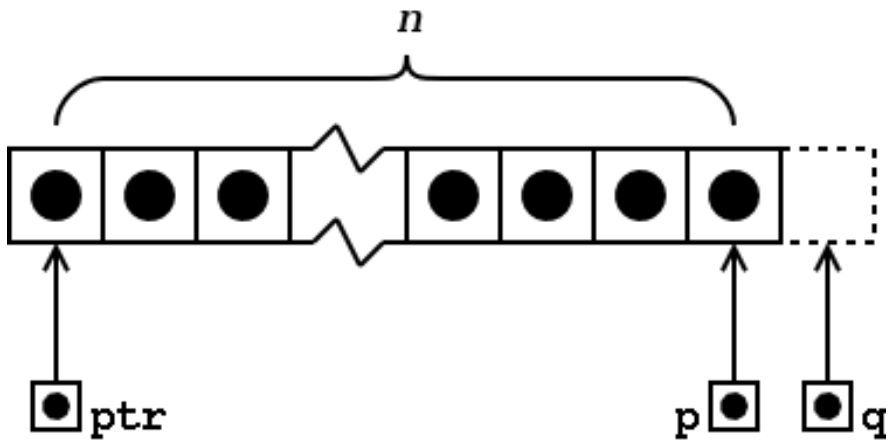
- Когато към указател се добавя цяло число, резултатът ще бъде указател, отместен със съответния брой елементи към края на масива.

Аритметика на указател и цяло число



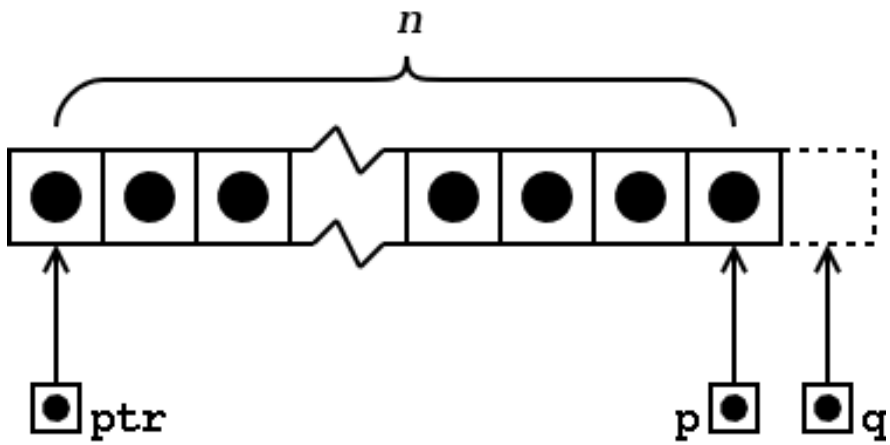
- Когато от указател се изважда цяло число, резултатът ще бъде указател, отместен със съответния брой елементи към началото на масива.
- И в двата случая, ако полученият указател не сочи към елемент на масива или с едно след последния елемент, резултатът не е дефиниран.

Елемент с едно след последния



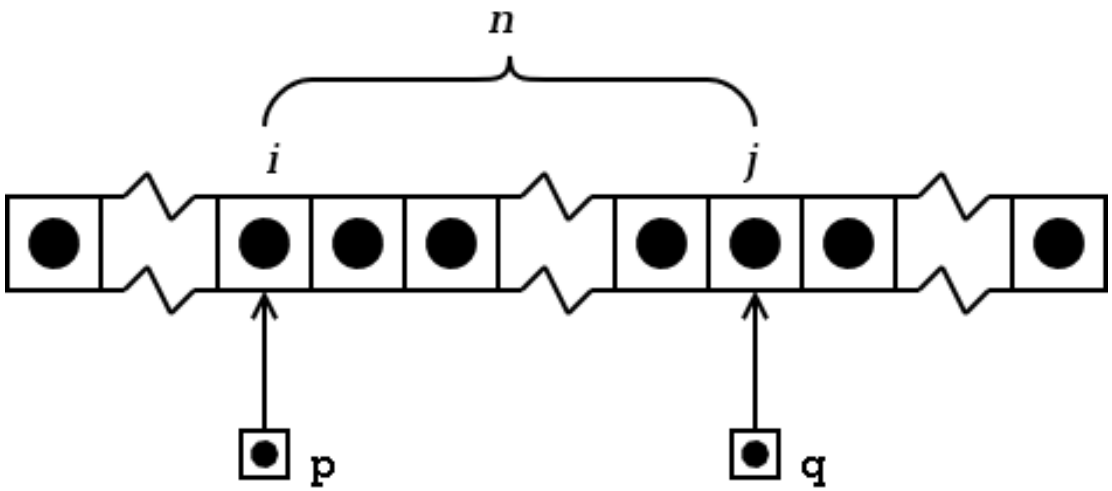
- Дефиницията на езика гарантира, че стойността на указател, насочен с едно след последния елемент на масива, е смислена.

Елемент с едно след последния



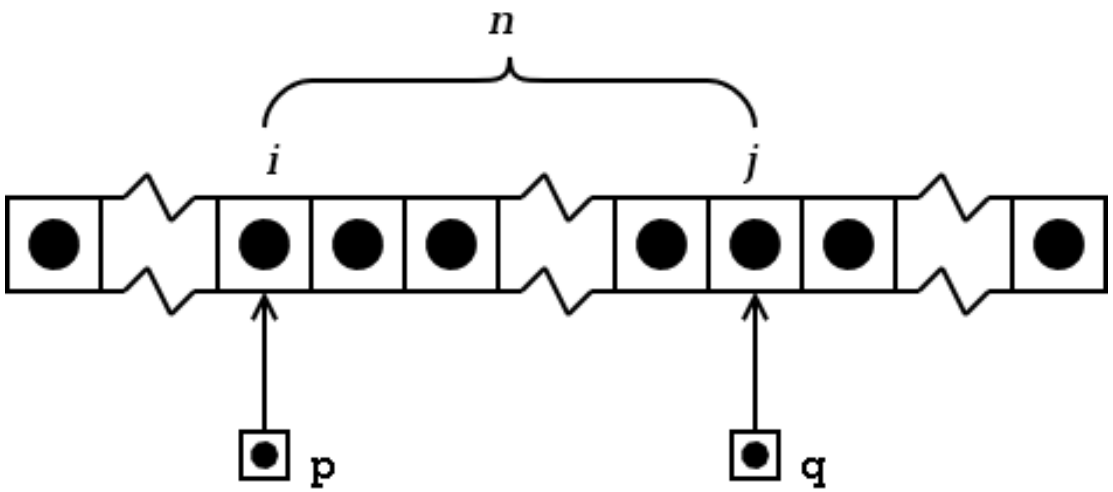
- Ако указателят `p` сочи към последния елемент на даден масив, то `(p+1)` е указател насочен с едно след последния елемент на масива.
- Ако указателят `q` сочи с едно след последния елемент на масива, то `(q-1)` сочи към последния елемент на масива.

Изваждане на указатели



- Изваждането на един указател от друг указател е дефинирано само в случай, че двата указателя сочат към елементи на един и същ масив.

Изваждане на указатели



- Ако указателят `p` сочи към i-тия елемент от масива, а указателя `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 $i - j = -n$.

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 групираният код да се използва лесно и многократно.

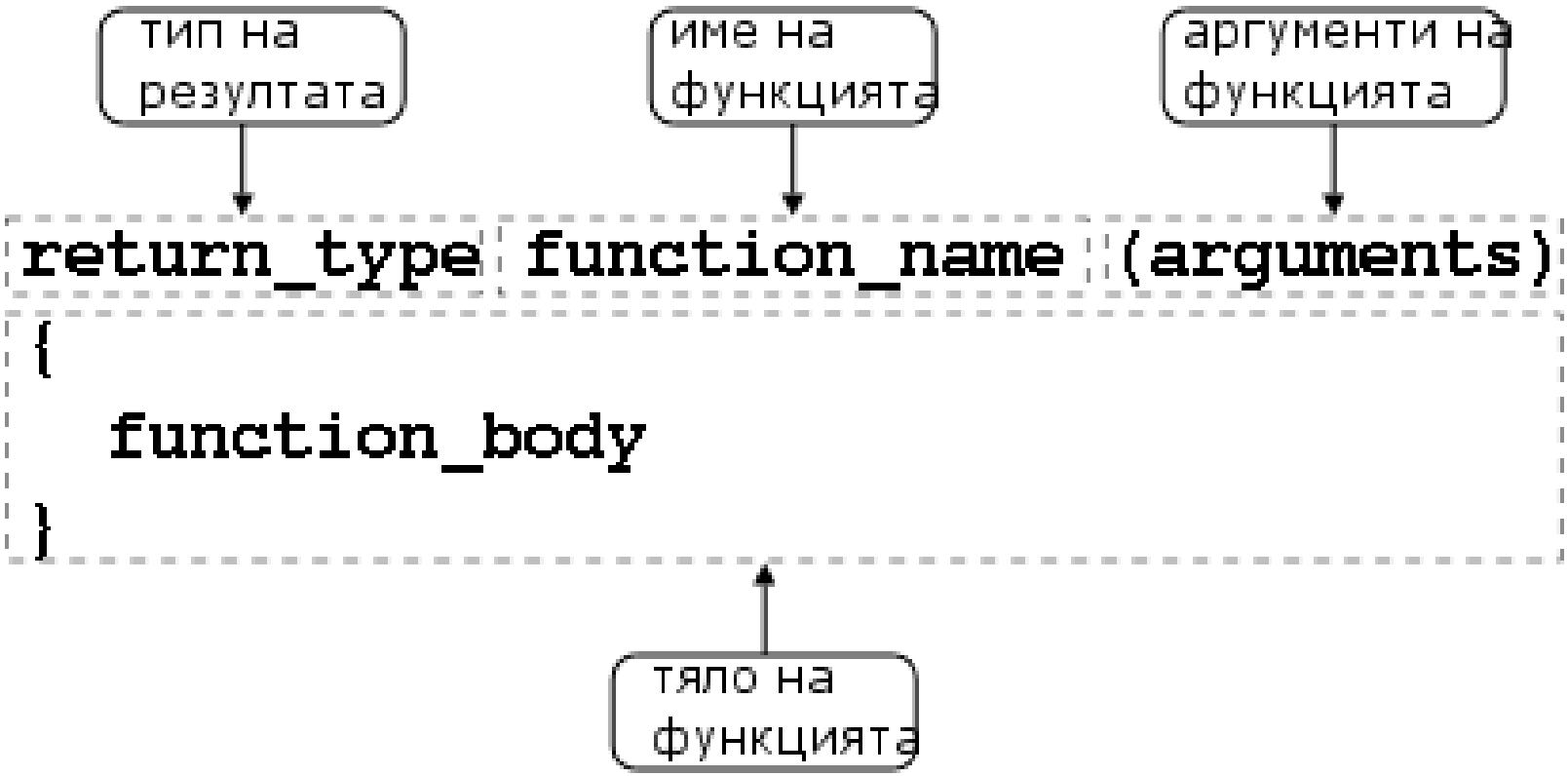
Деклариране на функции

- 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.

Дефиниране на прототип на функция


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

Дефиниране на функция



Примери

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

```
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 }
```

Предаване на аргументи по стойност

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

Предаване на указатели към аргументите

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

Предаване на препратки

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

Предефиниране 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
- 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*, *припокриване на функции*.

Пример

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

Аргументи по подразбиране

При дефиниране на функции в C++ на параметрите на функцията могат да се задават стойности по подразбиране.

```
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 }
```

Дефиниране на структура

- Структурата е съвкупност на елементи от (почти) произволен тип.

```
1 struct person {
2   char* name;
3   long int age;
4 };
```

- Името на структурата `person` се превръща в име на тип и могат да се дефинират променливи.

```
6 person somebody;
```

- За инициализирането на структура се използва запис, подобен на инициализацията на масив.

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

Достъп до членове на структура

- Достъпът до членовете (полетата) на структурата се осъществява с използването на оператора . (точка).

```
7 somebody.name="ivan";
8 somebody.age=16;
```

- Когато достъпът до структурата се извършва чрез указател, то членовете на структурата са достъпни чрез оператора ->.

```
11 void dump(person* ptr) {
12   cout << ptr->name << endl
13        << ptr->age << endl;
14 }
```