

4.8. Функции в C++

С увеличением объема программы становится невозможно удерживать в памяти все детали. Чтобы уменьшить сложность программы, ее разбивают на части. В C++ задача может быть разделена на более простые подзадачи с помощью функций. Разделение задачи на функции также позволяет избежать избыточности кода, т.к. функцию записывают один раз, а вызывают многократно. Программу, которая содержит функции, легче отлаживать.

Часто используемые функции можно помещать в библиотеки. Таким образом, создаются более простые в отладке и сопровождении программы.

4.8.1. Объявление и определение функций

Функция – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т.д. (рис. 23).

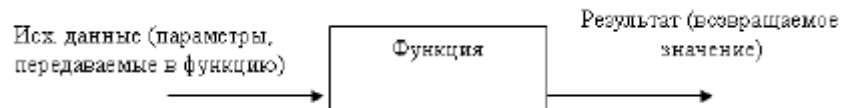


Рис. 23. Функция

Любая функция должна быть **объявлена и определена**.

Объявление функции (прототип, заголовок) задает имя функции, тип возвращаемого значения и список передаваемых параметров.

Определение функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов.

Общая форма определения функции:

**тип имя_функции ([список_формальных_параметров])
{ тело_функции }**

Тело функции – это блок или составной оператор. Внутри функции нельзя определить другую функцию.

В теле функции должен быть оператор, который возвращает полученное значение функции в точку вызова. Он может иметь формы:

- 1) **return выражение;**
- 2) **return.**

Форма 1) используется для возврата результата, поэтому выражение должно иметь тот же тип, что и тип функции в определении. Форма 2) используется, если функция не возвращает значения, т.е. имеет тип *void*. Программист может не использовать этот оператор в теле функции явно, компилятор добавит его автоматически в конец функции перед знаком «}».

Тип возвращаемого значения может быть любым, кроме массива и функции, но может быть указателем на массив или функцию.

Список формальных параметров – это те величины, которые требуется передать в функцию и/или из функции. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя. В объявлении имена можно не указывать.

Для того, чтобы выполнялись операторы, записанные в теле функции, функцию необходимо вызвать. **При вызове указываются: имя функции и фактические параметры.** Фактические параметры заменяют формальные параметры при выполнении операторов тела функции.

Фактические и формальные параметры должны совпадать по количеству и типу в общем случае (возможны исключения).

Объявление функции должно находиться в тексте раньше вызова функции, чтобы компилятор мог осуществить проверку правильности вызова. Если функция имеет тип не *void*, то ее вызов может быть операндом выражения.

Пример 69. Заданы координаты сторон треугольника. Если такой треугольник существует, то найти его площадь.

1. Математическая модель:

- 1) $l = \sqrt{\text{pow}(x_1 - x_2, 2) + \text{pow}(y_1 - y_2, 2)}$; /*длина стороны треугольника*/
- 2) проверка существования треугольника ($a + b > c \ \&\& \ a + c > b \ \&\& \ c + b > a$)
- 3) $p = (a + b + c) / 2$; $s = \sqrt{p * (p - a) * (p - b) * (p - c)}$; //формула Герона

2. Алгоритм:

- 1) Ввести координаты сторон треугольника $(x_1, y_1), (x_2, y_2), (x_3, y_3)$;
- 2) Вычислить длины сторон ab, bc, ca ;
- 3) Проверить существует ли треугольник с такими сторонами. Если да, то вычислить площадь и вывести результат.
- 4) Если нет, то вывести сообщение.
- 5) Если все координаты равны 0, то конец, иначе возврат на п. 1.

3. Программа:

```
#include <iostream>
#include <math.h>
using namespace std;
//функция возвращает длину отрезка, заданного координатами x1,y1 и x2,y2
double line(double x1, double y1, double x2, double y2)
{
    return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
}
//функция возвращает площадь треугольника, заданного длинами сторон a,b,c
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
}
// функция возвращает true, если треугольник существует
bool triangle(double a, double b, double c)
{
    if (a + b > c && a + c > b && c + b > a) return true;
    else return false;
}
void main()
{
    double x1, y1, x2, y2, x3, y3, point1_2, point1_3, point2_3;
    do
    {
        cout << "\nEnter coordinates of triangle:";
        cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
```

```

    point1_2=line(x1,y1,x2,y2);
    point1_3=line(x1,y1,x3,y3);
    point2_3=line(x2,y2,x3,y3);
    if(triangle(point1_2, point1_3, point2_3)==true)
        cout<<"S="<<square(point1_2, point2_3, point1_3)<<"\n";
    else cout<<"\nTriagle doesnt exist";
}
while(!(x1==0&&y1==0&&x2==0&&y2==0&&x3==0&&y3==0));
}

```

4.8.2. Прототип функции

Для того, чтобы к функции можно было обратиться, в том же файле должно находиться определение или описание функции (прототип).

```

double line(double x1,double y1,double x2,double y2);
double square(double a, double b, double c);
bool triangle(double a, double b, double c);
double line(double, double, double, double);
double square(double, double, double);
bool triangle(double, double, double);

```

Это прототипы функций, описанных выше.

При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут оформляться в виде отдельных модулей и храниться в откомпилированном виде в библиотеке объектных модулей. Это относится и к функциям из стандартных модулей. В этом случае определения библиотечных функций уже оттранслированные и оформленные в виде объектных модулей, находятся в библиотеке компилятора, а описания функций необходимо включать в программу дополнительно. Это делают с помощью препроцессорных команд **include**< имя файла>.

Имя_файла – определяет заголовочный файл, содержащий прототипы группы стандартных для данного компилятора функций. Например, почти во всех программах использовали команду **#include <iostream>** для описания объектов потокового ввода-вывода и соответствующие им операции.

При разработке своих программ, состоящих из большого количества функций, и, 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"**.

4.8.3. Параметры функции

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Существует два способа передачи параметров в функцию: по адресу и по значению.

При передаче по значению выполняются следующие действия:

- 1) вычисляются значения выражений, стоящие на месте фактических параметров;
- 2) в стеке выделяется память под формальные параметры функции;
- 3) каждому формальному параметру присваивается значение фактического параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

Пример 70

```
double square(double a, double b, double c)
{
    //функция возвращает площадь треугольника, заданного длинами сторон a,b,c
    double s, p=(a+b+c)/2;
    return s=sqrt(p*(p-a)*(p-b)*(p-c)); //формула Герона
}

...
double s1=square(2.5,2,1);
double a=2.5,b=2,c=1;
double s2=square(a,b,c);
double x1=1,y1=1,x2=3,y2=2,x3=3,y3=1;
double s3=square(sqrt(pow(x1-x2,2)+pow(y1-y2,2)), //расстояние между 1 и 2
                sqrt(pow(x1-x3,2)+pow(y1-y3,2)), //расстояние между 1 и 3
                sqrt(pow(x3-x2,2)+pow(y3-y2,2))); //расстояние между 2 и 3
...
p и s – локальные переменные.
```

При передаче по значению в стек заносятся копии фактических параметров и операторы функции работают с этими копиями. Доступа к самим фактическим параметрам у функции нет, следовательно, нет возможности их изменить.

При передаче по адресу в стек заносятся копии адресов параметров, следовательно, у функции появляется доступ к ячейке памяти, в которой находится фактический параметр и она может его изменить.

Пример 71

```
void Change(int a,int b) //передача по значению
{int r=a; a=b; b=r;}

int x=1, y=5;
Change(x, y);
A      1      5
B      5      1
r      1
cout<<"x="<<x<<"y="<<y;
```

Результат работы программы: $x=1$ $y=5$

```
void Change(int *a,int *b) //передача по адресу
{int r=*a;*a=*b;*b=r;}

int x=1,y=5;
Change(&x,&y);
A      &x     5
B      &y     1
r      1
cout<<"x="<<x<<"y="<<y;
```

Результат работы программы: $x=5$ $y=1$

Для передачи по адресу также могут использоваться ссылки. При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

Пример 72

```
void Change(int &a, int &b)
{int r=a; a=b; b=r;}
int x=1, y=5;
Change(x,y);
A      &x      5
B      &y      1
r      1
cout<<"x="<<x<<"y="<<y;
Результат работы программы: x=5 y=1
```

Использование ссылок вместо указателей улучшает читаемость программы, т.к. не надо применять операцию разыменовывания. Использование ссылок вместо передачи по значению также более эффективно, т.к. не требует копирования параметров. Если требуется запретить изменение параметра внутри функции, используется модификатор *const*. Рекомендуется ставить *const* перед всеми параметрами, изменение которых в функции не предусмотрено (по заголовку будет понятно, какие параметры в ней будут изменяться, а какие нет).

Пример 73

```
void Change(int &a, const int &b)
{int r=a; a=b; b=r;}

int x=1, y=5;
Change(x,y);
cout<<"x="<<x<<"y="<<y;
Результат работы программы: x=5 y=5
```

4.8.4. Локальные и глобальные переменные

Переменные, которые используются внутри данной функции, называются **локальными**. Память для них выделяется в стеке, поэтому после окончания работы функции они удаляются из памяти. Нельзя возвращать указатель на локальную переменную, т.к. память, выделенная такой переменной, будет освобождаться.

Пример 73

```
int*f()
{
    int a;
    . . . .
    return&a; // НЕВЕРНО
}
```

Глобальные переменные – это переменные, описанные вне функций. Они видны во всех функциях, где нет локальных переменных с такими именами.

Пример 74

```
int a,b; //глобальные переменные
void change()
{
    int r; //локальная переменная
    r=a; a=b; b=r;
}
```

```

void main()
{
    cin>>a>>b;
    change();
    cout<<"a="<<a<<"b="<<b;
}

```

Глобальные переменные также можно использовать для передачи данных между функциями, но этого не рекомендуется делать, т.к. это затрудняет отладку программы и препятствует помещению функций в библиотеки. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

4.8.5. Функции и массивы

4.8.5.1. Передача одномерных массивов как параметров функции

При использовании массива как параметра функции, в функцию передается указатель на его первый элемент, т.е. массив всегда передается по адресу. При этом теряется информация о количестве элементов в массиве, поэтому размерность массива следует передавать как отдельный параметр. Так как в функцию передается указатель на начало массива (*передача по адресу*), то массив может быть изменен за счет операторов тела функции.

Пример 75. Удалить из массива все четные элементы

```

#include <iostream>
using namespace std;
//ФУНКЦИЯ ФОРМИРОВАНИЯ МАССИВА
int form(int a[100])
{
    int n;
    cout<<"\nEnter n";
    cin>>n;
    for(int i=0;i<n;i++)
        a[i]=rand()%100;
    return n;
}
//ПЕЧАТЬ МАССИВА
void print(int a[100], int n)
{
    for(int i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<"\n";
}
//ФУНКЦИЯ УДАЛЕНИЯ ИЗ МАССИВА ЧЕТНЫХ ЭЛЕМЕНТОВ
void Dell(int a[100], int&n)
{
    int j=0,i,b[100];
    for(i=0; i<n; i++)
        if(a[i]%2!=0)
        {
            b[j]=a[i]; j++;
        }
    n=j;
}

```

```

    for(i=0; i<n; i++) a[i]=b[i];
}
void main()
{
    int a[100];
    int n;
    n=form(a);
    print(a,n);
    Dell(a,n);
    print(a,n);
}

```

Пример 76. Удалить из массива все элементы, совпадающие с первым элементом, используя динамическое выделение памяти.

```

#include <iostream>
using namespace std;
int* form(int&n)
{
    cout<<"\nEnter n";
    cin>>n;
    int*a=new int[n];    /*указатель на динамическую область памяти*/
    for(int i=0; i<n; i++)
        a[i]=rand()%10;
    return a;
}
void print(int*a, int n)
{
    for(int i=0; i<n; i++)
        cout<<a[i]<<" ";
    cout<<"\n";
}
int*Dell(int *a, int&n)
{
    int k,j,i;
    for(k=0,i=0; i<n; i++)
        if(a[i]!=a[0]) k++; /*определяем количество элементов, не совпадающих с первым */
    int*b;
    b=new int [k]; /*указатель на динамическую область памяти для нового массива*/
    for(j=0,i=0; i<n; i++)
        if(a[i]!=a[0]) b[j++]=a[i];
    n=k;
    return b;
}
void main()
{
    int *a;
    int n;
    a=form(n);
    print(a,n);
    a=Dell(a,n);
    print(a,n);
}

```

4.8.5.2. Передача многомерных массивов в функцию

По определению многомерные массивы в С и С++ не существуют. Если описываем массив с несколькими индексами, например, массив `int mas[3][4]`, то это означает, что описан одномерный массив `mas`, элементами которого являются указатели на одномерные массивы `int[4]`.

При передаче многомерных массивов в функцию размерности должны либо передаваться в качестве параметров, либо заранее определены как константы.

Пример 78. Транспонирование матрицы

Если определить заголовок функции:

`void transp(int a[][N], int n)` {...} – то получится, что в функцию передается массив с неизвестными размерами.

По определению массив должен быть одномерным, и его элементы должны иметь одинаковую длину. При передаче массива ничего не сказано и о размере элементов, поэтому компилятор выдаст ошибку.

Самый простой вариант решения этой проблемы определить функцию следующим образом:

```
const int N=4;           //глобальная переменная
//ввод матрицы
void form(int a[][N], int n, int m) тогда размер каждой строки будет не более N.
```

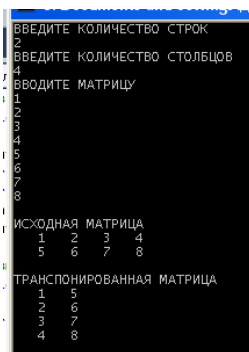
```
#include <iostream>
#include <windows.h>
#include <iomanip> // файл, где определена функция setw(4)
using namespace std;
const int N=4;           //глобальная переменная
//ввод матрицы
void form(int a[][N],int n,int m)
{
    cout<<"ВВОДИТЕ МАТРИЦУ\n";
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            cin>>a[i][j];
    cout<<"\n";
}
//вывод матрицы
void print(int a[][N],int n, int m)
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m; j++)
            cout<<setw(4)<<a[i][j];
        cout<<"\n";
    }
    cout<<"\n";
}
void transp(int a[][N], int b[][N], int n, int m)
{
    for(int i=0; i<n; i++)
```



```

        for(int j=0; j<m; j++)
            b[j][i]=a[i][j];
    }
void main()
{
    SetConsoleOutputCP(1251);
    const int k=4;
    int mas[k][N],mas1[N][k];
    int p,l;
    cout<<"ВВЕДИТЕ КОЛИЧЕСТВО СТРОК\n";
    cin>>p;
    cout<<"ВВЕДИТЕ КОЛИЧЕСТВО СТОЛБЦОВ\n";
    cin>>l;
    form(mas,p,l);
    cout<<"ИСХОДНАЯ МАТРИЦА\n";
    print(mas,p,l);
    transp(mas,mas1,p,l);
    cout<<"ТРАНСПОНИРОВАННАЯ МАТРИЦА\n";
    print(mas1,l,p);
}

```



4.8.5.3. Передача строк в качестве параметров функций

Строки при передаче в функции могут передаваться как одномерные массивы типа *char* или как указатели типа *char**. В отличие от обычных массивов в функции не указывается длина строки, т.к. в конце строки есть признак конца строки `/0`.

Пример 77. Функция поиска заданного символа в строке

```

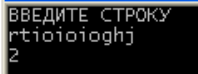
#include <iostream>
#include <windows.h>
using namespace std;
int find (char *s,char c)
{
    for (int I=0;I<strlen(s);I++)
        if(s[I]==c) return (I);
    return (-1);
}
/*С помощью этой функции подсчитаем количество различных
гласных букв в строке.*/

```

```

void main()
{
    SetConsoleOutputCP(1251);
    char s[255];
    int k=0;
    cout<<"ВВЕДИТЕ СТРОКУ";
    cin>>s;
    char*gl="aouiey";
    for(int I=0;I<strlen(gl);I++)
    if(find(s,gl[I])>=0)k++;
    cout<<k;
}

```



```

ВВЕДИТЕ СТРОКУ
rtioioioighj
2

```

4.8.6. Функции с начальными (умалчиваемыми) значениями параметров

В определении функции может содержаться начальное (умалчиваемое) значение параметра. Это значение используется, если при вызове функции соответствующий параметр опущен. Все параметры, описанные справа от такого параметра, также должны быть умалчиваемыми.

Пример 79

```

void print(char*name="Номер дома: ", int value=1)
{cout<<"\n"<<name<<value;}

```

Вызовы:

1. *print()*;

Вывод: Номер дома: 1

2. *print("Номер квартиры:",15)*;

Вывод: Номер квартиры: 15

3. *print(15)*; - ошибка, т.к. параметры можно опускать только с конца

Поэтому функцию лучше переписать так:

```

void print(int value=1, char*name="Номер дома: ")
{cout<<"\n"<<name<<value;}

```

Вызовы:

1. *print()*;

Вывод: Номер дома: 1

2. *print(15)*;

Вывод: Номер дома: 15

3. *print(6, "Размерность пространства: ")*;

Вывод: Размерность пространства: 6

4.8.7. Подставляемые (inline) функции

Некоторые функции в C++ можно определить с использованием служебного слова *inline*. Такая функция называется подставляемой или встраиваемой.

Пример 80

```
inline float Line(float x1, float y1, float x2=0, float y2=0)
{return sqrt(pow(x1-x2,2)+pow(y1-y2,2));} //функция возвращает расстояние от точки
//с координатами(x1,y1) до точки с координатами (x2,y2) (по умолч. центр координат).
```

Обрабатывая каждый вызов подставляемой функции, компилятор пытается подставить в текст программы код операторов ее тела. Спецификатор *inline* определяет для функции так называемое внутреннее связывание, которое заключается в том, что компилятор вместо вызова функции подставляет команды ее кода. При этом может увеличиваться размер программы, но исключаются затраты на передачу управления к вызываемой функции и возврата из нее. Подставляемые функции используют, если тело функции состоит из нескольких операторов.

Подставляемыми не могут быть:

1. рекурсивные функции;
2. функции, у которых вызов размещается до ее определения;
3. функции, которые вызываются более одного раза в выражении;
4. функции, содержащие циклы, переключатели и операторы переходов;
5. функции, которые имеют слишком большой размер, чтобы сделать подстановку.

4.8.8. Функции с переменным числом параметров

В C++ допустимы функции, у которых при компиляции не фиксируется число параметров, и, кроме того может быть неизвестен тип этих параметров. Количество и тип параметров становится известным только в момент вызова, когда явно задан список фактических параметров.

Каждая функция с переменным числом параметров должна иметь хотя бы один обязательный параметр. Определение функции с переменным числом параметров:

```
тип имя (явные параметры, ... )
{ тело функции }
```

После списка обязательных параметров ставится запятая, а затем многоточие, которое показывает, что дальнейший контроль соответствия количества и типов параметров при обработке вызова функции производить не нужно.

Сложность заключается в определении начала и конца списка параметров, поэтому каждая функция с переменным числом параметров должна иметь механизм определения количества и типов параметров.

Существует два подхода:

- 1) известно количество параметров, которое передается как обязательный параметр;
- 2) известен признак конца списка параметров.

Пример 81. Найти среднее арифметическое последовательности чисел, если известно количество параметров (подход 1).

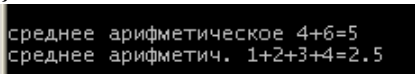
```
#include <iostream>
#include <windows.h>
using namespace std;
```

```

float sum(int k, ...)
{
    int *p=&k;    //настроили указатель на параметр k
    double s=0,n=k;
    for(;k!=0;k--)
        s+=*(++p);
    return s/n;
}

void main()
{
    SetConsoleOutputCP(1251);
    cout<<"\nsреднее арифметическое 4+6="<<sum(2,4,6); /* среднее арифметич. 4+6*/
    cout<<"\nsреднее арифметич. 1+2+3+4="<<sum(4,1,2,3,4);/*среднее арифметич.
1+2+3+4*/
}

```



В примере 81 для доступа к списку параметров используется указатель **p* типа *int*. Он устанавливается на начало списка параметров в памяти, а затем перемещается по адресам фактических параметров (*++p*).

Пример 82. Найти среднее арифметическое последовательности чисел, если известен признак конца списка параметров (подход 2).

```

#include <iostream>
using namespace std;
double sum(int k, ...)
{
    int *p=&k;                                //настроили указатель на параметр k
    double s=0,i;
    for( i=1;*p!=0;i++)                       //пока нет конца списка
        s+=*(p++);
    return s/(i-1);
}

void main()
{
    cout<<"\n4+6="<<sum(4,6,0);              /*находит среднее арифметическое 4+6*/
    cout<<"\n1+2++3+4="<<sum(1,2,3,4,0);     /*находит среднее арифметич.1+2+3+4*/
}

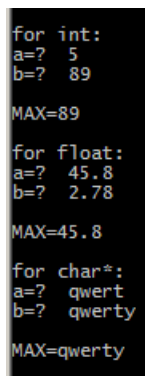
```

4.8.9. Перегрузка функций

Цель перегрузки состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров. Для обеспечения перегрузки необходимо для каждой перегруженной функции определить возвращаемые значения и передаваемые параметры так, чтобы каждая перегруженная функция отличалась от другой функции с тем же именем. Компилятор определяет, какую функцию выбрать по типу фактических параметров.

Пример 83 Поиск максимального значения

```
#include <iostream>
using namespace std;
int max(int a,int b)
{
    if(a>b)return a;
    else return b;
}
float max(float a,float b)
{
    if(a>b)return a;
    else return b;
}
char*max(char*a,char*b)
{
    if(strcmp(a,b)>0) return a;
    else return b;
}
void main()
{
    float a2, b2;
    char s1[20];
    char s2[20];
    cout<<"\nfor float:\n";
    cout<<"a=?";cin>>a2;
    cout<<"b=?";cin>>b2;
    cout<<"\nMAX="<<max(a2,b2)<<"\n";
    cout<<"\nfor char*:\n";
    cout<<"a=?";cin>>s1;
    cout<<"b=?";cin>>s2;
    cout<<"\nMAX="<<max(s1,s2)<<"\n";
}
```



```
for int:
a=? 5
b=? 89
MAX=89

for float:
a=? 45.8
b=? 2.78
MAX=45.8

for char*:
a=? qwerty
b=? qwerty
MAX=qwerty
```

Правила описания перегруженных функций:

- 1) Перегруженные функции должны находиться в одной области видимости.
- 2) Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В разных вариантах перегруженных функций может быть разное количество умалчиваемых параметров.
- 3) Функции не могут быть перегружены, если описание их параметров отличается только модификатором *const* или наличием ссылки.

Например,

Функции `int&f1(int&,const int&){. . . }` и `int f1(int,int){. . . }` – не являются перегруженными, т.к. компилятор не сможет узнать какая из функций вызывается: нет синтаксических отличий между вызовом функции, которая передает параметр по значению и функции, которая передает параметр по ссылке.

```
void print(double);
void print(long);
void f()
{
    print(1L); // print(long)
    print(1.0); // print(double)
    print(1); // ошибка - непонятно, что вызывать:
              // print(long(1)) или print(double(1))
    print('s'); // ошибка
}
```

4.8.10. Шаблоны функций

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов. При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризуется, т.е. тип данных передается как параметр шаблона.

Формат шаблона:

```
template <class имя_типа [,class имя_типа]>
заголовок_функции
{тело функции}
```

Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: **template<список параметров шаблона>** и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

Пример 84. Шаблон функции, которая находит абсолютное значение числа любого типа.

```
template<class type> //type – имя параметризуемого типа
type abs(type x)
{
    if(x<0)return -x;
    else return x;
}
```

Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе. Например, если в программе вызов функции осуществляется как `abs(-1.5)`, то компилятор сформирует определение функции `double abs(double x){. . . }`.

Пример 85. Шаблон функции, которая меняет местами две переменных

```
template <class T> //T – имя параметризуемого типа
void change(T*x,T*y)
{ T z=*x; *x=*y; *y=z; }
```

Вызов этой функции может быть:

```
long k=10, l=5;  
change(&k, &l);
```

Компилятор сформирует определение:

```
void change(long*x,long*y)  
{ long z=*x;*x=*y;*y=z;}
```

Пример 86. Найти максимальный элемент массива и заменить его на 0. Выполнить для целого и вещественного массивов. Для вывода массива и поиска максимального использовать шаблоны функции.

```
#include <iostream>  
#include <windows.h>  
using namespace std;  
template<class Data>  
void print(Data a[],int n)  
{  
    for(int i=0;i<n;i++)  
        cout<<a[i]<<" ";  
    cout<<"\n";  
}  
template<class Data>  
Data &rmax(int n,Data a[])  
{  
    int im=0;  
    for(int i=0;i<n;i++)  
        if(a[im]<a[i])im=i;  
    return a[im]; //возвращает ссылку на максимальный элемент в массиве  
}  
void main()  
{  
    SetConsoleOutputCP(1251);  
    int n=5,i;  
    int x[]={ 10,20,30,15,5};  
    cout<<"ИСХОДНЫЙ МАССИВ\n";  
    print(x,n);  
    cout<<"rmax(n,x)="<<rmax(n,x)<<"\n";  
    rmax(n,x)=0;//заменяет максимальный элемент массива на 0  
    cout<<"МАССИВ ПОСЛЕ ЗАМЕНЫ\n";  
    print(x,n);  
    float y[]={ 10.4,20.2,30.6,15.5};  
    cout<<"\nИСХОДНЫЙ МАССИВ\n";  
    print(y,4);  
    cout<<"rmax(n,y)="<<rmax(n,y)<<"\n";  
    rmax(4,y)=0;//заменяет максимальный элемент массива на 0  
    cout<<"МАССИВ ПОСЛЕ ЗАМЕНЫ\n";  
    print(y,4);  
}
```

```

ИСХОДНЫЙ МАССИВ
10 20 30 15 5
rmax(n,x)=30
МАССИВ ПОСЛЕ ЗАМЕНЫ
10 20 0 15 5

ИСХОДНЫЙ МАССИВ
10.4 20.2 30.6 15.5
rmax(n,y)=30.6
МАССИВ ПОСЛЕ ЗАМЕНЫ
10.4 20.2 0 15.5

```

Основные свойства параметров шаблона функций

1. Имена параметров должны быть уникальными во всем определении шаблона.
2. Список параметров шаблона не может быть пустым.
3. В списке параметров шаблона может быть несколько параметров, каждый из них начинается со слова *class*.

4.8.11. Указатель на функцию

Каждая функция характеризуется типом возвращаемого значения, именем и списком типов ее параметров. Если имя функции использовать без последующих скобок и параметров, то оно будет выступать в качестве указателя на эту функцию, и его значением будет выступать адрес размещения функции в памяти. Это значение можно будет присвоить другому указателю. Тогда этот новый указатель можно будет использовать для вызова функции.

Указатель на функцию определяется следующим образом:

тип_функции (*имя_указателя) (спецификация параметров)

Пример 87

```

1. int f1(char c){...}           //определение функции
int(*ptrf1)(char);              //определение указателя на функцию f1

2. char*f2(int k,char c){...}   //определение функции
char*ptrf2(int,char);           //определение указателя

```

В определении указателя количество и тип параметров должны совпадать с соответствующими типами в определении функции, на которую ставится указатель.

Вызов функции с помощью указателя имеет вид:

(*имя_указателя)(список фактических параметров);

Правила передачи аргументов одинаковы как для обычного вызова функции, так и для её вызова через указатель.

Пример:

```

void (*pf)(char*);              // указатель на void(char*)
void f1(char*);                 // функция void(char*);
int f2(char*);                  // функция int(char*);
void f3(int*);                  // функция void(int*);

void f()
{
    pf = &f1;                  // правильно
    pf = &f2;                  // ошибка - не тот тип возвращаемого
                                // значения
    pf = &f3;                  // ошибка - не тот тип аргумента
    (*pf)("asdf");              // правильный вызов
}

```



```

(*pf)(1);           // ошибка - не тот тип аргумента
int i = (*pf)("qwer"); // ошибка - void присваивается int
}

```

Пример 88

```

#include <iostream>
using namespace std;
void f1() {cout<<"\nfunction f1";}
void f2() {cout<<"\nfunction f2";}
void main()
{
    void(*ptr)(); //указатель на функцию
    ptr=f2;       /*указателю присваивается адрес функции f2*/
    (*ptr)();     //вызов функции f2
    ptr=f1;       /*указателю присваивается адрес функции f1*/
    (*ptr)();     //вызов функции f1с помощью указателя
}

```

```

function f2
function f1

```

При определении указатель на функцию может быть сразу проинициализирован:

```
void (*ptr)()=f1;
```

Указатели на функции могут быть объединены в массивы. Например,

`float(*ptrMas[4])(char)` – описание массива, который содержит 4 указателя на функции. Каждая функция имеет параметр типа *char* и возвращает значение типа *float*. Обратиться к такой функции можно следующим образом:

```
float a=(*ptrMas[1])('x'); //обращение ко второй функции*/
```

Пример 89

```

#include <iostream>
using namespace std;
void f1() {cout<<"The end of work"; getch(); exit(0);}
void f2() {cout<<"The work #1\n";}
void f3() {cout<<"The work #2\n";}
void main()
{
    void(*fptr[])()={f1,f2,f3};
    int n;
    while(1)//бесконечный цикл
    {
        cout<<"Enter the number";
        cin>>n;
        fptr[n](); // аналогично (*fptr[n]) () - вызов функции с номером n
    }
}

```

```

Enter the number 1
The work #1
Enter the number 2
The work #2
Enter the number 0
The end of work_

```

Указатели на функции удобно использовать в тех случаях, когда функцию надо передать в другую функцию как параметр.

Часто бывает удобно с помощью объявления *typedef* присвоить новое имя типу указателя на функцию, и пользоваться этим именем вместо того, чтобы все время использовать синтаксически достаточно сложную форму записи указателя на функцию. Например:

```
typedef float (*MyFuncPtrType)(int, char*);  
MyFuncPtrType my_func_ptr;
```

4.8.12. Ссылки на функцию

Подобно указателю на функцию определяется и ссылка на функцию:

тип_функции(&имя_ссылки)(параметры) = инициализирующее_выражение;

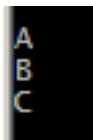
Пример 91.

```
int f(float a,int b){... }      /*определение функции*/  
int(&fref)(float,int)=f;      //определение ссылки
```

Использование имени функции без параметров и скобок будет восприниматься как адрес функции. Ссылка на функцию является синонимом имени функции. Изменить значение ссылки на функцию нельзя, поэтому более широко используются указатели на функции, а не ссылки.

Пример 92.

```
#include <iostream>  
using namespace std;  
void f(char c)  
{ cout<<"\n"<<c; }  
void main()  
{  
    void (*pf)(char);      //указатель на функцию  
    void(&rf)(char)=f;     //ссылка на функцию  
    f('A');                //вызов по имени  
    pf=f;                  //указатель ставится на функцию  
    (*pf)('B');            //вызов с помощью указателя  
    rf('C');               //вызов по ссылке  
}
```



4.8.13. Рекурсивные функции

Функции, вызывающие самих себя, называются рекурсивными. В общем случае, *рекурсия* – это определение какой – либо сущности с использованием её же.

Распространённым примером рекурсивной функции является функция, вычисляющая факториал. (Факториалом целого положительного числа n называется произведение всей последовательности целых чисел от 1 до n (в математике это записывается коротко как $n!$)).

```
// Рекурсивная функция
int fac(int n)
{
    if(n == 1)
        return 1;
    else
        return n*fac(n -1);
}
// Не рекурсивная функция
int facn(int n)
{
    int fact = 1;
    for(int m = 1; m <= n; m++)
        fact *= m;
    return fact;
}
```

В рекурсивной функции обязательно должно быть условие выхода из рекурсии, при котором функции не нужно вызывать саму себя. В примере с факториалом это

```
if(n == 1) return 1;
```

Рекурсивный вызов внутри функции должен изменять аргументы исходной функции (в приведённом примере вычисляется факториал числа, на 1 меньшего).

Рекурсивные варианты большинства функций выполняются медленнее, чем их итеративные (не рекурсивные) варианты, так как при рекурсии тратится время на вызовы функции самой себя.

Реализация механизма рекурсии

Переменные функции хранятся в сегменте стека (локальной памяти). При каждом рекурсивном вызове функции все локальные данные тоже помещаются в стек (данные первого вызова функции остаются в сегменте стека). Этот процесс называется прямым ходом рекурсии или рекурсивным спуском.

Прямой ход рекурсии продолжается до достижения условия выхода из рекурсии, после чего запускается процесс обратного хода рекурсии (рекурсивного подъёма). При этом из стека вынимаются сохранённые ранее значения и используются для последующих вычислений.

Структура рекурсивной функции может принимать три разных формы:

- форма с выполнением действий до рекурсивного вызова (на рекурсивном спуске):
`void Rec(void) { S; if (условие) Rec(); }`

- форма с выполнением действий после рекурсивного вызова (на рекурсивном возврате — подъёме):

```
void Rec(void) { if (условие) Rec(); S; }
```

- форма с выполнением действий как до (на рекурсивном спуске), так и после рекурсивного вызова (на рекурсивном возврате):

```
void Rec(void) { S1; if (условие) Rec(); S2; }
```

Названия «рекурсивный спуск» и «рекурсивный подъем» связаны с понятием глубины рекурсии – функция спускается на глубину рекурсии и поднимается «оттуда». Глубина рекурсии – это максимальная степень вложенности рекурсивных вызовов.

В принципе, любой цикл можно заменить эквивалентной рекурсивной программой.

Пример 3.3. Используя рекурсивную функцию, вычислить сумму $S=1+1/2+1/3+...+1/n$

```
#include <conio.h>
#include <iostream>
using namespace std;
void SumRec(int n, double& sum)
{
    if(n==1)
        sum = 1;
    else
        SumRec(n-1, sum);
    sum = sum + 1.0/n; // будет выполнено на обратном ходе рекурсии
}
int main()
{
    cout<<"Input n:\n";
    int n;
    cin>>n;
    double sum;
    SumRec(n, sum);
    cout<<"sum = "<<sum;
}
```