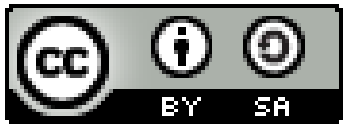
Функции C++



March 3, 2021

Функции определяют логически обособленные **действия** в программе.

Определить функцию – задать действие.

Вызвать функцию – выполнить действие.

Весь (почти) исполняемый код программы находится в функциях. Функции могут вызываться как явно (программистом), так и неявно (компилятором).

Описание функции

```
double f( int a, bool b ); //  описание функции
```

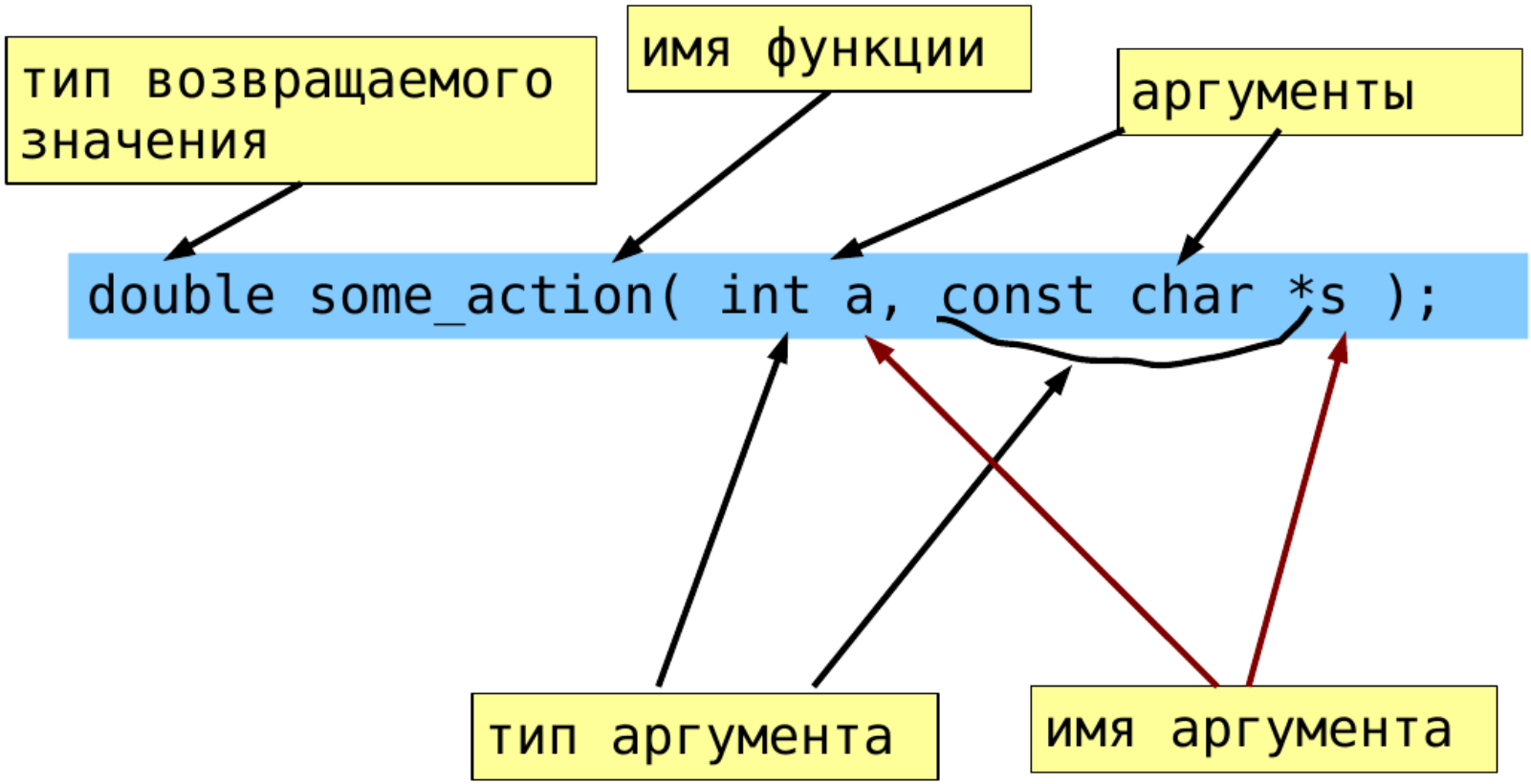
После описания функцию можно использовать (вызывать).

```
double x = f( 4, true ); //– вызов функции: оператор ()
```

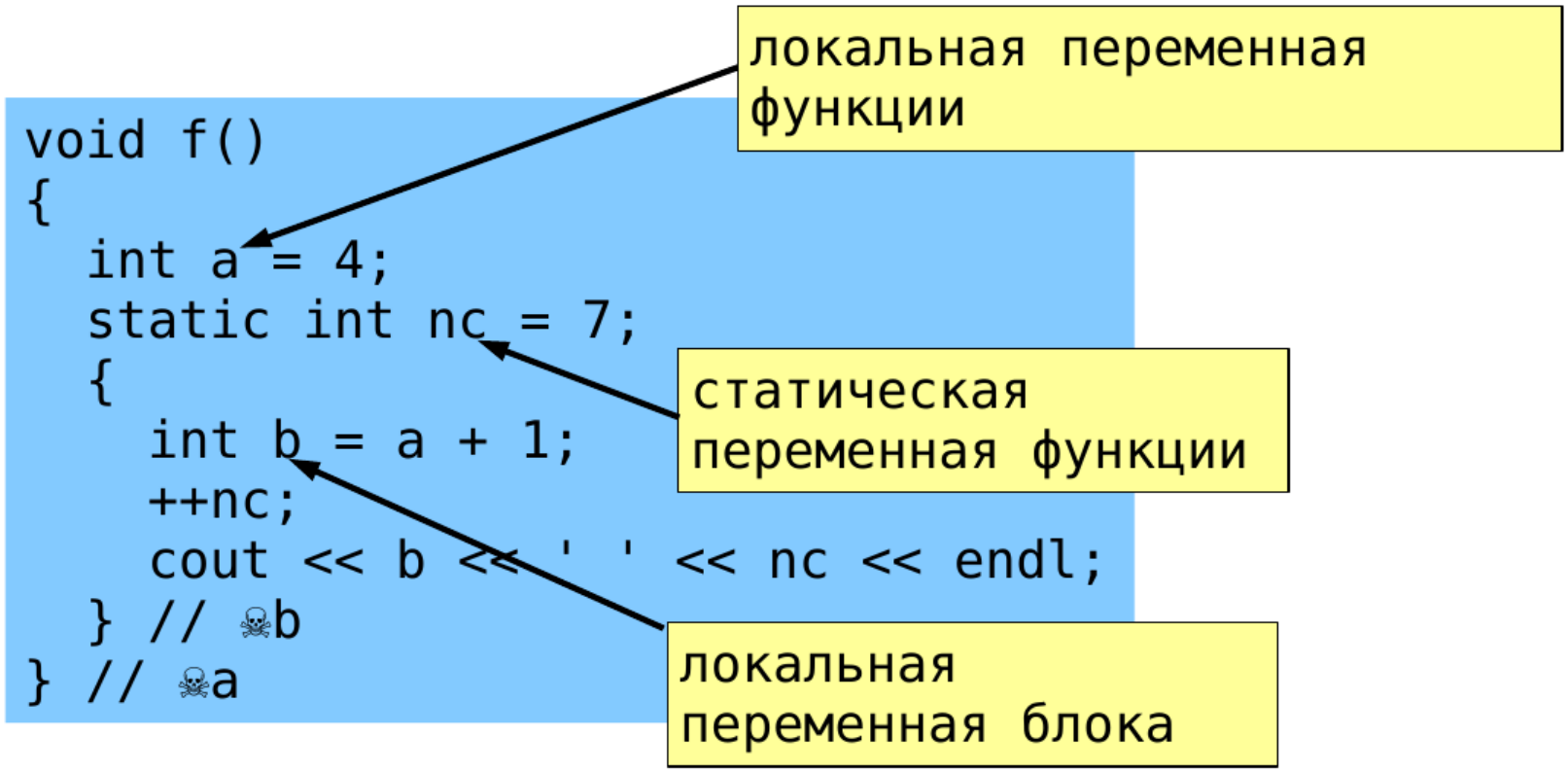
Определение функции может быть в любом месте проекта (в том числе и в другом файле, в стандартной или специальной библиотеке). Определение может быть и описанием. Все необходимые для программы функции собирает компоновщик (редактор связей).

```
double f( int a, bool b ) //  определение функции
{
  double z = .1 * a;
  if( b )
    z += 0.5;
  return z;
}
```

Структура описания функции



При описании функции имя аргумента можно упускать (плохо). При определении функции отсутствие имени аргумента говорит о том, что он передается, но не используется.



Локальные переменные функции или блока создаются **каждый раз**, когда управление доходит до инструкции их инициализации, и уничтожаются при выходе из блока. Статические переменные функции создаются только **один раз**, при первом вызове функции, и уничтожаются при завершении программы — после завершения функции main (сохраняют значение между вызовами функции).

Передача аргументов

В C++ аргументы передаются **по значению** (копии). Но сам тип аргумента может позволить изменять объект в вызывающей функции.

```
void f(      int a,   int *pa,       int &ra  )
//  по значению     по указателю     по ссылке
{
  a++;
  *pa = 4;
  ra += 3;
}
void g()
{
  int x = 0, y = 0, z = 0;
  f( x, &y, z );
  cout << x << ' ' << y << ' ' << z << endl;
  //       0            4            3
}
```

При передаче по ссылке или указателю передаётся копия **адреса** объекта, что позволяет изменять объект.
C++11: rvalue reference: &&

Передача массивов

Массивы передаются как указатели.

```
void f( int *a, const char *s, int n )
{
  a[n-1] = a[0];
  cout << strlen(s);
  s[0] = 0; // <--- это ошибка !!!!!!
}
void g()
{
  int m[4] = { 5, 4, 3, 2 };
  char t[] = "ITS";
  f( m, t, 4 );
}
```

Можно передать массив по значению вместе с объектом, который его содержит.

Передача больших объектов

Передача большого объекта по значению дороже, чем по указателю или ссылке. Если надо запретить изменение — используйте **const**.

```
struct T {
  int u[1024]; // подозрительно...
  char name[512];
};
void f( const T *t )
{
  cout << t->name << endl;
}
void g()
{
  T x1;
  strcpy( x1, "Vasya" );
  f( &x1 );
}
```

Возврат значения

Для возврата значения используется инструкция **return**. Тип выражения после return должен приводится к типу возвращаемого значения.

```
double f( int n )
{
  if( n & 0x0F )
    return 4;
  return 0.1 * n;
}
void g( int n, int m )
{
  if( n > m )
    return;
  cout << m + n << endl;
  // return здесь не обязателен
}
```

Перегруженные функции

Может быть несколько функций с одинаковым именем. Они должны различаться типами аргументов и называются **перегруженными** (overload). Ими реализуется *полиморфизм* уровня функции.

```
void print( int a )
{
  cout << "int: " << a << ' ';
}
void print( const char *s )
{
  cout << "C string: " << s << endl;
}
void g()
{
  print( 5 );
  print( "Horror!" );
  print( 'c' );
}
```

Выбор перегруженной функции – 1 аргумент

Для выбора перегруженной функции анализируются типы передаваемых и требуемых параметров. Если для вызова функции требуется преобразование типа, то уровень соответствия позволяет выбрать наиболее точно соответствующую функцию.

1. Точное соответствие (никакие или неизбежные преобразования) **int**=>**int**, **char**[] => **char***, **int** =>**int**&
2. Повышающие преобразования **short**=>**int**, **float**=>**double**, **char**=>**int**, но не **int**=>**long**
3. Стандартные преобразования **double**=>**int**, **int**=>**double**, T*=>**void***, **int**=>**long**
4. Пользовательские преобразования — определяются пользователем для пользовательских типов.
5. ... – любое кол-во любых аргументов.

Порядок поиска перегруженной функции

Уровни рассматриваются, начиная с 1. Если на уровне 1 функция — она вызывается. Если ни одной — переход на следующий уровень. Если > 1 — неопределенность, ошибка компиляции.

```
void f(int a);
void f(double z);

short s = 1;
f(s); // void f(int a); 2 уровень
f(0.1) // void f(double z); 1 уровень

void g( long a );
void g( double a);
g(5); //  Ошибка, две функции на
//        одном уровне соответствия (3).
g((long)5); // void g( long a ); // (1)
g(5L);      // void g( long a ); // (1)
g(5.);      // void g( double a ); // (1)
```

Значения по умолчанию

У нескольких **последних** аргументов могут быть значения по умолчанию. Если при вызове их не задать явно, то используется значение по умолчанию.

```
void f( int a, int b = 5, int c = 2 );

f(2,3,4); // все задано явно

f(1,2);   // f(1, 2, 2);

f(0);     // f(0, 5, 2)
```

Значения по умолчанию можно задавать только в одном месте (обычно при описании функции). Функция со значениями по умолчанию тоже рассматривается как кандидат при перегрузке.

Встроенные функции

Ключевое слово **inline** рекомендует компилятору встроить эту функцию (для простых функций). При том код, реализующий функцию, вставляется в место её вызова.

```
inline int max( int a, int b )
{
  return ( a > b ) ? a : b;
}
```

Для встраивания компилятору необходимо определение функции, поэтому такие функции часто размещают в заголовочных файлах.
Функции, помеченные как **constexpr**, могут вычисляться на этапе компиляции, и могут использоваться вместо литералов (C++11). В C++11 появились безымянные функции (лямбда-функции).