

# Программирование на C++

## Тема 4. Функции

# Структура программы на языке C(C++)

директивы препроцессора

Описания глобальных переменных и типов данных

// определение функции с именем f1

void f1(список формальных параметров)

{ // тело (операторы) функции f1  
}

// объявление (прототип) функции с именем f2

int f2(список формальных параметров);

// определение главной функции

int main(void)

{ f1(список фактических параметров); // вызов функции f1

int a;

a = f2(список фактических параметров); // вызов функции f2

cout << f2(список фактических параметров); // вызов ф-ции f2

}

// определение функции с именем f2

int f2(список формальных параметров)

{ // операторы функции f1  
return [<выражение>];

}

# Структура программы

- Программа на языке C++ состоит из директив препроцессора, описаний глобальных переменных и типов данных и из функций. Одна из функций (главная функция) должна иметь имя main. С нее начинается выполнение программы.
- Функция в C(C++) - это подпрограмма, т. е. логически заверченный и определенным образом оформленный фрагмент программы, который может быть вызван из других частей программы (других функций).
- Функции подразделяют большие вычислительные задачи на более мелкие и дают возможность пользоваться уже написанными подпрограммами, в том числе и другими разработчиками.
- Если функция написана грамотно и корректно, внутри ее тела оказываются скрытыми несущественные для других частей программы детали ее реализации, что делает программу в целом более ясной, а также облегчает ее отладку и внесение изменений.
- При работе с функциями используются такие конструкции, как объявление (прототип) функции, определение функции и вызов функции. Определение и объявление функции - это конструкция языка программирования для описания функции.
- Определение функции состоит из ее заголовка и тела. Заголовок функции включает: тип возвращаемого значения, имя функции (например, f1, f2, main) и в круглых скобках список формальных параметров. Тело функции – это составной оператор.
- Объявление (прототип) функции состоит из заголовка функции, который заканчивается ‘;’.
- Функции в C++ являются внешними, т. е. внутри одной функции нельзя определять другую функцию.
- Определение вызываемой функции можно разместить в программе как до, так и после определения вызывающей функции.
- Например, функции f1 и f2 вызываются в главной функции. Определение функции f1 находится до определения главной функции программы, а определение функции f2 - после определения главной функции.

## Структура программы

- Если определение вызываемой функции находится после определения вызывающей функции, то вызываемая функция должна быть объявлена перед определением вызывающей функции.
- Функция может вернуть в вызываемую функцию через оператор возврата **return** значение любого из базовых типов, а также может возвращать указатель на массив (или более сложный объект). Например, функция с именем f2 возвращает значение целого типа **int**.
- Операторов **return** в одной функции может быть несколько. Каждый оператор **return** прекращает выполнение функции. Все следующие за ним операторы не будут выполнены. Поэтому операторы **return** чаще всего располагаются в конце функции, в условных операторах или в операторах варианта.
- В функциях с типом возвращаемого **void** (пустой), не возвращающих никакого значения, тоже могут встречаться операторы **return**, но в них не должно быть никакого выражения, только одно слово **return**: **return**;. Такой оператор **return** служит только для немедленного выхода из функции.
- Функция **main()** возвращает целое значение, если оно равно нулю, то ОС считает, что функция завершилась удачно. Стандарт C++ предусматривает, что функция **main()** возвращает 0 по умолчанию, если оператор **return** не использован явно. Некоторые компиляторы требуют завершать функцию **main()** оператором: **return 0**;
- Вызов функции включает имя функции и в круглых скобках список фактических параметров.
- Если функция не возвращает никакого значения (тип возвращаемого **void**), то для ее вызова используется оператор вызова, который включает имя функции и в круглых скобках список фактических параметров (например, функция с именем f1).
- Если функция возвращает значение через оператор возврата **return**, то она вызывается, либо в правой части оператора присваивания, либо в операторе вывода (например, функция с именем f2).

- Варианты синтаксиса объявления функции:

тип имя\_функции (тип имя\_пар-ра1, тип имя\_пар-ра2, ...) ;

или

тип имя\_функции (тип , тип, ...) ;

- Пример:

```
void My_Func (int Par1, float Par2, float Par3) ;
```

или

```
void My_Func (int, float, float) ;
```

## Объявление функций

- Приведенные формы записи объявлений (прототипов) функций эквивалентны, так как компилятор игнорирует имена формальных параметров, обращая внимание только на их тип.
- Это связано с тем, что описание функции с помощью объявления дает возможность компилятору выполнять проверку на соответствие количества и типов фактических и формальных параметров при каждом обращении к функции, а также возможности проводить необходимые преобразования.

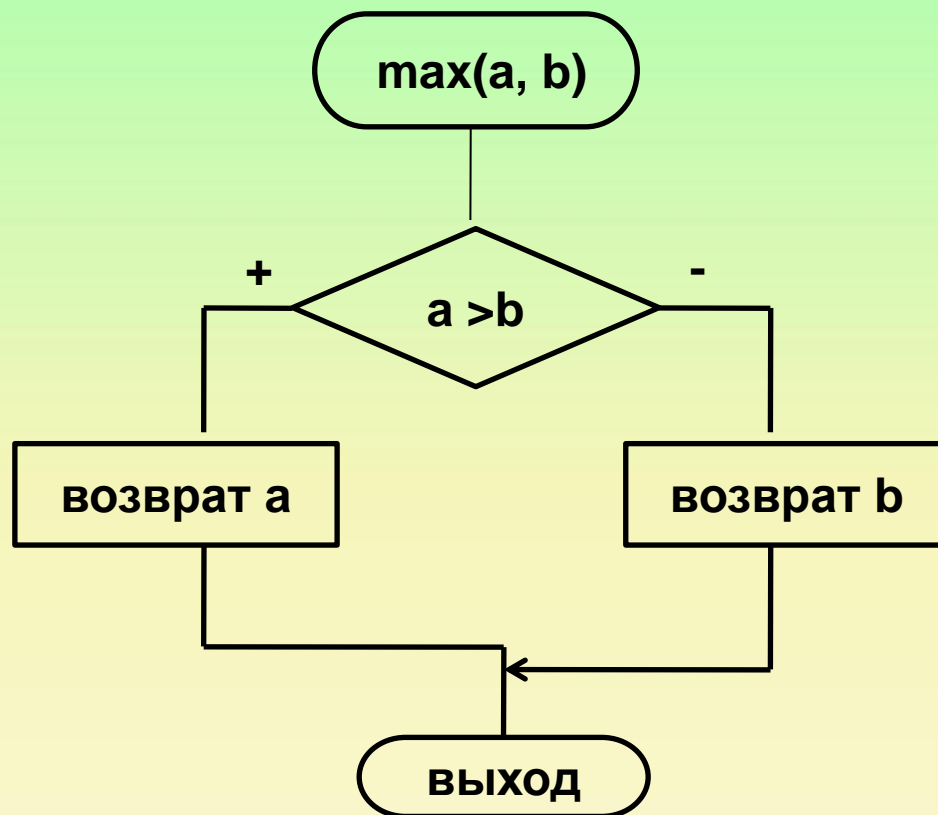
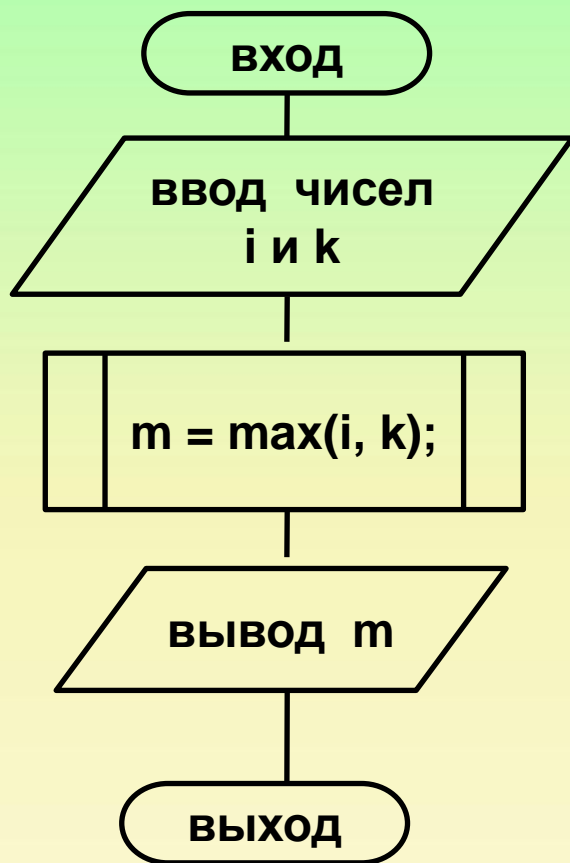
## Задача:

Поиск максимального из двух чисел.

## Постановка задачи:

- *Исходные данные:* целые  $i$ ,  $k$  – числа.
- *Результаты работы программы:* целое число  $m$  – максимальное из чисел  $i$  и  $k$ .

■ Блок – схема алгоритма решения задачи





```
// Поиск максимального из двух чисел
```

```
#include <iostream>
```

```
using namespace std;
```

```
//объявление функции поиска максимального из двух чисел
```

```
int max(int a, int b);
```

```
int main(void)
```

```
{ int m, k, i;
```

```
    cout << "i = "; cin >> i;
```

```
    cout << "k = "; cin >> k;
```

```
    m = max(i, k); // вызов функции
```

```
    cout << "max = " << m << endl;
```

```
}
```

```
//определение функции поиска максимального из двух чисел
```

```
int max(int a, int b)
```

```
{ if (a > b) return a;
```

```
    else return b;
```

```
}
```

```
i = 5
k = 7
max = 7
```

## Понятие об указателях

```
char C = '$'; // будет выделена память под переменную C
              // и ей присвоено начальное значение
cout << C; // из ячейки памяти с именем C будет извлечено
           // значение и выведено на экран
```

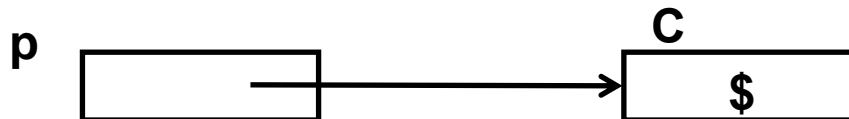
### ■ Синтаксис объявления указателя:

тип\_данных \*имя\_переменной;

■ Пример: float \*x, \*y, \*z;  
          char \*p, ch;

### ■ Пример использования операции получения адреса (&) и операции разыменования (\*)

```
p = &C; //в указатель p записывается адрес переменной C
ch = *p; // в переменную ch записывается символьное
         // значение, хранящееся по адресу p
```



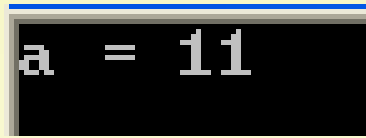
## Понятие об указателях

- Как правило, при обработке оператора описания переменной компилятор автоматически выделяет память под переменную в соответствии с указанным типом. При завершении программы или функции, в которой была описана переменная, память автоматически освобождается.
- Доступ к объявленной переменной осуществляется по ее имени. При этом все обращения к переменной меняются на адрес ячейки памяти, в которой хранится ее значение.
- Доступ к значению переменной можно получить иным способом – определить собственные переменные для хранения адресов памяти. Такие переменные называют указателями.
- Итак, указатель – это переменная, значением которой является адрес памяти, по которому храниться объект определенного типа (другая переменная).
- Как и любая переменная, указатель должен быть объявлен. При объявлении указателей всегда указывается тип переменной, значение которой будет храниться по данному адресу. Звездочка в описании указателя относится непосредственно к имени, поэтому, чтобы объявить несколько указателей, ее ставят перед именем каждого из них.
- При работе с указателями часто используют операции получения адреса (&) и разыменования (\*). Операция получения адреса (&) возвращает адрес своего операнда. Операция разыменования (\*) возвращает значение переменной, хранящееся по заданному адресу, то есть выполняет действие, обратное операции получения адреса (&).
- С помощью указателей можно передавать адреса фактических параметров функциям и адреса функций в качестве параметров, создавать новые переменные в процессе выполнения программы, обрабатывать массивы, строки и структуры.

### ■ Формат описания ссылки:

тип &идентификатор\_1 = идентификатор\_2;

```
#include <iostream>
using namespace std;
int main(void)
{ int a = 5, b = 10;
  int &aRef = a; // aRef является ссылкой на a
  aRef = b; // a равно b
  aRef++; // a++;
  cout << "a = " << a << endl;
}
```



```
a = 11
```

- Ссылка на некоторую переменную может рассматриваться как указатель, который при работе с ним всегда разыменовывается. Для ссылки не требуется дополнительного пространства в памяти: она является просто другим именем или псевдонимом переменной. Для определения ссылки применяется унарный оператор &.
- Ссылка не создает копию объекта, а лишь является другим именем объекта. Чаще всего ссылки используются для передачи параметров в функции.

## Способы передачи параметров

// Поиск максимального из двух чисел

```
#include <iostream>
```

```
using namespace std;
```

// передача параметров по значению

```
int max(int a, int b)
```

```
{ if (a > b) return a;
```

```
    else return b;
```

```
}
```

```
int main(void)
```

```
{ int m, k, i;
```

```
    cout << "i = "; cin >> i;
```

```
    cout << "k = "; cin >> k;
```

```
    m = max(i, k); // вызов функции
```

```
    cout << "max = " << m << endl;
```

```
}
```

```
i = 5
k = 7
max = 7
```

## Способы передачи параметров

- Передача фактических параметров в функцию может происходить:
  - по значению,
  - по адресу.
  
- При передаче в функцию фактического параметра по значению, соответствующий формальный параметр получит копию значения фактического параметра. Функция работая с копией фактического параметра не может изменить оригинальное значение фактического параметра. Это происходит потому, что копия фактического параметра (формальный параметр) создается в стеке. В момент завершения работы функции стековое пространство, в котором находятся формальные параметры функции, очищается, уничтожая находящиеся в нем данные. При этом значение-оригинал фактического параметра оказывается недоступным для функции.
  
- В примере фактические параметры  $i$  и  $k$  передаются в функцию `max` по значению, т. е. формальный параметр `a` получит копию значения фактического параметра  $i$ , а формальный параметр `b` получит копию значения фактического параметра  $k$ .

# Способы передачи параметров

// Обмен значений двух переменных

```
#include <iostream>
```

```
using namespace std;
```

// передача параметров по адресу с использованием указателей

```
void changel(int *a1, int *b1)
```

```
{ int c = *a1;  
  *a1 = *b1;  
  *b1 = c;  
}
```

```
int main(void)
```

```
{ int a, b;  
  cout << " a : "; cin >> a;  
  cout << " b : "; cin >> b;  
  
  changel(&a, &b);  
  
  cout << endl << " a = " << a << endl;  
  cout << " b = " << b << endl;  
}
```

a	:	4
b	:	6
a	=	6
b	=	4



## Способы передачи параметров

- Если функция в процессе своей работы должна изменить значение фактического параметра, то его нужно передать по адресу, с использованием указателя или по ссылке.
- Для передачи в функцию фактического параметра по адресу с использованием указателя необходимо:
  1. Оформить фактический параметр, либо как указатель ранее инициализированный, либо как обычную переменную к которой применена операция прямой адресации (&);
  2. Оформить в заголовке функции соответствующий формальный параметр как указатель;
- При передаче в функцию фактического параметра по адресу с использованием указателя соответствующий формальный параметр получает адрес фактического параметра и в функции можно применить операцию разыменования (\*) формального параметра указателя, получая таким образом доступ к значению фактического параметра и возможность его изменения.
- В примере формальные параметры указатели a1 и b1 получают адреса фактических параметров a и b.

# Способы передачи параметров

// Обмен значений двух переменных

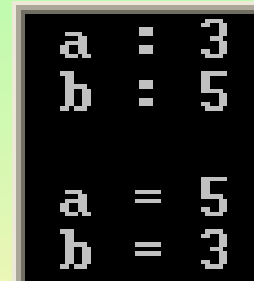
```
#include <iostream>
```

```
using namespace std;
```

// передача параметров по адресу с использованием ссылок

```
void change2(int& a1, int& b1)
```

```
{ int c = a1;  
    a1 = b1;  
    b1 = c;  
}
```



```
a : 3  
b : 5  
  
a = 5  
b = 3
```

```
int main(void)
```

```
{ int a, b;  
    cout << " a : "; cin >> a;  
    cout << " b : "; cin >> b;  
  
    change2(a, b);  
  
    cout << endl << " a = " << a << endl;  
    cout << " b = " << b << endl;  
}
```

## Способы передачи параметров

- Для передачи в функцию фактического параметра по адресу с использованием ссылки необходимо:
  1. Оформить фактический параметр как обычную переменную;
  2. Оформить в заголовке функции соответствующий формальный параметр как ссылку;
- При передаче в функцию фактического параметра по адресу с использованием ссылки соответствующий формальный параметр получает адрес фактического параметра , в функции автоматически выполнятся доступ по адресу к значению фактического параметра и появляется возможность изменения значения фактического параметра .
- В примере формальные параметры ссылки a1 и b1 получают адреса фактических параметров a и b.

## Пример 1.

## Вычисление факториала

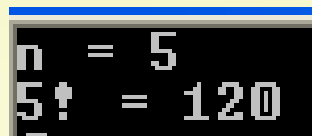
$$n! = \begin{cases} n*(n-1)!, & \text{если } n > 1 \\ 1, & \text{если } n \leq 1 \end{cases}$$

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1, \quad 0! = 1, \quad 1! = 1$$

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

long int fact(int n1)
{ if (n1 <= 1) return 1;
  else
    return (n1*fact(n1-1));
}

int main(void)
{ int n;
  cout << "n = "; cin >> n;
  cout << n << "! = " <<
fact(n) << "\n";
}
```



```
n = 5
5! = 120
```

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

long int fact(int n1)
{ long int f = 1;
  for(int i = n1; i >= 1; i--)
    f = f*i;
  return f;
}

int main(void)
{ int n;
  cout << "n = "; cin >> n;
  cout << n << "! = " <<
fact(n) << "\n";
}
```

# Рекурсивные функции

- Рекурсивной называется функция, вызывающая сама себя. Рекурсивные функции, которые прямо вызывают сами себя, называются прямо рекурсивными. Если две функции рекурсивно вызывают друг друга, они называются косвенно рекурсивными.
- Рекурсивные функции чаще всего используют для компактной реализации рекурсивных алгоритмов. Например, классическими рекурсивными алгоритмами могут быть вычисление факториала (пример 1), вычисление чисел Фибоначчи (пример 2), возведение числа в целую положительную степень.
- Любой рекурсивный алгоритм можно реализовать без применения рекурсии. Достоинством рекурсии является компактная запись, а недостатком – расход памяти на повторные вызовы функций и передачу параметров, кроме того, существует опасность переполнения памяти.
- Последовательность чисел Фибоначчи (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, .....): нулевой элемент последовательности равен нулю, первый – единице, а каждый последующий представляет собой сумму двух предыдущих.

**Пример 2.** Вычислить n-е число Фибоначчи. Последовательность чисел Фибоначчи (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, .....).

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

```
long int fib(unsigned int n1)
{ if ((n1 == 0) || (n1 == 1))
    return n1;
  else return (fib(n1-1) + fib(n1-2));
}
```

```
int main(void)
{ int n;
  cout << "Введите число n: "; cin >> n;
  cout << n << " - е число Фибоначчи = " << fib(n) << "\n";
}
```

```
Введите число n: 8
8 - е число Фибоначчи = 21
```

## Встраиваемые функции

Пример. Встраиваемая функция, определяет четность числа.

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

inline int even(int x)
{
    return !(x % 2);
}
```

```
int main(void)
{ int dig;
  cout << "Введите число: ";
  cin >> dig;
  if (even(dig))
    cout << dig << " - четное" << endl;
  else
    cout << dig << " - нечетное" << endl;
}
```

```
Введите число: 5
5 - нечетное
```

```
Введите число: 8
8 - четное
```

## Встраиваемые функции

- Вызов функции всегда сопровождается дополнительными действиями по обращению к функциям, передачей параметров через стек, передачей возвращаемого значения. Все это ухудшает характеристики программы, в частности ее быстродействие.
- Но C++ позволяет задавать функции, которые не вызываются, а встраиваются в программу непосредственно в месте ее вызова. В результате этого в программе создается столько копий встраиваемой функции, сколько раз к ней обращалась вызывающая программа.
- Использование встраиваемых функций не связано с механизмами вызова и возврата, следовательно, работают они гораздо быстрее обычных. Однако они способны значительно увеличить объем программного кода.
- Для объявления встраиваемой функции указывают спецификатор inline перед определением функции.
- Спецификатор inline является не командой для компилятора, а только запросом. Поэтому, если компилятор по каким-либо причинам не может встроить функцию, она компилируется как обычная и никаких сообщений об этом на экран не выдается.
- Некоторые компиляторы не могут сделать функцию встраиваемой, если:
  - функция содержит операторы цикла,
  - функция содержит операторы switch или goto,
  - функция рекурсивная.
- Реально компилятор может проигнорировать объявление inline, если сочтет функцию слишком большой, вследствие чего характеристики программы могут оказаться хуже, чем в случае вызываемой функции.



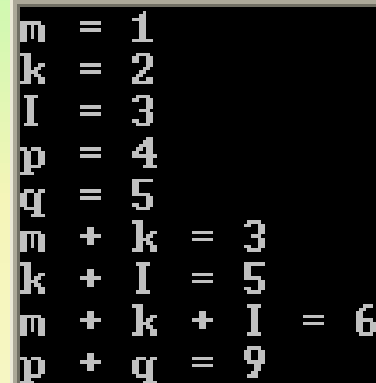
## Перегрузка функций

**Пример. Перегруженная функция суммирования: двух целых чисел, трех целых чисел, двух вещественных чисел.**

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

int sum(int a, int b) {return a + b;}
int sum(int a, int b, int c) {return a + b + c;}
float sum(float a, float b) {return a + b;}

int main(void)
{ int m, k, I; float p, q;
  cout << "m = "; cin >> m;
  cout << "k = "; cin >> k;
  cout << "I = "; cin >> I;
  cout << "p = "; cin >> p;
  cout << "q = "; cin >> q;
  cout << "m + k = " << sum(m, k) << endl;
  cout << "k + I = " << sum(k, I) << endl;
  cout << "m + k + I = " << sum(m, k, I) << endl;
  cout << "p + q = " << sum(p, q) << endl;
}
```



```
m = 1
k = 2
I = 3
p = 4
q = 5
m + k = 3
k + I = 5
m + k + I = 6
p + q = 9
```

## Перегрузка функций

- В программах, написанных на языке C++, возможно существование нескольких различных функций, имеющих одно имя. При этом одноименные функции обязательно должны отличаться числом и/или типом своих аргументов. Такие функции называются перегруженными.
- Для того чтобы перегрузить функцию, необходимо задать все требуемые варианты ее реализации. Компилятор автоматически выбирает правильный вариант вызова согласно числу и типу аргументов. Тип возвращаемого значения при перегрузке большой роли не играет.
- Перегруженные функции позволяют упростить программу, допуская обращение к одному имени для выполнения близких по смыслу (но, возможно, алгоритмически различных) действий.
- Возможность существования в одной программе нескольких одноименных функций является одним из проявлений полиморфизма. Поскольку выбор нужного варианта происходит на этапе компиляции программы, то это - статический полиморфизм.
- Возможность использования одинаковых идентификаторов для выполнения близких по смыслу, но, возможно, алгоритмически различных действий называется полиморфизмом.
- Функции не могут быть перегружены, если
  1. Они отличаются только типом возвращаемого значения. Попытки перегрузить функции таким образом приводят к появлению ошибки компиляции «Type mismatch in redeclaration». Это происходит потому, что компилятор принимает во внимание только аргументы функций.
  2. Их аргументы отличаются только использованием ссылок.
  3. Их аргументы отличаются только применением модификаторов const.

## Использование аргументов по умолчанию

```
■ void Fn (int arg1 = 0, int arg2 = 1000);  
void Fn (int = 0, int = 1000);
```

```
■ Fn();           //arg1 = 0, arg2 = 1000  
Fn(10);          //arg1 = 10, arg2=1000  
Fn(10,99);       // arg1 = 10, arg2 = 99
```

Пример. Функция возвращает произведение четырех целых чисел, из которых по умолчанию может быть задано от нуля до четырех значений.

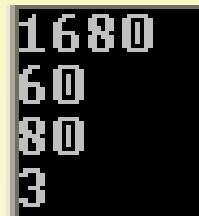
```
#include <iostream>
```

```
using namespace std;
```

```
int multiple(int m1 = 1, int m2 = 1, int m3 = 1, int m4 = 1)  
{ return m1 * m2 * m3 * m4; }
```

```
int main(void)
```

```
{ cout << multiple(5,6,7,8)<< endl;  
  cout << multiple(3,4,5) << endl;  
  cout << multiple(8,10) << endl;  
  cout << multiple(3) << endl;  
}
```



```
1680  
60  
80  
3
```

## Использование аргументов по умолчанию

- Применение аргументов по умолчанию представляет собой самую простую форму перегрузки функций. При этом один или несколько аргументов функции при ее вызове могут отсутствовать, тогда им передается значение по умолчанию. Для того чтобы задать аргументу значение по умолчанию, нужно в заголовке функции поставить после этого аргумента знак равенства и значение, которое будет передаваться по умолчанию:
- Если при вызове функции часть аргументов отсутствует, то компилятор считает, что эти аргументы находятся в конце списка аргументов, т. е. нельзя задать явно второй аргумент, задав первый по умолчанию.
- В общем случае все параметры по умолчанию должны находиться правее параметров, задаваемых явно.
- Аргументы по умолчанию должны быть константами или глобальными переменными.
- При создании функции с аргументами по умолчанию их следует задавать единожды: либо в прототипе функции, либо в заголовке ее определения, но не вместе.
- Если значение по умолчанию не задано, а соответствующий аргумент при вызове функции отсутствует, то это вызывает ошибки компиляции.

## Указатель на функцию

### ■ Определение указателя на функцию:

тип (\*имя\_указателя) (типы параметров функции)

### ■ Вызов функции по указателю:

(\*имя\_указателя) (список аргументов)

```
#include <iostream>
```

```
using namespace std;
```

```
//суммирование двух чисел
```

```
float sum(float a, int b)
```

```
{ return a + b;}
```

```
int main(void)
```

```
{ float (*ptr)(float, int); // описание указателя на функцию
```

```
ptr = sum; // присвоение указателю адреса функции
```

```
float x = sum(0.5, 1); // вызов функции по имени
```

```
float y = (*ptr)(1.5, 2); // вызов функции по указателю
```

```
cout << "x = " << x << ' ' << "y = " << y << endl;
```

```
}
```

```
x = 1.5 y = 3.5
```

- В определении указателя на функцию количество и тип параметров должны совпадать с соответствующими типами в определении функции, на которую ставится указатель.
- Присвоение указателю адреса функции не требует операции взятия адреса (&), поскольку имя функции само по себе обозначает этот адрес.

## Указатель на функцию как параметр функции

```
// Нахождение корня нелинейного уравнения методом деления
// пополам
#include <iostream>
#include <math.h>
using namespace std;

double func(double x);
double bisection(double a, double b, double eps,
                 double (*f)(double))
{ double c, y;
  do { c = 0.5*(a+b); y = f(c);
    if (fabs(y) < eps) break; //корень найден. Выход из цикла
    // Если на концах отрезка [a, c] функция имеет разные знаки
    if (f(a)*y < 0.0) b = c; // значит, корень здесь.
                                // Переносим точку b в точку c
    // В противном случае:
    else a = c; // переносим точку a в точку c
    // Продолжаем, пока отрезок [a, b] не станет мал
  }while(fabs(b-a) >= eps);
  return c;
}
```

## Указатель на функцию как параметр функции

```
int main(void)
{ double a = 0.0, b = 1.5, eps = 1e-5;
  cout << bisect(a, b,eps,func)<< endl;
}
```

```
double func(double x)
{ return x*x*x - 3*x*x + 3; }
```



1.3473\_



## Указатель на функцию как параметр функции

## Пример программы, состоящей из двух модулей

```
// Заданы координаты сторон треугольника, найти его площадь

// Основной модуль

#include <iostream>
#include "function.h"
using namespace std;

int main(void)
{
    double x1 = 2, y1 = 3, x2 = 4, y2 = 6, // координаты
           x3 = 7, y3 = 9;                // сторон треугольника
    double line1, line2, line3; // длины сторон треугольника
    // вызов функции вычисления длин сторон треугольника
    line1 = line(x1, y1, x2, y2);
    line2 = line(x1, y1, x3, y3);
    line3 = line(x2, y2, x3, y3);
    // вызов функции вычисления площади треугольника
    cout << "S = " << square(line1, line2, line3) << endl;
}
```

## Пример программы, состоящей из двух модулей

```
// Файл function.h
```

```
// Объявления функций
```

```
double line(double x1, double y1, double x2, double y2);
```

```
double square(double a, double b, double c);
```

```
// Файл function.cpp
```

```
// Определения функций
```

```
#include <math.h>
```

```
double line(double x1, double y1, double x2, double y2)
```

```
{
```

```
    return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
```

```
}
```

```
double square(double a, double b, double c)
```

```
{ double s, p = (a + b + c) / 2;
```

```
    return s = sqrt(p * (p - a) * (p - b) * (p - c)); // формула Герона
```

```
}
```

- При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут оформляться в виде отдельных модулей.
- При разработке программ, которые состоят из большого количества функций, размещенных в разных модулях, прототипы функций и описания внешних объектов (констант, переменных, массивов) помещают в отдельный файл, который включают в начало каждого из модулей программы с помощью директивы `#include "имя_файла"`.

## Контрольные вопросы

1. Понятие и назначение функции в C(C++).
2. Объявление и определение функций: понятие, назначение, синтаксис записи, примеры. Формальные параметры функций: понятие, синтаксис записи.
3. Понятие и синтаксис объявления указателя в C++. Что понимается под инициализацией и разыменовыванием указателя?
4. Понятие, назначение и синтаксис описания ссылок в C++.
5. Вызов функции: синтаксис записи, примеры различных вариантов вызова функций. Фактические параметры функций: понятие, синтаксис записи. Дайте характеристику способов передачи параметров в функцию.
6. Как в C/C++ оформляются программы, состоящие из нескольких модулей?
7. Понятие, достоинства и недостатки рекурсивных функций.
8. Понятие, назначение и синтаксис оформления встраиваемых функций и функций с аргументами по умолчанию.
9. Понятие и назначение перегрузки функций в C++.
10. Как определяется указатель на функцию? Когда удобно использовать указатель на функцию?