# 15. ФУНКЦИИ

## 15.1. Определение и вызов функции

Программа C++ состоит из одной или нескольких функций. Функции разбивают большие задачи на маленькие подзадачи.

Имя одной из функций (**main)** является зарезервированным. Эта функция обязательно должна присутствовать в любой программе. Функция main необязательно должна быть первой, хотя именно с нее начинается выполнение программы.

Функция не может быть определена в другой функции.

С использованием функции связаны 3 понятия – определение функции, объявление функции и вызов функции. Определение функции имеет вид

**тип имя ( список описаний аргументов )**{ **операторы** }

Здесь **имя** – это имя функции; **тип** – тип возвращаемого функцией значения; **операторы** в фигурных скобках { } часто называют телом функции.

Аргументы в списке описаний называют **формальными параметрами**.

Например, функцию, находящую и возвращающую максимальное значение из двух целых величин a и b, можно определить так:

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

Это определение говорит о том, что функция с именем **max** имеет два целых аргумента и возвращает целое значение. Если функция действительно должна возвращать значение какого-либо типа, то в ее теле обязательно должен присутствовать оператор return **выражение**; при выполнении этого оператора выполнение функции прекращается, управление передается в функцию, вызывающую данную функцию, а значением функции будет значение **выражения**. Напишем программу:

```
#include <iostream>
int max (int a, int b){
return a >= b ? a: b;
```

```
}
void main( ){
int i = 2, j = 3; int c = max( i, j );
cout << " max= " << c << '\n'; c
= max( i * i, j ) * max( 5, i - j );
cout << " max= " << c << '\n'; }
```

В этой программе приведены определение функции max и 3 обращения к ней. При обращении (или вызове) указывается имя функции и в круглых скобках список **фактических параметров**.

Если у функции нет формальных параметров, то она определяется, например, так:

```
double f(void){тело функции}
или, эквивалентно,  double f(
){тело функции}
```

Вызывают в программе подобную функцию, например, так: a
= b * f( ) + c;

Функция может и не возвращать никакого значения. В этом случае ее определение таково:

void **имя (список описаний аргументов)**{ **операторы** }

Вызов такой функции имеет вид **имя**

**(список фактических аргументов);**

Выполнение функции, не возвращающей никакого значения, прекращается оператором return без следующего за ним выражения. Выполнение такой функции и возврат из нее в вызывающую функцию происходит также и в случае, если при выполнении тела функции произошел переход на самую последнюю закрывающую фигурную скобку этой функции.

В качестве примера приведем функцию, копирующую одну строку в другую:

```
void copy(char* to, char* from){ while(
*to++ = *from++ );
} void main( ){ char
str1[ ] = "string1"; char
str2[ ] = "string2";
```

```
copy (str2, str1); cout
<< str2 << "\n";
}
```

## 15.2. Передача аргументов в функцию

В приведенных выше примерах происходит так называемая передача аргументов ***по значению***. Такая передача аргументов означает, что в вызываемой функции для каждого формального аргумента создаётся локальный объект, который инициализируется значением фактического аргумента. Следовательно, при такой передаче изменение значений формальных параметров функции не приводит к изменению значений соответствующих им фактических аргументов.

Рассмотрим, например, вариант функции, возводящей целое **x** в целую степень **n**, где используется это обстоятельство.

```
int power (int x, int n){
for (int p = 1; n > 0; n --)
p* = x; return p; }
```

Аргумент **n** используется как временная переменная. Что бы ни происходило с **n** внутри функции power, это никак не влияет на фактический аргумент, с которым первоначально обратились к этой функции в вызываемой функции:

```
void main( ){
…
int n = 6,  x = 3; x = power(x, n);        //
n – не меняется.
…
}
```

Рассмотрим процесс вызова функции более подробно. При вызове функции:

- в стеке резервируется место для формальных параметров, в которые записываются значения фактических параметров; обычно это производится в порядке, обратном их следованию в списке;
- в стек записывается точка возврата – адрес той части программы, где находится вызов функции;
- в начале тела функции в стеке резервируется место для локальных (автоматических) переменных.

В случае, если функция должна менять свои аргументы, можно использовать указатели. Указатели также передаются по значению, внутри функции создается локальная переменная – указатель. Но так как

этот указатель инициализируется адресом переменной из вызываемой программы, то эту переменную можно менять, используя этот адрес.

В качестве примера рассмотрим функцию, меняющую местами свои аргументы: void swap(int* x, int* y){

int t = *x; *x
= *y;
*y = t; }

Обратиться к этой функции можно так:

int a = 3, b = 7; swap(&a,
&b);

Теперь a = 7, и b = 3.

Некоторую особенность имеет использование массивов в качестве аргументов. Эта особенность заключается в том, что имя массива преобразуется к указателю на его первый элемент, то есть при передаче массива происходит передача указателя. По этой причине вызываемая функция не может отличить, относится ли передаваемый ей указатель к началу массива или к одному единственному объекту.

```
int summa(int array [ ], int size){
int res = 0;
for (int i = 0; i <size; i++) res += array[i];
return res; }
```

В заголовке int array [ ] можно заменить на int* array, а выражение в теле функции array[i] заменить на *(array + i) или даже на *array++, так как array не является именем массива и, следовательно, не является константным указателем.

К функции summa можно обратиться так: int
mas [100];
for (int i = 0; i < 100; i++)mas[i] = 2 * i + 1; int
j = summa(mas, 100);

Пример: вычисление многочлена по его коэффициентам.

Пусть требуется вычислить многочлены

$$P\ x_3(\ ) \square\ 4x^3 \square\ 2x^2 \square 1,$$

$$P\ x_5(\ ) \square\ \square\ \square\ \square\ \square\ \square x^5\ x^4\ x^3\ x^2\ x\ 7, \qquad (1)$$

$$P\ x_9(\ ) \square\ \square x^9\ \ 2x^7 \square 3x^6 \square\ \square\ \square x^5\ x^2\ \ 2$$

в точке x = 0.6.

Ввиду важности вычисления многочленов составим функцию, осуществляющую вычисление многочлена степени $n$

$P(x) = C_0 + C_1 x + C_2 x^2 + \ldots + C_n x^n$ по его коэффициентам $C_i$. Для эффективного вычисления многочлена используем так называемую схему Горнера (которую на самом деле за 100 лет до Горнера описал Ньютон). Эта схема состоит в том, что многочлен переписывается в следующей форме:

$$P(x_n) = (\ldots((0 + x C_n) + x C_{n-1}) + x C_{n-2}) + \ldots + x C_1) + C_0.$$

При таком порядке вычислений для получения значения $P(x_n)$ требуется лишь $n$ умножений и $n$ сложений.

Коэффициенты многочленов будем хранить в массиве **c**.

Для вычисления многочленов (1) напишем программу, в которой схема Горнера реализована в функции pol().

```
const int n = 10;                    // Число коэффициентов многочлена
double pol (int n, double c[ ], double x){ //
Вычисление многочлена степени n-1
double p = 0; for (int i =
n; i >= 0; i --)
p = p * x + c[i];
return p; } void
main( ){
double x = 0.6, p, c[n] = {1, 0, 2, 4}; p
= pol(3, c, x);
cout << "x= " << x << "  Polynom = " << p << "\n"; c[0]
= 7;
c[1] = c[2] = c[3] = c[4] = c[5] = 1; p
= pol(5, c, x);
cout << "x= " << x << "  Polynom = " << p << "\n";
c[0] = 0.5;   c[1] = -2.3;  c[2] = 1;      c[3] = 1.5;
c[4] = 3;      c[5] = 2;
c[6] = 0;      c[7] = 0.35; c[8] =
c[9] = 0;
cout << "x= " << x << "  Polynom = " << pol(n-1, c, x) << "\n"; }
```

## 15.3. Передача многомерных массивов

Если функции передаётся двумерный массив, то описание соответствующего аргумента функции должно содержать число столбцов в этом массиве; количество строк – несущественно, поскольку фактически передаётся указатель.

Рассмотрим пример функции, перемножающей матрицы A и B; результат – матрица C.

Размер матриц – не более 10.

```
const int nmax = 5; void product (int a[ ][nmax], int b[
][nmax], int c[ ][nmax],  int m, int n, int k){
/* m –  число строк в матрице a; n –  число строк в матрице b (должно
   быть равно числу столбцов в матрице a); k –  число столбцов в
   матрице b.*/
for (int i = 0; i < m; i++) for
(int j = 0; j < k; j++){ c[i][j]
= 0;
for (int l = 0; l < n; l++) c[i][j] += a[i][l] * b[l][j];
}
}
```

Если заданы, например, квадратные матрицы A и B размером 5 5 , то получить их произведение C можно так:

```
product(A, B, C, 5, 5, 5);
```

В приведённом примере есть недостаток – здесь заранее фиксируется максимальная размерность матриц. Обойти это можно различными способами, один из которых – использование вспомогательных массивов указателей на массивы.

Напишем функцию, транспонирующую квадратную матрицу произвольной размерности n.

```
void trans(int n, double* p[ ])
{double x;
for (int i = 0; i < n-1; i++) for
(int j = i + 1; j < n; j++)
{x = p[i][j]; p[i][j] = p[j][i]; p[j][i] = x;
}
}
```

```
void main(){ double A[4][4] = {
10, 12, 14, 17,
                    15, 13, 11, 0,
                    -3, 5.1, 6, 6, 2,
                    8, 3, 1};
double ptr[ ] = {(double*)&A[0], (double*)&A[1],
                 (double*)&A[2], (double*)&A[3]};
int n = 4; trans
(n, ptr);
for (int i = 0; i < n; i++){ count << "\n строка" << (i + 1) << ":";
      for (int j; j < n; j++) cout << "\t" << A[i][j]; cout << "\n";
}
}
```

В функции main матрица представлена двумерным массивом double A[4][4]. Такой массив нельзя непосредственно использовать в качестве фактического параметра, соответствующего формальному double *p[]. Поэтому вводится дополнительный вспомогательный массив указателей double *ptr[]. В качестве начальных значений элементам этого массива присваиваются адреса строк матрицы, преобразованные к типу double*.

Многомерный массив с переменными размерами может быть динамически сформирован внутри функции. В вызываемую функцию его можно передать как указатель на одномерный массив указателей на одномерные массивы с элементами известной размерности и заданного типа.

В качестве примера приведем функцию, формирующую единичную матрицу порядка n.

```
int** singl (int n){ int **p = new
(std::nothrow)int*[n];
/* Тип int* [n] – массив указателей на целые.
```

Операция new возвращает указатель на выделенную память под этот массив. Тип переменной p есть int**. Таким образом, p есть массив указателей на строки целых будущей матрицы. */ if(p == NULL){

```
cout<<"Динамический массив не создан!\n";
exit(1); }
// цикл создания одномерных массивов – строк матрицы:
for (int i= 0; i < n; i++){ p[i] =
new(std::nothrow)int[n];
```

```
if( !p[i] ){cout<<"Не создана динамическая строка!\n";  exit(1);}
for (int j = 0; j < n; j++) p[i][j] = (i == j) ? 1: 0;
} return p; }
void main(){
int n;
cout << "\n Задайте порядок матрицы:  "; cin
>> n;

int** matr;            // Указатель для формируемой матрицы matr =
singl(n);
for(int i = 0; i < n; i++){cout<<"\n строка";
cout.width(2); cout << i + 1 << ": "; for (int
j = 0; j < n; j++){
cout.width(4);  cout
<< matr[i][j];
} }
for(i= 0; i < n; i++) delete []matr[i];
delete []matr; }
```

В этой программе обращение к функции cout.width(**k**) устанавливает ширину поля следующего вывода на печать в **k** позиций, что позволяет произвести выравнивание таблицы.

## 15.4. Указатели на функции

Указатель на функцию определим следующим образом:

**Тип_функции (*имя_указателя) (список параметров);** Например,

```
int(*fptr)(double);
```

Здесь определяется fptr как указатель на функцию с одним аргументом типа double, которая возвращает значение int. Имя функции без следующих за ним ( ) – это указатель на функцию, который содержит адрес начала кода этой функции.

Пример:

```
void f1(void){
cout << "\n Выполняется f1().";
}            void
f2(void){
```

```
cout<<"\n Выполняется f2().";}
void main(){ void (*ptr)(void);
ptr = f2;
(*ptr)();   ptr  //вызов функции f2();
= f1;
(*ptr)();        //вызов f1();
ptr();   }       //альтернативная запись!
```

Результат:

```
Выполняется f2().
Выполняется f1().
Выполняется f1().
```

Иногда удобно использовать формальные параметры функции, являющиеся указателями на функции.

Проиллюстрируем это при решении следующей задачи.

Вычислить интеграл, используя метод трапеций, от двух разных функций.

```
// Файл TRAP.CPP
double trap(double (*func)(double), double a, double b, int n){
double x, h = (b - a)/n, i =(func(a) + func(b))/2; for (x = a + h;
x < b - h/2; x += h) i += func(x); return h * i;
}
// Файл INTEGRAL.CPP
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include "trap.cpp"  double
f1(double x){ return x * x
+ sin(3 + x);} double
f2(double x){
return x/(x * x + 1) + exp(-2 * x *x);}
void main( ){ double a = 1,
b = 3; double i = trap(f1, a,
b, 50);
cout << "Интеграл от первой функции = " << i<< "\n"; i
= trap(f2, a, b, 50);
cout << "Интеграл от второй функции = " << i << "\n";
}
```

## 15.5. Ссылки

Тип «ссылка на **тип**» (ссылка *l*-value) определяется так: **тип**&.

Например, int&

**или** double&

Ссылочные типы устанавливают псевдонимы объектов. Ссылка обязательно должна быть инициализирована. После инициализации использование ссылки дает тот же результат, что и прямое использование переименованного объекта.

Рассмотрим инициализацию ссылки:

```
int i = 0; int& iref = i;
```

Здесь создана новая переменная типа ссылка на int с именем iref. Физически iref – это **постоянный** указатель на int (переменная типа int* const), следовательно, значение ссылки после инициализации не может быть изменено. Инициализирующим значением в нашем случае является адрес переменной i, т. е. при инициализации ссылка ведёт себя как указатель.

При использовании ссылка ведёт себя не как указатель, а как переменная, адресом которой она инициализирована:

```
iref++;                 // то же, что и i++; int
*ip = &iref; // то же, что и ip = &i.
```

Итак, iref стало другим именем, псевдонимом переменной i. Ссылку можно определить так:

*ссылка есть такой константный указатель на объект, к которому всегда при его использовании неявно применяется операция разыменования указателя* *.

Ссылка может быть объявлена константной (ссылка на константу). Такая константная ссылка может быть инициализирована объектом другого типа, или даже безадресной величиной – такой, как константа или выражение (*r*-value).

Например, double

```
d = 2.71828;
// далее – верно только для константных ссылок:
const int & irtf1 = 888; const int & iref2 = d; const
double & dr = d - 8.0;
```

Здесь использование спецификатора const обязательно; без этого все три определения приведут к ошибке компиляции.

## 15.6. Ссылки в качестве параметров функций

Ссылки часто используются в качестве формальных параметров функций. Механизм передачи параметров в функции с помощью ссылок называют в программировании передачей аргументов ***по ссылке***. С помощью ссылок можно добиться изменения значений фактических параметров из вызывающей программы (без применения указателей).

```
void swap(int & x, int & y){
int t = x;
x = y; y
= t; }
```

Теперь обращение в вызывающей функции будет иметь вид

```
int a = 8, b = 32;  swap(a, b);
```

Здесь создаются локальные относительно функции swap() переменные x и y ссылочного типа, которые являются псевдонимами переменных a и b и инициализируются переменными a, b. После этого все действия с x и y эквивалентны действиям с a и b, что приводит к изменению значений a и b.

Ссылки в качестве параметров функций часто применяются с целью сокращения накладных расходов на копирование аргументов (как при передаче аргументов по значению). Если при этом требуется защитить фактический аргумент от изменения, то нужно использовать константную ссылку.

Константная ссылка в качестве параметра дает возможность передавать неконстантный аргумент *l*-value, константный аргумент *l*-value или результат выражения: void funk(const int & arg){ cout << arg << endl;

```
}
int main( ){ int
x = 8;
funk (x);              // x – неконстантное l-value
const int z = 16;
funk(z);               // x – константное l-value;
funk(-7);              // константа в качестве r-value;
funk(7 - z);   return // выражение в качестве r-value;
0;
}
```

## 15.7. Рекурсивные функции

Ситуацию, когда функция тем или иным образом вызывает саму себя, называют рекурсией. Рекурсия, когда функция обращается сама к себе непосредственно, называется **прямой**; в противном случае она называется **косвенной**.

Все функции языка C++ (кроме функции main) могут быть использованы для построения рекурсии.

В рекурсивной функции обязательно должно присутствовать хотя бы одно условие, при выполнении которого последовательность рекурсивных вызовов должна быть прекращена.

Обработка вызова рекурсивной функции в принципе ничем не отличается от вызова функции обычной: перед вызовом функции в стек помещаются её аргументы, затем адрес точки возврата, затем, уже при выполнении функции, автоматические переменные, локальные относительно этой функции. Но если при вызове обычных функций число обращений к ним невелико, то для рекурсивных функций число вызовов и, следовательно, количество данных, размещаемых в стеке, определяется глубиной рекурсии. Поэтому при рекурсии может возникнуть ситуация переполнения стека.

Если попытаться отследить по тексту программы процесс выполнения рекурсивной функции, то мы придем к такой ситуации: войдя в рекурсивную функцию, мы «движемся» по ее тексту до тех пор, пока не встретим ее вызова, после чего мы опять начнем выполнять ту же самую функцию сначала. При этом следует отметить самое важное свойство рекурсивной функции – ее первый вызов еще не закончился. Чисто внешне создается впечатление, что текст функции воспроизводится (копируется) всякий раз, когда функция сама себя вызывает. На самом деле этот эффект воспроизводится в компьютере. Однако копируется при этом не весь текст функции (не вся функция), а только ее части, связанные с данными (формальные, фактические параметры, локальные переменные и точка возврата). Алгоритм (операторы, выражения) рекурсивной функции не меняется, поэтому он присутствует в памяти компьютера в единственном экземпляре.

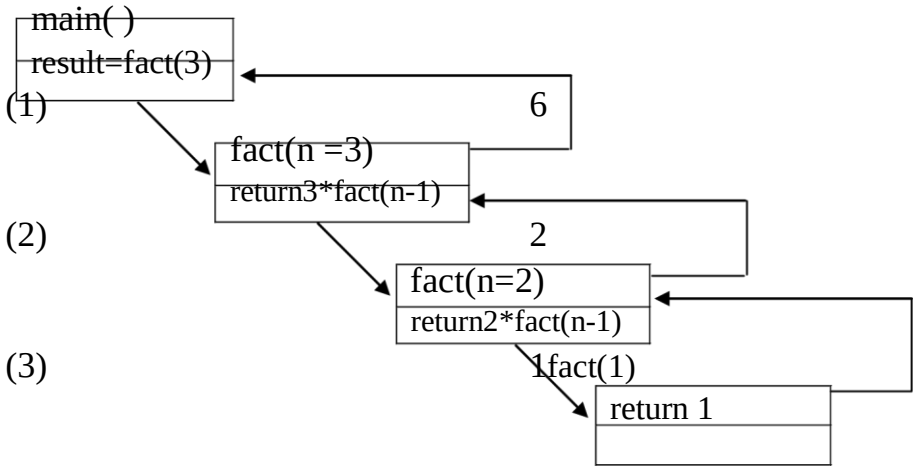
Пример 1. Вычисление n!

Определение факториала рекурсивно:
0! = 1; n! = (n - 1)! * n при n = 1,2,3, …

Поэтому определение функции, вычисляющей факториал, можно написать следующим образом:

```
long fact(int n){ if( n < 1
) return 1; else return n *
fact(n - 1); }
```

Если, например, в main написать long result=fact(3), то последовательность вызовов можно показать так:




Пример 2:

По заданному целому числу распечатать символьную строку цифр, изображающую это число.

```
void cnum(int n){int a = 10; if(n
== 0) return;
else{ cnum(n/a); cout << n%a; } }
```

Однако надо заметить, что гораздо лучше вычислить факториал в простом цикле:

```
long fact (int n){
for (long p = 1, int i=0; i <= n; i++ )
p*= i; return p;}
```

При **косвенной рекурсии** осуществляется перекрёстный вызов функциями друг друга. Хотя бы в одной из них должно быть условие, вызывающее прекращение рекурсии.

Косвенная рекурсия является одним из тех случаев, когда нельзя определить функцию до использования её имени в программе.

Пусть функция f1() вызывает f2(), которая, в свою очередь, обращается к f1(). Пусть первая из них определена ранее второй. Для того чтобы иметь возможность обратиться к функции f2() из f1(), мы должны

поместить объявление функции f2() раньше определения обеих этих функций:

```
void f2(); void
f1(){
…
if(…);
f2();
 …  }
void f2(){
…
f1();
…}
```

## 15.8. Аргументы по умолчанию

Удобным свойством C++ является наличие предопределённых инициализаторов аргументов. Значения аргументов по умолчанию можно задать в объявлении функции, при этом они подставляются автоматически в вызов функции, содержащий меньшее число аргументов, чем объявлено. Например, следующая функция объявлена с тремя аргументами, два из которых инициализированы:

```
void error (const char* msg, int level = 0, int kill = 0);
```

Эта функция может быть вызвана с одним, двумя или тремя аргументами:

```
error ("Ошибка!"); // Вызывается error ("ошибка", 0, 0); error
("Ошибка!", 1);    // вызывается error ("ошибка", 1, 0); error
("Ошибка!", 3, 1);          // значения аргументов по умолчанию  //
не используются.
```

Все аргументы по умолчанию должны быть последними аргументами в списке – ни один явный аргумент не может находиться правее их.

Если аргумент по умолчанию уже определён в одном объявлении, он не может быть переопределён в другом. Аргументы по умолчанию должны быть объявлены при первом объявлении имени функции и не обязаны быть константами:

```
int i = 8; void func
(int = i);
```

Заметим, что если инициализация аргументов произведена в прототипе функции, в определении функции задавать инициализацию аргументов не надо.

## 15.9. Перегрузка функций

В C++ можно перегружать имена функций и использовать одно и то же имя для нескольких функций с различным типом или числом аргументов.

Пусть объявлены следующие функции:

```
int func(int, int); int
func(char, double); int
func(long, double);
int func(float, …);          // Функция с неопределенным числом
                             // аргументов.
int func(char*, int);
```

Рассмотрим, что будет происходить при вызове функции с именем func с некоторым списком аргументов.

Первое, что будет делать компилятор, – это пытаться найти функцию, формальные аргументы которой соответствуют фактическим без всяких преобразований типов или с использованием только неизбежных преобразований, например имени массива к указателю. char string[ ]= "Строка – это массив символов"; int i = func(string, 13);  // func(char*, int); int j = func( 2020L, 36.6) ;  // func(long, double);

Если на первом этапе подходящая функция не найдена, то на втором этапе совершается попытка подобрать такую функцию, чтобы для соответствия формальных и фактических аргументов достаточно было использовать только такие стандартные преобразования, которые не влекут за собой преобразований целых типов к плавающим и наоборот. При этом подбирается функция, для которой число таких преобразований было бы минимально.

Пусть обращение к функции выглядит так:

float x = 36.6; j = func('a', x);

Здесь будет вызвана функция с прототипом func (char, double) и аргумент x будет преобразован к типу double.

Третьим этапом является подбор такой функции, для вызова которой достаточно осуществить любые стандартные преобразования аргументов (и опять так, чтобы этих преобразований было как можно меньше). Так, в следующем фрагменте char y[] = "ГОД:"; int l = func(y, 2020.3);

будет вызвана функция func (char*, int), фактический аргумент типа double которой будет преобразован к int с отбрасыванием дробной части числа.

На четвёртом этапе подбираются функции, для которых аргументы можно получить с помощью всех преобразований, рассмотренных до этого, а также преобразований типов, определённых самим программистом.

Если и в этом случае единственная нужная функция не найдена, то на последнем, пятом этапе, компилятор пробует найти соответствие с учётом списка неопределённых аргументов. Так, при вызове func(1, 2, 3);

подходит лишь одна функция func(float, …). При
обращении

```
int i, j, n;
…
n = func(&i, &j);
```

компилятор не найдёт ни одной подходящей функции и выдаст сообщение об ошибке.

## 15.10. Шаблоны функций

Цель введения шаблонов функций – автоматизация создания функций, которые могут обрабатывать разнотипные данные.

В определении шаблонов семейства функций используется служебное слово **template**, за которым в угловых скобках следует список параметров шаблона. Каждый формальный параметр шаблона обозначается служебным словом **class**, за которым следует имя параметра.

Пример: определение шаблона функций, вычисляющих модули величин различных типов.

```
template <class type>
type abs (type x){return x >0 ? x: -x;}
```

Шаблон функций состоит из двух частей – заголовка шаблона и обычного определения функции, в котором тип возвращаемого значения и типы любых параметров и локальных переменных могут обозначаться именами параметров шаблона, введённых в его заголовке. Пример (снова функция swap):

```
template <class T>
void swap(T& x, T& y){T  z = x;  x = y;  y = z;}
```

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в тексте программы. Например, при обращении

abs(-10, 3) компилятор сформирует такое

определение функции:

```
double abs (double x){return x > 0 ? x: -x;}
```

Далее будет организовано выполнение именно этой функции и в точку вызова в качестве результата вернется значение 10.3.

Пример: шаблон функций для поиска в массиве максимального элемента.

```
#include <iostream>
// Функция устанавливает ссылку  // на
элемент с максимальным значением
template <class type> type &
r_max( int n, type d[ ] ){ int im
= 0;
for (int i = 1; i < n; i++) im = d[im] > d[i] ? im : i;
return d[im];} void main(){
int n = 4, x [ ] = {10, 20, 30, 5};
cout<<"\n  r_max(n, x)= "<< r_max (n, x);       // Печать максимального
                                                 // элемента.
r_max(n, x) = 0;                    // Замена в целом массиве
                                    // максимального элемента нулем.
for (int i = 0; i < n; i++) cout << "\t
x[" << i <<"] = " << x[i]; float f[] =
{10.3, 50.7, 12.6};
cout << "\n  r_max(3, f) = " << r_max (3, f); r_max(3,
f) = 0;
for (i=0; i < 3; i++)
cout << "\t f[" << i << "]=" << f[i]; }
```

Результат выполнения программы:

```
r_max (n, x) = 30 x[0]=10  x[1]=20  x[2]=0  x[3]=5


r_max (3, f) = 50.7 f[0]=10.3  f[1]=0  f[2]=12.6
```

При использовании шаблонов уже нет необходимости готовить заранее все варианты функций с перегруженным именем. Компилятор автоматически, анализируя вызовы функций в тексте программы, формирует необходимые определения именно для таких типов параметров, которые использованы в обращении.

Перечислим основные свойства параметров шаблона:

- Имена параметров шаблона должны быть уникальными во всём определении шаблона.
- Список параметров шаблона функций не может быть пустым.
- В списке параметров шаблона функций может быть несколько параметров. Каждый из них должен начинаться со служебного слова class.
- Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами.
- Имя параметра шаблона имеет в определяемой шаблоном функции все права имени типа. Имя параметра шаблона видно во всём определении и скрывает объекты с аналогичным именем в глобальной по отношению к определению шаблонной функции области видимости.
- Все параметры шаблона должны быть обязательно использованы в спецификациях формальных параметров определения функции.

Заметим, что при необходимости можно использовать прототипы шаблона функций. Например, прототип функции swap():

```
template <class type> void
swap (type&, type&);
```

При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми.

Так, недопустимо:

```
int n = 5;  double d
= 4.3; swap(n, d);
```