

Функции

Функция — подпрограмма, выполняющая некоторый набор операторов, в итоге, результатом является некоторое законченное действие.

В языке C++ вся программа представляет собой набор функций. Основной функцией, с которой взаимодействует операционная система, является функция `int main()` `{...}`. Все остальные функции вызываются внутри `int main(){...}`.

При работе с функциями возможно три действия:

1. Объявление функции.
2. Определение функции.
3. Вызов функции.

1.1. Указатели

До этого в программе всегда объявлялись простые переменные. Оператор объявления определяет тип и символическое имя переменной, а также требует от программы выделить память под эту переменную и определяет скрытым образом ее местоположение. Для того, чтобы определить адрес простой переменной необходимо применить операцию *взятия адреса*: `<имя переменной>`.

Следующая программа демонстрирует операцию взятия адреса для переменной типа `int`:

Листинг 1.1. Операция взятия адреса переменной

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     int a = 5;
6     cout << "value=" << a << " address=" << &a << endl;
7     return 0;
8 }
```

Результат работы программы:

```
value=5 address=0026FE40
```

Результат работы программы — это адрес ячейки памяти, в которой расположен первый байт, занимаемый этой переменной. Обычно операция взятия адреса дает результат в шестнадцатичной нотации.

Таким образом, значение переменной воспринимается как именованная величина, а ее адрес — как производная. Однако, во многих языках программирования существуют специальные типы переменных, которые в качестве именованной величины хранят не значение переменной, а ее адрес. Такие переменные называются *указателями*. Для получения значения, находящегося в ячейке памяти, адрес которой хранит указатель, используется операция *разыменования*: $\ast\langle\text{имя переменной}\rangle$. Следующая программа демонстрирует операцию разыменования для указателя `p`:

Листинг 1.2. Пример операции разыменования типов

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     int a = 5;
6     int *p = &a;
7     cout << "address: " << p << " value: " << *p << endl;
8     return 0;
9 }
```

Результат работы программы:

```
address=002EF8D4 value=5
```

Как видно из листинга 1.2 объявление указателя имеет следующий вид:

$$\langle\text{тип}\rangle \ast\langle\text{имя переменной}\rangle$$

Например, запись `int *p` означает, что значение переменной `*p` имеет тип `int`, а `p` — это указатель на тип `int`.

Переменная-указатель никогда не бывает просто указателем, она всегда указывает на какой-нибудь тип. Указатели могут указывать на различные типы данных, имеющие разный размер, но сами указатели в памяти обычно занимают 2 или 4 байта.

Необходимо помнить, что при объявлении указателей автоматически выделяется память только для записи адреса, но выделения памяти под хранение значения, расположенного по этому адресу, не происходит.

Если указатель не инициализировать, то в поле для записи адреса будет записано случайное шестнадцатиричное число, которое может означать любую ячейку памяти, в том числе, занятую системными данными. Попытка изменить данные по этому адресу может привести к непредсказуемым последствиям.

Способов инициализации несколько:

- Как показано в листинге 1.2, сначала инициализируется переменная определенного типа, затем объявляется указатель на адрес этой переменной:

```
int a = 5; //переменная типа int
int *p = &a //указатель на адрес переменной типа int
```

- Указателю присваивается адрес в явном виде, но необходимо точно знать, что нужная ячейка памяти свободна:

```
int *p;

p = (int *)0x002EF8D4; //указан адрес ячейки памяти
```

- Используется нулевой указатель (null-указатель). Стандарт языка C++ гарантирует, что нулевой указатель никогда не указывает на корректные данные, поэтому он часто используется в качестве признака окончания каких-либо действий:

```
int *p = NULL; //null-указатель или
int *p = 0;
```

- Используется операция выделения памяти **new**. С помощью операции выделения памяти, память выделяется не на стадии компиляции, а на стадии выполнения программы, что позволяет экономить память при условии, если размер и количество данных заранее неизвестны. Общий вид:

$$\langle \text{тип} \rangle * \langle \text{имя указателя} \rangle = \text{new } \langle \text{тип} \rangle$$

Поскольку память выделяется программистом на стадии выполнения программы, то и освобождается память тоже программистом с помощью операции **delete**.

Например, объявляется указатель на тип **int** и выделяется память под переменную типа **int**, выполняются какие-либо действия и освобождается память:

Листинг 1.3.

```
1 int main(){
2     ...
3     int *p = new int;
4     //операторы
5     delete p;
6     ...
7 }
```

1.2. Объявление функции

Объявление функции (еще называют *прототип функции*) представляет собой заголовок функции, заканчивающийся точкой с запятой.

Заголовок функции содержит: тип возвращаемого значения, имя функции, а также число и типы параметров, которые требуется передать функции при ее вызове.

Синтаксис прототипа функции имеет вид:

⟨тип возвращаемого значения⟩ ⟨имя функции⟩ (⟨[число и типы параметров]⟩);

Функция может не содержать ни одного параметра, но круглые скобки в заголовке функции обязательны:

```
float print();
```

Список параметров содержит все данные, необходимые для корректной работы функции. Параметры перечисляются через запятую, перед каждым пишется тип данного параметра.

Например, прототип функции, определяющей площадь треугольника по трем сторонам:

```
float square (float, float, float);
```

Прототип функции может содержать имена параметров. Компилятор их игнорирует, но наличие имен позволяет легче понимать код программы.

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

Данная запись понятнее предыдущей. В принципе из названия и списка параметров можно догадаться, что речь идет о нахождении площади треугольника.

Прототип функции используется тогда, когда определение функции находится где-то в другом месте файла (или в другом файле).

Например, в случае крупного проекта прототипы функции полезны для того, чтобы разные программисты могли разрабатывать свои части проекта, не дожидаясь пока будут готовы какие-либо функции.

1.3. Определение функции

Любая функция должна быть определена и только один раз. Определение функции — это объявление функции (без точки с запятой в конце) плюс тело функции. Типы и число параметров в объявлении и определении должны совпадать, имена параметров совпадать не обязаны.

Например, определение функции, определяющей площадь треугольника по трем сторонам:

```
1 float square (float a, float b, float c){  
2     float p = (a + b + c)/2; //полупериметр треугольника  
3     float s = sqrt(p*(p - a)*(p - b)*(p - c)); //площадь по формуле  
        Герона  
4     return s;  
5 }
```

Любая функция должна возвращать значение. Исключение составляют функции, объявленные с помощью типа `void` (специальный тип для функций, не возвращающих значений, например, функция, печатающая на экран двумерный массив.)

Возвращение значения происходит с помощью оператора `return`. Тип переменной, используемой в операторе `return` должен совпадать с типом возвращаемого значения функции. Например, в функции, приведенной выше — возвращаемым является значение переменной `s`.

Функция, объявленная типом `void` не может возвращать значение.

Следующие определения функции являются правильными:

```
int func(){ return 1;}  
  
void print(){}  
  
void print(){return;}
```

1.4. Вызов функции

Для работы с функцией, ее необходимо вызывать. Вызовов функции может быть сколько угодно, в том месте, где это необходимо для правильной работы программы.

Вызов функции может быть либо в функции `int main()`, либо в любой другой функции. Вызываемая функция должна быть до этого либо определена, либо объявлена.

Функция, возвращающая значение, вызывается «справа», как самостоятельное выражение, либо в составе выражения:

```
1 int func (int a, float b){
2     //тело функции
3     return S;
4 }
5
6 int main(){
7     int a, b, f;
8     float c;
9     //операторы
10    cout << a + func (func (a, c), c);
11    //операторы
12    return 0;
13 }
```

Функция, не возвращающая значение, вызывается только «слева»:

```
1 void func (int a, float b){
2     //тело функции
3 }
4
5 int main(){
6     int a, b, f;
7     float c;
8     //операторы
9     func (a, c);
10    //операторы
11    return 0;
```

Типы и число параметров, использующихся в вызовах функции должны обязательно совпадать с типами и числом параметров в объявлении или определении функции. Если типы не совпадают, то в случае отсутствия перегрузки (см. дальше), будет произведено неявное преобразование типов, при этом возможна потеря данных.

Следовательно, надо внимательно следить за тем, чтобы списки параметров в вызываемых и объявляемых функциях совпадали по типам, числу и логике использования.

Например, в случае нахождения площади треугольника по формуле Герона неважно в какой последовательности будут заданы стороны треугольника (треугольники ABC , CBA или CAB — это одно и то же). Но, если необходимо, например найти x^y и типы x и y совпадают, то есть разница в какой последовательности вызывать переменные, так как в общем случае $x^y \neq y^x$.

Таким образом, следующий листинг демонстрирует объявление, определение и вызов функции.

Пример 1.1. Дано множество точек, заданных своими координатами. Найти треугольник максимальной площади.

Листинг 1.4.

```

1 #include<iostream>
2 #include<cmath>
3 using namespace std;
4
5 double dlina (double ax, double ay, double bx, double by); //объявление функции
    расстояния м/у двумя точками
6
7 double square(double a, double b, double c){ //определение функции площади
    треугольника
8     double p = (a + b + c)/2;           //полупериметр
9     return sqrt(p*(p - a)*(p - b)*(p - c)); //ф-ла Герона
10 }
11
12 int main(){
13     int pointx[40], pointy[40]; //массивы точек
14     int n;
15     cout << "n = "; cin >> n; //кол-во точек
16     for(int i = 0; i < n; i++){ //ввод точек

```

```

17     cout << "Input koordinats " << i << " point\n";
18     cin >> pointx[i] >> pointy[i];
19 }
20 double max = 0;
21 int maxi, maxj, maxk;
22 for(int i = 0; i < n - 2; i++) //перебираем точки пирамидой,
23 for(int j = i + 1; j < n - 1; j++)//чтобы не считать совпадающие треугольники
24 for(int k = j + 1; k < n; k++){//например, 1, 2, 3 и 3, 2, 1
25     double AB = dlina(pointx[i], pointy[i], //вызов функции dlina
26                         pointx[j], pointy[j]); //i-j
27     double BC = dlina(pointx[j], pointy[j], //вызов функции dlina
28                         pointx[k], pointy[k]); //j-k
29     double AC = dlina(pointx[i], pointy[i], //вызов функции dlina
30                         pointx[k], pointy[k]); //i-k
31     double S = square(AB, BC, AC); //вызов функции square
32     if (S > max) { //поиск max
33         max = S;
34         maxi = i; //сохраняет номера точек
35         maxj = j;
36         maxk = k;
37     }
38 }
39 cout << "Max square = " << S << " in points " << maxi << ", " << maxj << ", "
    << maxk << endl;
40 return 0;
41 }
42
43 double dlina(double ax, double ay, double bx, double by){
44     return sqrt((ax - bx)*(ax - bx) + (ay - by)*(ay - by));
45 }

```

□

Вызов функции устроен следующим образом: при вызове функции сохраняется адрес памяти, куда должно быть возвращено значение функции, выделяется память под локальные переменные и работу функции. После завершения работы функции память, выделенная под работу функции, очищается и значение функции возвращается в точку «возврата».

1.5. Типы переменных

1.5.1. Локальные, глобальные и статические переменные

Переменные, определенные внутри функции, называются *локальными*. Локальные переменные инициализируются каждый раз, как только поток исполнения достигает точки их определения. Это происходит при каждом вызове функции. Каждый такой вызов приводит к созданию локальной переменной, которая существует только во время работы текущей функции.

Переменные, описанные вне всех функций, называются *глобальными*. Эти переменные доступны во всех функциях. Но если, в функции определена переменная с тем же именем, что и глобальная, то функция будет работать с локальной переменной. В этом случае возможны непредвиденные ситуации, поэтому желательно избегать глобальных переменных.

Если глобальные переменные все-таки используются, то в случае конфликта для использования именно глобальной переменной необходимо использовать операцию доступа: `::`.

Листинг 1.5.

```
1 #include <iostream>
2 using namespace std;
3
4 int a = 5; //глобальная переменная
5
6 void func(int x, int y){
7     x++;           //изменение локальных переменных
8     y--;
9     cout << "vnutri\n";
10    cout << "x = " << x << " y = " << y << " a = " << a << endl;
11 }
12
13 int main(){
14     int a = 1; //локальная переменная
15     cout << "local a = " << a << " global a = " << ::a << endl; //::a - вызов
        глобальной переменной
16     int x = 5, y = 5;
17     func(x, y); //вызов функции
18     cout << "vne\n";
```

```

19  cout << "x = " << x << " y = " << y << endl;
20  return 0;
21  }

```

□

Результат работы программы:

```

local a = 1 global a = 5
vnutri
x = 6 y = 4 a = 5
vne
x = 5 y = 5

```

Из приведенного примера видно, что переменные, определенные внутри тела функции, не существуют. Но иногда нужно, использовать значение переменных, например, подсчитать число вызовов функции. Для этого используются *статические* переменные — переменные, определенные с помощью спецификатора `static`. Такие переменные инициализируются лишь однажды при первом достижении точки определения переменной. Например,

Листинг 1.6.

```

1  #include <iostream>
2  using namespace std;
3
4  void func(int a){
5      while (a--){          //переменные n, x определяются внутри блока
6          static int n = 0; //инициализируется один раз
7          int x = 0;        //инициализируется a раз заново
8          cout << "n = " << n++ << " x = " << x++ << endl;
9      }
10 }
11
12 int main(){
13     func(3); //вызов функции
14     return 0;
15 }

```

□

Результат работы программы:

```

n = 0 x = 0

```

```
n = 1 x = 0
```

```
n = 2 x = 0
```

1.5.2. Формальные и фактические переменные

Переменные, используемые при объявлении или определении функции, называются *формальными* или *параметрами*.

Переменные, используемые при вызове функции, называются *фактическими* или *аргументами*.

Разницу можно объяснить на примере нахождения площади треугольника по формуле Герона.

Определение функции:

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

Можно сказать, что определяется площадь треугольника со сторонами a , b , c . Говорят, что площадь треугольника — $\sqrt{p(p-a)(p-b)(p-c)}$, где $p = \frac{a+b+c}{2}$ — полупериметр треугольника. Это символьная, или формальная, запись.

Вызов функции:

```
double S = square(3, 4, 5);
```

Говорим, что площадь треугольника со сторонами 3, 4, 5 равна 6. Это площадь конкретного (фактического) треугольника.

1.5.3. Аргументы по умолчанию

Очень часто функции имеют больше аргументов, чем это необходимо. В таких случаях используют аргументы по умолчанию (особенно часто это встречается в конструкторе классов.) Для аргументов по умолчанию параметрам в списке может присваиваться некоторое значение:

Например, функция, печатающая числа в любой системе счисления. По умолчанию, числа печатаются в десятичной системе счисления.

```
void print(int value, int base = 10);

int main(){
    print(31); //вызов по умолчанию
    print(31, 8);
    print(31, 2);
}
```

Если функция вызывается с одним аргументом, то в качестве второго параметра используется значение по умолчанию. Если функция вызывается с двумя аргументами, в качестве второго параметра используется введенное значение. В примере, первый вызов напечатает число 31 в десятичной системе счисления, второй вызов — число 31 в восьмеричной системе счисления.

Аргументами по умолчанию могут быть аргументы, находящиеся только в конце списка параметров:

```
int f(int a, int b = 0, int c = 1); //правильно
int f(int a = 0, int b, int c = 1); //неправильно
```

1.6. Передача данных в функцию

Существует два способа передачи данных в функцию: по значению и по адресу. Каждый из этих способов применяется часто, в зависимости от ситуации.

Передача данных по значению используется чаще всего.

Листинг 1.7.

```
1 #include<iostream>
2 using namespace std;
3
4 int func (int a, int b){
5     cout << "function:" << endl;
6     cout << "address: " << &a << " " << &b << endl;
7     a = a + 1;
8     b = b + 2;
9     cout << "value: " << a << " " << b << endl;
10    return a + b;
11 }
12
13 int main(){
14     int a = 5;
15     int b = 3;
16     cout << "before: " << endl;
17     cout << "value: " << a << " " << b << endl;
18     cout << "address: " << &a << " " << &b << endl;
19     int c = func(a,b);
20     cout << "after: " << endl;
```

```

21     cout << "value: " << a << " " << b << endl;
22     cout << "address: " << &a << " " << &b << endl;
23     return 0;
24 }

```

Результат работы программы:

before:

value: 5 3 — значение переменных до вызова функции

address: 0042F95C 0042F950 — адрес памяти, где расположены переменные

function:

address: 0042F86C 0042F870 — адрес переменных, используемых в функции.

value: 6 5 — значение переменных, используемых в функции.

after:

value: 5 3 — значение переменных после вызова функции

address: 0042F95C 0042F950 — адрес памяти, где расположены переменные.

В приложении 1.7 приведен пример передачи данных в функцию по значению.

При вызове функции создаются локальные переменные, выступающие в качестве параметров функции, и им присваиваются значения переменных, выступающих в качестве аргументов при вызове функции.

В строке 18 приведена операция вывода на экран адресов переменных до вызова функции. Как видно из примера, эти адреса могут быть такими: **address:** 0042F95C 0042F950. Для функции **func** эти переменные являются фактическими параметрами.

В 19 строке вызывается функция. Переменные параметры (формальные переменные) и аргументы (фактические переменные) имеют одинаковые имена, но из результатов работы программы видно, что переменные, используемые в функции, расположены по другим адресам: **address:** 0042F86C 0042F870, то есть, это абсолютно другие переменные. И, следовательно, изменение значений переменных, используемых в функции, не приводит к изменению значений переменных, используемых в функции **main()**, что видно из результатов работы программы, где выведены значения переменных **a** и **b** до вызова функции, во время работы функции и после вызова функции.

Таким образом, при передаче параметров по значению происходит следующие:

1. Создаются переменные, выступающие в качестве формальных параметров.
2. Этим переменным присваиваются значения соответствующих фактических параметров.

3. После окончания работы функции созданные переменные уничтожаются.

Передача данных по адресу может происходить двумя способами: по указателю и по ссылке.

Пример передачи данных по указателю приведен в приложении 1.8.

Листинг 1.8.

```
1 #include<iostream>
2 using namespace std;
3
4 int func (int *a, int *b){
5     cout << "function:" << endl;
6     cout << "address: " << a << " " << b << endl;
7     *a = *a + 1;
8     *b = *b + 2;
9     cout << "value: " << *a << " " << *b << endl;
10    return *a + *b;
11 }
12
13 int main(){
14     int a = 5;
15     int b = 3;
16     cout << "before: " << endl;
17     cout << "value: " << a << " " << b << endl;
18     cout << "address: " << &a << " " << &b << endl;
19     int c = func(&a,&b);
20     cout << "after: " << endl;
21     cout << "value: " << a << " " << b << endl;
22     cout << "address: " << &a << " " << &b << endl;
23     return 0;
24 }
```

Результат работы программы:

before:

value: 5 3 — значение переменных до вызова функции

address: 0028F908 0028F8FC — адрес памяти, где расположены переменные

function:

address: 0028F908 0028F8FC — адрес переменных, используемых в функции.

value: 6 5 — значение переменных, используемых в функции.

after:

value: 6 5 — значение переменных после вызова функции

address: 0028F908 0028F8FC — адрес памяти, где расположены переменные.

Как уже говорилось, указатель — это переменная, в которой хранится адрес памяти, на которую «указывает» этот указатель.

При передаче данных через указатели в функцию создаются новые переменные, являющиеся указателями, которым присваиваются адреса соответствующих фактических переменных.

Как видно из результатов работы программы, теперь функция выполняет действия над переменными, расположенными по тому же адресу, что и фактические переменные, то есть, над теми же самыми переменными и, следовательно, изменение значения переменных внутри функции приведет к изменению значения этих же переменных вне функции.

Таким образом, передача по адресу изменяет значение переменных в функции `main()`, поэтому с передачей данных по адресу надо быть очень осторожными, чтобы не получить неожиданных результатов.

Основной недостаток работы с указателями, что приходится не забывать про операцию разыменования и взятия адреса, что может привести к ошибкам. Следовательно, передачу данных через указатели лучше использовать только в случае, когда вся программа работает с указателями, например, при работе с массивами.

Передача данных по ссылке используется в случае, когда необходимо передать данные по адресу (например, необходимо изменение фактических параметров, или в результате работы функции надо получить больше одного результата).

При вызове функции по ссылке создается псевдокопии переменных, которые и используются при работе функции.

Пример передачи данных по ссылке приведен в приложении 1.9.

Листинг 1.9.

```
1 #include<iostream>
2 using namespace std;
3
4 int func (int &a, int &b){
5     cout << "function:" << endl;
6     cout << "address: " << &a << " " << &b << endl;
7     a = a + 1;
8     b = b + 2;
```

```

9   cout << "value: " << a << " " << b << endl;
10  return a + b;
11 }
12
13 int main(){
14     int a = 5;
15     int b = 3;
16     cout << "before: " << endl;
17     cout << "value: " << a << " " << b << endl;
18     cout << "address: " << &a << " " << &b << endl;
19     int c = func(a,b);
20     cout << "after: " << endl;
21     cout << "value: " << a << " " << b << endl;
22     cout << "address: " << &a << " " << &b << endl;
23     return 0;
24 }

```

Результат работы программы:

before:

value: 5 3 — значение переменных до вызова функции

address: 0018F858 0018F84C — адрес памяти, где расположены переменные

function:

address: 0018F858 0018F84C — адрес переменных, используемых в функции.

value: 6 5 — значение переменных, используемых в функции.

after:

value: 6 5 — значение переменных после вызова функции

address: 0018F858 0018F84C — адрес памяти, где расположены переменные.

1.7. Рекурсивные функции

Выше уже встречались с понятием рекуррентных соотношений. В случае функций также можно воспользоваться рекурсивным вызовом функции.

Рекурсивная функция — вызов функцией самой себя.

Примеров рекурсивной функции достаточно много. Рекурсия встречается во всех областях науки.

Данные в рекурсивные функции передаются по значению, поэтому при каждом новом вызове происходит выделение дополнительной памяти под новые переменные, что при

достаточной глубине рекурсии может привести к переполнению памяти. Именно это является основным недостатком рекурсивных функций, поэтому, лучше пользоваться обычными функциями, а не рекурсивными.

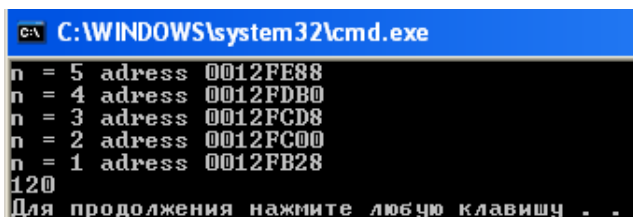
Обычно работа рекурсивной функции представляет собой цепочку вызовов этой рекурсивной функции. Этот процесс происходит до тех пор, пока что-то не прервет эту цепочку. Чаще всего для этого используется оператор `if`, например, если условие истинное, то цепочка вызовов рекурсии прерывается.

Пример 1.2. Примером работы рекурсивной функции служит функция вычисления факториала:

Листинг 1.10.

```
1 #include<iostream>
2 using namespace std;
3
4 int f(int n){
5     if (n < 0) return -1; //выход из рекурсии для отриц. чисел
6     if (n == 0 || n == 1) { //выход из рекурсии
7         cout << "n = " << n << " address " << &n << endl;
8         return 1;
9     }
10    else {
11        cout << "n = " << n << " address " << &n << endl;
12        return n*f(n - 1); //рекурсивный вызов
13    }
14 }
15
16 int main(){
17     cout << f(5) << endl;
18     return 0;
19 }
```

Результат работы программы:

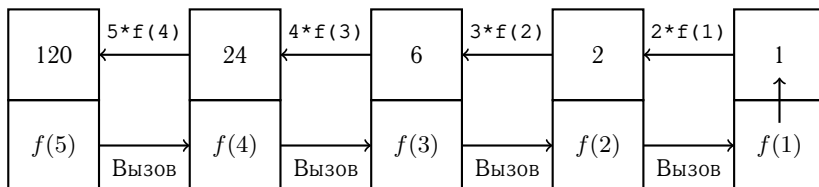


```
C:\WINDOWS\system32\cmd.exe
n = 5 address 0012FE88
n = 4 address 0012FDB0
n = 3 address 0012FCD8
n = 2 address 0012FC00
n = 1 address 0012FB28
120
Для продолжения нажмите любую клавишу . .
```

Видно, что действительно каждый раз выделяется память под новые переменные.

На схеме ниже показана работа рекурсивной функции f . Сначала вызывается функция $f(5)$ (строка 16 Листинга 1.10), потом последовательно вызываются функции $f(4)$, $f(3)$, $f(2)$, $f(1)$ (строка 12). При $n == 1$ выражение в операторе `if` (строка 6) становится истинным, следовательно, происходит выход из рекурсии.

Результат работы функции $f(1)$ равен единице (строка 8). Это значение возвращается в функцию $f(2)$. Результат работы этой функции равен $2 * 1$ (строка 12). Это значение возвращается в функцию $f(3)$ и т. д.



□

Как видно из примера, надо обязательно следить, что всегда была возможность выхода из рекурсии. Например, в случае поиска факториала отдельно рассмотрен выход из рекурсии в случае отрицательного числа (-1 или сообщение, что факториал определен только для положительных чисел), отдельно для 0 и 1 (по определению факториала $0! = 1$). Если не рассмотреть случай отрицательного числа и вызвать функцию, допустим, для $n = -5$, то условие выхода из рекурсии всегда будет ложным и рекурсия станет бесконечной. Программа завершится аварийно после переполнения памяти.

Рассмотрим пример рекурсивной функции, не возвращающей значение.

Пример 1.3. Вывести на экран цифры числа в прямом порядке и обратном порядке.

Листинг 1.11.

```

1  #include<iostream>
2  using namespace std;
3
4  void func_0_n(int n){
5      if (n > 0){
6          func_0_n(n/10); //вызов рекурсии
7          cout << n % 10 << " ";
8      }
9  }
10
11 void func_n_0(int n){

```

```

12  if (n > 0){
13      cout << n % 10 << " ";
14      func_n_0(n/10); //вызов рекурсии
15
16  }
17  }
18  int main(){
19      int n = 12345678;
20      func_0_n(n); //вызов в прямом порядке
21      cout << endl;
22      func_n_0(n); //вызов в обратном порядке
23      cout << endl;
24      return 0;
25  }

```

Результат работы программы:

1 2 3 4 5 6 7 8

8 7 6 5 4 3 2 1

□

Поскольку при отделении цифр числа отбрасывается последняя цифра, то в функции `func_0_n(n)` сначала вызываются функции (строка 6 Листинга 1.11):

`func_0_n(123456789) → func_0_n(12345678) → func_0_n(1234567) → func_0_n(123456) → func_0_n(12345) → func_0_n(1234) → func_0_n(123) → func_0_n(12) → func_0_n(1) → func_0_n(0)`.

Потом на экран выводится последняя цифра числа (строка 7:)

`func_0_n(1) → func_0_n(12) → func_0_n(123) → func_0_n(1234) → func_0_n(12345) → func_0_n(123456) → func_0_n(1234567) → func_0_n(12345678) → func_0_n(123456789)`.

Соответственно, цифры выводятся в прямом порядке.

В функции `func_n_0(n)` сначала на экран выводится последняя цифра числа (строка 13 Листинга 1.11):

`func_0_n(123456789) → func_0_n(12345678) → func_0_n(1234567) → func_0_n(123456) → func_0_n(12345) → func_0_n(1234) → func_0_n(123) → func_0_n(12) → func_0_n(1) → func_0_n(0)`.

Потом вызываются функции (строка 14:)

`func_0_n(123456789) → func_0_n(12345678) → func_0_n(1234567) → func_0_n(123456) → func_0_n(12345) → func_0_n(1234) → func_0_n(123)`

→func_0_n(12) → func_0_n(1) → func_0_n(0).

Соответственно, цифры выводятся в обратном порядке.

1.8. Перегрузка функций

В языке C++ существует возможность перегрузки функции, когда в одной программе можно использовать несколько функций с одинаковыми именами, но выполняющими разные значения.

Компилятор определяет какую из функций использовать при вызове по сигнатуре вызываемой функции (список аргументов функции).

При использовании перегруженных функций необходимо следить за соответствием количества и типом формальных и фактических параметров, поскольку в данном случае неявного преобразования типов не происходит.

Принято считать, что передача данных по ссылке и по значению — это одно и то же, так как запись вызова функции при передаче по значению и по ссылке одинакова.

Пример работы перегруженной функции.

Пример 1.4. Создание и вывод двумерного массива различных типов данных:

Листинг 1.12.

```
1 #include<iostream>
2 using namespace std;
3
4 int **create(int c, int n, int m){//создание int
5     int **a = new int *[n]; //выделение памяти
6     for (int i = 0; i < n; i++)
7         a[i] = new int [m];
8     for (int i = 0; i < n; i++) //ввод массива
9         for (int j = 0; j < m; j++)
10             cin >> a[i][j];
11     return a;
12 }
13
14 void print(int **a, int n, int m){ //печать int
15     for (int i = 0; i < n; i++,cout << endl)
16         for (int j = 0; j < m; j++)
17             cout << a[i][j] << " ";
18 }
```

```

19
20 double **create(double c, int n, int m){//создание int
21     double **a = new double *[n]; //выделение памяти
22     for (int i = 0; i < n; i++)
23         a[i] = new double [m];
24     for (int i = 0; i < n; i++) //ввод массива
25         for (int j = 0; j < m; j++)
26             cin >> a[i][j];
27     return a;
28 }
29
30 void print(double **a, int n, int m){ //печать int
31     for (int i = 0; i < n; i++,cout << endl)
32         for (int j = 0; j < m; j++)
33             cout << a[i][j] << " ";
34 }
35
36 char **create(char c, int n, int m){//создание int
37     char **a = new char *[n]; //выделение памяти
38     for (int i = 0; i < n; i++)
39         a[i] = new char [m];
40     for (int i = 0; i < n; i++) //ввод массива
41         for (int j = 0; j < m; j++)
42             cin >> a[i][j];
43     return a;
44 }
45
46 void print(char **a, int n, int m){ //печать int
47     for (int i = 0; i < n; i++,cout << endl)
48         for (int j = 0; j < m; j++)
49             cout << a[i][j] << " ";
50 }
51
52 int main(){
53     int n,m;
54     cout << "Input dimension\n";
55     cin >> n >> m;
56     int c = 1; //определение для int
57     cout << "Input int matrix:\n";
58     int **a = create(c,n,m);

```

```

59  cout << "Print int matrix:\n";
60  print(a,n,m);
61  cout << "Input double matrix:\n";
62  double d = 1;//определение для double
63  double **a1 = create(d,n,m);
64  cout << "Print double matrix:\n";
65  print(a1,n,m);
66  cout << "Input char matrix:\n";
67  char d1 = '1';//определение для char
68  char **a2 = create(d1,n,m);
69  cout << "Print char matrix:\n";
70  print(a2,n,m);
71  return 0;
72  }

```

Результат работы программы:

```

C:\WINDOWS\system32\cmd.exe
Input dimension
2 2
Input int matrix:
1 2
3 4
Print int matrix:
1 2
3 4
Input double matrix:
1.1 2.1
1.2 3.3
Print double matrix:
1.1 2.1
1.2 3.3
Input char matrix:
v d
w a
Print char matrix:
v d
w a
Для продолжения нажмите любую клавишу . . . _

```

□

Созданы три функции для создания и заполнения массивов (`create`) для массивов типа `int` (строки 4–12), типа `double` (строки 20–28) и типа `char` (строки 36–44) Листинга 1.12 и три функции для вывода массивов на экран (`print`) для массивов типа `int` (строки 14–18), типа `double` (строки 30–34) и типа `char` (строки 46–50).

В функции `main()` определены три переменные `c`, `d`, `d1`, тип которых и определяет какую из функций необходимо вызвать.

1.9. Шаблоны функций

Как видно из предыдущего примера, можно создать несколько функций, имеющих одно имя. Если внимательно присмотреться к этим функциям, то видно, что они отличаются только типами данных, а в остальном одинаковы.

Следовательно, для упрощения программы можно воспользоваться шаблоном функции. То есть написать одну функцию для переменных некоторого типа `X`, а уже при вызове функции по списку параметров определять переменные какого типа подставлять вместо типа `X`.

Для написания шаблона функции необходимо перед каждой функцией написать ключевую фразу:

```
template <typename имя типа>.
```

Здесь треугольные скобки являются обязательным элементом.

В списке формальных параметров функции обязательно должен быть параметр типа `X`.

Пример работы перегруженной функции.

Пример 1.5. Создание и вывод двумерного массива различных типов данных:

Листинг 1.13.

```
1 #include<iostream>
2 using namespace std;
3
4 template <typename X>
5 X **create(X c, int n, int m){//создание
6   X **a = new X *[n]; //выделение памяти
7   for (int i = 0; i < n; i++)
8     a[i] = new X [m];
9   for (int i = 0; i < n; i++) //ввод массива
10     for (int j = 0; j < m; j++)
11       cin >> a[i][j];
12   return a;
13 }
```

```

14
15 template <typename X>
16 void print(X **a, int n, int m){ //печать
17     for (int i = 0; i < n; i++,cout << endl)
18         for (int j = 0; j < n; j++)
19             cout << a[i][j] << " ";
20 }
21
22 int main(){
23     int n,m;
24     cout << "Input dimension\n";
25     cin >> n >> m;
26     int c = 1; //определение для int
27     cout << "Input int matrix:\n";
28     int **a = create(c,n,m);
29     cout << "Print int matrix:\n";
30     print(a,n,m);
31     cout << "Input double matrix:\n";
32     double d = 1; //определение для double
33     double **a1 = create(d,n,m);
34     cout << "Print double matrix:\n";
35     print(a1,n,m);
36     cout << "Input char matrix:\n";
37     char d1 = '1'; //определение для char
38     char **a2 = create(d1,n,m);
39     cout << "Print char matrix:\n";
40     print(a2,n,m);
41     return 0;
42 }

```

Создана функция для создания и заполнения массивов (**create**) (строки 4–13) и функция для вывода массивов на экран (**print**) (строки 15–20) Листинга 1.13 для массивов любого типа

В функции **main()** определены три переменные **c**, **d**, **d1**, тип которых и определяет параметры какого типа используются в соответствующей функции.

Результат работы программы:


```
C:\WINDOWS\system32\cmd.exe
Input dimension
2 2
Input int matrix:
1 2
3 4
Print int matrix:
1 2
3 4
Input double matrix:
1.1 2.1
1.2 3.3
Print double matrix:
1.1 2.1
1.2 3.3
Input char matrix:
v d
w a
Print char matrix:
v d
w a
Для продолжения нажмите любую клавишу . . . _
```

□