Функции

Объявление функции (прототип, заголовок, сигнатура) задает ее имя, тип возвращаемого значения и список передаваемых параметров. ***Определение функции*** содержит, кроме объявления, ***тело*** функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )]
{ тело функции }
```

класс = [extern | static]

Пример функции, возвращающей сумму двух целых величин:

```
#include <iostream.h>
int sum(int a, int b);       // объявление функции
int main(){
   int a = 2, b = 3, c, d;
   c = sum(a, b);            // вызов функции
   cin >> d;
   cout << sum(c, d);        // вызов функции
   return 0;
}
int sum(int a, int b){              // определение функции
   return (a + b);
}
```

Использование модификатора типа переменной static для сохранения ее величины между вызовами функции:

```
#include <iostream.h>
void f(int a){
   int m = 0;
   cout << "n m p\n";
   while (a--){
      static int n = 0;
      int p = 0;
      cout << n++ << ' ' << m++ << ' ' << p++ << '\n';
   }
}
int main(){ f(3); f(2); return 0;}
```

Программа выведет на экран:

```
n m p
0 0 0
1 1 0
2 2 0
n m p
3 0 0
4 1 0
```

Возвращаемое значение функции:

```
int f1(){return 1;}   // правильно
void f2(){return 1;}  // неправильно, f2 не должна возвращать значение
double f3(){return 1;}// правильно, 1 преобразуется к типу double
```

Передача параметров функции

При передаче по значению в стек заносятся копии значений аргументов, и операторы функции работают с этими копиями. Доступа к исходным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить.

При передаче по адресу в стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов:

При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются. Поэтому использование ссылок вместо указателей улучшает читаемость программы, избавляя от необходимости применять операции получения адреса и разыменования.

```
#include <iostream.h>
void f(int i, int* j, int& k);
int main(){
   int i = 1, j = 2, k = 3;
   cout <<"i j k\n";
   cout << i <<' '<< j <<' '<< k <<'\n';
   f(i, &j, k);
   cout << i <<' '<< j <<' '<< k;
   return 0;
}
void f(int i, int* j, int& k){
   i++; (*j)++; k++;
}
```

Результат работы программы:

i	j	k
1	2	3
1	3	4

Передача массива в качестве параметра.
В функцию передается указатель на первый элемент массива, то есть массив всегда передается по адресу. Размерность массива следует передавать через отдельный целый параметр функции.

Пример функции подсчитывающей сумму элементов массива:

```
#include <iostream.h>
int sum(const int* mas, const int n);
int const n = 10;
int main(){
   int marks[n] = {3, 4, 5, 4, 4};
   cout << "Сумма элементов массива: " << sum(marks, n);
   return 0;
}
int sum(const int* mas, const int n){
   // варианты: int sum(int mas[], int n)
   // или       int sum(int mas[n], int n)
   // (величина n должна быть константой)
   int s = 0;
   for (int i = 0; i<n; i++) s += mas[i];
   return s;
}
```

Передача имен функции в качестве параметров:

```
void f(int a ){ /* ... */ } // определение функции
void (*pf)(int);            // указатель на функцию
…
pf = &f;                    // указателю присваивается адрес функции
                            // (можно написать pf = f;)
pf(10);                     // функция f вызывается через указатель pf
                            // (можно написать (*pf)(10) )
```

Функции со значениями по умолчанию:

```
int f(int a, int b = 0);
void f1(int, int = 100, char* = 0);   /* обратите внимание на пробел между * и
= (без него получилась бы операция сложного присваивания *=) */
void err(int errValue = errno);          // errno – глобальная переменная
…
f(100); f(a, 1);                         // варианты вызова функции f
f1(a); f1(a, 10); f1(a, 10, "Vasia");    // варианты вызова функции f1
f1(a,,"Vasia")                           // неверно!
```

Функции с переменным числом параметров.

Тип данных va_list и макросы va_start ( ); va_arg( ) и va_end( ) для работы с переменным числом параметров определены в заголовочном файле stdarg.h

Прием функции. подсчитывающей сумму всех своих аргументов

```
void sum(int a, …);
{
  va_list args;
  int result = a, t;
  va_start(args, a);
  while(( t = va_args(args, int) != 0)
        result += t;
  va_end(args);
  return result;
}
```

Рекурсивные функции

```
long fact(long n){
   if (n==0 || n==1) return 1;
   return (n * fact(n - 1));
}
```

То же самое можно записать короче:

```
long fact(long n){
   return (n>1) ? n * fact(n - 1) : 1;
}
```

Перегрузка функций

```
// Возвращает наибольшее из двух целых:
int max(int, int);
// Возвращает подстроку наибольшей длины:
char* max(char*, char*);
// Возвращает наибольшее из первого параметра и длины второго:
int max (int, char*);
// Возвращает наибольшее из второго параметра и длины первого:
int max (char*, int);
void f(int a, int b, char* c, char* d){
   cout << max (a, b) << max(c, d) << max(a, c) << max(c, b);
}
```

Шаблоны функций

Шаблоны позволяют определить алгоритм, который применяется к разным типам данных, а конкретный тип передается функции на этапе компиляции. template <class Type> заголовок { /* тело функции */ }. Пример:

```
template <class Type>
   void sort_vybor(Type *b, int n){
   Type a;          //буферная переменная для обмена элементов
   for (int i = 0; i<n-1; i++){
      int imin = i;
      for (int j = i + 1; j<n; j++)
         if (b[j] < b[imin]) imin = j;
         a = b[i]; b[i] = b[imin]; b[imin] = a;
      }
   }
```