top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 8

Функции

Содержание

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Введение в мир функций

Необходимость использования. Объявление. Вызов

Зачастую складывается такая ситуация, когда в нашей программе некоторые отрезки кода повторяются несколько раз и нам приходится много раз подряд набирать один и тот же фрагмент кода. Такая ситуация имеет свои отрицательные стороны. Во-первых, процесс создания программы становится более нудным и длительным, во-вторых, увеличивается объем конечного файла. Как быть в этом случае?! Существует ли какой-нибудь механизм позволяющий автоматизировать действия программиста и сократить код программы?! Да, существует, — отвечаем вам мы. Называется такой механизм — функцией. Итак:

Функция — специальная конструкция, с помощью которой какой-либо фрагмент кода повторяющийся в программе два или более раз выносится за тело программы(в нашем случае за пределы main). Этот фрагмент получает собственное имя и, в дальнейшем, для того, чтобы воспользоваться вынесенным кодом, необходимо будет указать это имя.

Синтаксис объявления

Существует два способа объявления функции:

- Функция объявляется до функции main.
- Функция объявляется с помощью прототипа (о том, что такое прототип вы узнаете чуть позднее в этом

уроке), а после функции main, описывается тело объявленной функции.

Первый способ: до функции main

Общий синтаксис объявления функции:

```
возвращаемое_значение имя_функции (параметры) {
    блок_повторяющегося_кода (тело);
}
```

1. Имя функции подчиняется тем же правилам, что и имя переменной и, естественно, выбирается программистом. Имя должно соответствовать правилам именования в C++
2. **Параметры** — входные данные, которые необходимы функции для работы над кодом. В качестве параметров используют обычные переменные, указывая для каждого параметра его тип данных. Если функция не нуждается во входных данных, скобки следует оставить пустыми. Второе название параметров — аргументы.
3. **Возвращаемое значение** — результат работы функции. На место возвращаемого значения подставляется любой из базовых типов. Это тип тех данных, которые функция поставит на место своего вызова в программе. Если функция ничего не возвращает, на место возвращаемого значения подставляется void (*пусто*). В общем и целом «осмысленное» возвращаемое значение указывается в том случае, если результат работы функции необходим для дальнейших вычислений.

Примечание

- *Нельзя создавать одну функцию внутри другой.*
- *Нельзя вызвать функцию до ее объявления.*

Синтаксис вызова функции

Для того чтобы воспользоваться функцией на определенном отрезке кода, следует вызвать ее прямо на этом отрезке. Вызов функции является предписанием операционной системе начать выполнение фрагмента кода, который содержится в теле функции. По завершении выполнения функции, программа должна продолжить работу в основном коде с того же места, где была вызвана функция. Вызов функции состоит из указания имени функции, передачи аргументов (если таковые имеются) и получения возвращаемого значения (если есть необходимость его получать):

```
имя_переменной = имя_функции(параметр1,
                              параметр2,
                              ....,
                              параметрN);
```

1. Типы данных значений должны соответствовать типам данных аргументов в определении функции. Исключением являются случаи, когда передаваемое значение может быть с легкостью преобразовано к нужному типу.
2. При вызове функции всегда необходимо указывать такое количество параметров, какое было определено при объявлении.
3. Тип переменной, в которую запишется возвращаемое значение, должен совпадать с тем типом, который функ-

ция, собственно, возвращает. Это не обязательно, но желательно.

Ключевое слово return

Для возвращения Значения из функции в программу на то место, из которого была вызвана эта функция, используется оператор return. Синтаксис возврата таков:

```
return значение;
```

Если функция не возвращает никаких значений, то оператор return можно использовать просто для остановки функции, для этого просто пишем:

```
return; // в данном случае return отработает для
        // функции, как break для цикла.
```

Запомните важные моменты обращения с return:

1. Операторов возврата может быть несколько (в зависимости от ситуации), но отработает только один из них.
2. Если сработал return (вне зависимости от формы), все что расположено в функции ниже него уже не отработает.
3. Если тип возвращаемый функцией не void, то необходимо ВСЕГДА использовать форму: return-значение;

Теперь, когда с теоретической частью мы знакомы, переходим к следующему разделу — примеры на создание функций.

2. Примеры на создание и вызов функций

Функция, которая не принимает никаких параметров и не возвращает никакого значения.

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

// создание функции
void Hello(){
    // показ на экран строки текста
    cout<<"Hello, World!!!\n\n";
}

int main(){
    Hello(); // вызов
    Hello(); // вызов
    Hello(); // вызов
    return 0;
}
```

Результат — три раза фраза — *Hello, World!!!* — на экране.

Функция, которая принимает один параметр, но не возвращает никакого значения

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

// рисует линию из звездочек длиной count
void Star(int count){
    for(int i=0;i<count;i++)
```

```
        cout<<'*';
    cout<<"\n\n";
}

int main(){
    Star(3); // показ линии из трех звездочек
    Star(5); // показ линии из пяти звездочек
    return 0;
}
```

Функция, которая принимает два параметра, но все еще не возвращает никакого значения

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

// рисует линию из символа - symb, длиной count
void AnyLine(char symb, int count){
    for(int i=0;i<count;i++)
        cout<<symb;
    cout<<"\n\n";
}

int main(){
    AnyLine('+',3); // показ линии из трех плюсов
    AnyLine('=',5); // показ линии из пяти знаков
                    // равно
       return 0;
}
```

Функция, которая принимает два параметра, и возвращает значение

```
#include <iostream>
using namespace std;
// вычисляет степень (Pow) числа (Digit)
int MyPow(int Digit, int Pow){
```

```
    int key=1;
    for(int i=0;i<Pow;i++)
        key*=Digit;
        return key;
}

int main(){
    // запись возвращаемого результата в переменную
    res int res=MyPow(5,3);
    cout<<"Res = "<<res<<"\n\n";
    return 0;
}
```

Функция, которая принимает два параметра и возвращает максимальное из двух значений:

```
#include <iostream>
#include <string>

using namespace std;
int Max(int a, int b){
    return a>b? a: b;
}

int main()
{
    // Значение, которое вернул
    // return можно записать в переменную
    int result = Max(10,30);
    cout<<"Maximum is: "<<result<<endl;

    // Значение, которое вернул
    // return можно сразу вывести на экран
    cout<<"Maximum is: "<<Max(42,2);
    return 0;
}
```

Добавим, к примеру выше, ещё функцию подсчета минимума.

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

int Min(int a, int b){
    return a<b? a: b;
}

int Max(int a, int b){
    return a>b? a: b;
}

int main()
{
    // Значение, которое вернул return
    // можно записать в переменную
    int result = Max(10,30);

    cout<<"Maximum is: "<<result<<endl;

    // Значение, которое вернул return
    // можно сразу вывести на экран
    cout<<"Minimum is: "<<Min(10,30);

    return 0;

}
```

В этом примере у нас есть две функции, которые мы вызываем внутри main. Каждый раз, когда мы вызываем функцию мы переходим внутрь её тела. После выполнения функции программа возвращается в то место, откуда был произведен вызов.

Мы можем вызывать функцию внутри другой функции. Рассмотрим этот механизм на примере:

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

void Second(){
    cout<<"\nSecond function\n";
}

void First(){
    cout<<"\nBegin first function\n";
    Second();
    cout<<"\nEnd first function\n";
}

int main() {
    First();
    return 0;

}
```

Вывод на консоль :

```
Begin first function
Second function
End first function
```

Внутри main мы вызываем функцию First. Она отображает первое информационное сообщение и вызывает функцию Second. Выполняется тело функции Second (отображение сообщения), после чего выполнение программы снова возвращается внутрь функции First (отображается второе информационное сообщение). После выполнения

функции First программа возвращается на место её вызова внутрь main.

Можем ли мы внутри main отдельно вызвать функцию Second? Безусловно. Нет никакого правила, которое запретит вам это делать. Связь между First и Second только в том, что First вызывает Second для своих нужд.

3. Передача аргументов. Прототипы функций

Передача аргументов по значению

Поговорим, о том, что происходит в оперативной памяти. Аргументы, которые указываются при определении функции, называются формальными. Это связанно с тем что, они создаются в момент вызова функции в оперативной памяти. При выходе из функции такие параметры будут уничтожены. Поэтому, если в другой функции программы будут параметры с тем же именем, то конфликта не будет. Рассмотрим, один из способов передачи аргументов:

Пример работы формальных параметров при передаче данных по значению

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

// Должна менять значения переменных местами
void Change(int One, int Two){
    // 1 2
    cout<<One<<" "<<Two<<"\n\n";
    int temp=One;
    One=Two;
    Two=temp;
    cout<<One<<" "<<Two<<"\n\n";// 2 1


}
```

```
int main(){
    int a=1,b=2;
    cout<<a<<" "<<b<<"\n\n"; // 1 2
    // передача по значению
    Change(a,b);
    cout<<a<<" "<<b<<"\n\n"; // 1 2
    return 0;
}
```

1. В функцию передаются не a и b, а их точные копии.
2. Все изменения происходят с копиями (One и Two), при этом сами a и b остаются неизменными.
3. При выходе из функции временные копии уничтожаются.

Исходя из вышеописанного — пока будьте внимательны при обработке значений внутри функции. Впоследствии, мы научимся решать данную проблему.

Кое-что о массивах...

Массивы можно передавать внутрь функции в качестве аргументов. Важным отличием от обычных переменных, является то, что все изменения, происходящие с массивом внутри функции — сохраняются при выходе из неё. Для передачи одномерного массива достаточно просто указать пустые квадратные скобки:

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

Если вы укажите размер внутри пустых квадратных скобок это ни на что не повлияет.

Рассмотрим пример заполнения одномерного массива случайными числами и отображения результатов в консоль с помощью функций.

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

// отображение массива
void ShowArray(int arr[], int size){
    for(int i = 0; i < size; i++){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}

/* инициализируем массив внутри функции случайными
   числами в отличии от обычных переменных изменения
   элементов массива, происходящие внутри функции
   при выходе из функции сохраняются
*/

void InitArray(int arr[], int size){
    for(int i = 0; i < size; i++){
        arr[i] = rand()%100;
    }
    cout<<endl;
}

int main()
{
    srand(time(NULL));
    const int aSize = 10;
    int iArr[aSize];
```

```
    InitArray(iArr,aSize);
    ShowArray(iArr,aSize);
    return 0;
}
```

Ещё раз обращаем ваше внимание, что содержимое массива может измениться внутри функции и эти изменения сохранятся при выходе из неё. Именно поэтому после выполнения функции InitArray массив остаётся заполненным случайными значениями.

Если в функцию передаётся двумерный массив, то описание соответствующего аргумента функции должно содержать количество столбцов; количество строк — несущественно. Например:

```
int summa (int array[][5], int size_row, int size_col){
    int res=0;
    for (int i = 0; i < size_row; i++)
    for (int j = 0; j < size_col; j++)
    res +=array[i][j];
    return res;

}
```

Рассмотрим пример заполнения двумерного массива случайными числами и отображения результатов в консоль с помощью функций.

```
#include <iostream>
#include <string>
#include<time.h>

using namespace std;
```

```
// отображение матрицы
void ShowMatrix(int matr[][3], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++ ) {
            cout << matr[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/* инициализируем матрицу внутри функции случайными
   числами в отличии от обычных переменных изменения
   элементов массива, происходящие внутри функции
   при выходе из функции сохраняются
*/

void InitMatrix(int matr[][3], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matr[i][j] = rand() % 10;
        }
    }
    cout << endl;
}

int main()
{
    srand(time(NULL));
    const int mRows = 3;
    const int mCols = 3;
    int iMatr[mRows][mCols];
    InitMatrix(iMatr, mRows,mCols);
    ShowMatrix(iMatr, mRows, mCols);
    return 0;
}
```

Пример похож на уже рассмотренный ранее код для одномерного массива. Отличие состоит только в способе передачи параметров и обращении к элементам матрицы.

Прототипы функций или второй способ объявления

При втором способе объявления функции необходимо сообщить компилятору о том, что функция существует. Для этого до main предоставляется имя функции, ее аргументы, а также тип возвращаемого значения. Такую конструкцию называют прототипом функции. Когда компилятор встречает прототип функции он точно знает о том, что функция существует в программе после main — и, она должна там быть.

Общий синтаксис:

```
библиотеки

возвращаемое_значение имя_функции(аргументы);

void main(){
    тело main;
}

возвращаемое_значение
    имя_функции(аргументы){ тело функции;
}
```

```
#include <iostream>

using namespace std;
// прототипы
void MyFunc();
```

```
void MyFuncNext();
int main(){
    MyFunc(); // MyFunc
    MyFuncNext(); // MyFuncNext
    return 0;
}

// описания
void MyFunc(){
    cout<<"MyFunc\n";
}

void MyFuncNext(){
    cout<<"MyFuncNext\n";
}
```

Вы можете использовать и тот и другой механизм объявления функции.

4. Область видимости. Глобальные и локальные переменные

Область видимости

Любые фигурные скобки в программном коде образуют, так называемую область видимости, это означает, что переменные, объявленные внутри этих скобок будут видны только внутри этих скобок. Другими словами, если к переменной, созданной внутри функции, цикла, if и так далее обратится из другого места программы, то произойдет ошибка, так как после выхода из фигурных скобок эта переменная будет уничтожена.

```
int a=5;
if(a==5){
    int b=3;
}
// ошибка! b не существует
// она существовала только внутри if
cout<<b;
```

Глобальные и локальные переменные

Согласно правилам области видимости — переменные делятся на два вида — локальные и глобальные.

Локальные переменные создаются внутри какого — нибудь отрезка кода, что это значит для программы, мы уже знаем.

Глобальные переменные создаются вне всяких областей видимости. Преимущественно до функции main(). Такая переменная видна в любом месте программы. По умолчанию глобальные переменные в отличии от локальных инициализируются 0. И, главное, те изменения, которые происходят с глобальной переменной внутри функции, при выходе из последней сохраняются.

> **Примечание:** *Запомните — если есть, например, глобальная переменная под названием a, а внутри функции определяется переменная под тем же именем, то тогда внутри функции будет использоваться переменная, объявленная внутри этой функции. Поэтому избегайте применения имен переменных, которые незримо уже используются во внешних областях видимости. Этого можно достигнуть, вообще избегая использования в программе одинаковых идентификаторов.*

```
int a=23; // глобальная a
int main(){
    int a=7; // локальная a
    cout<<a; // 7, используется локальная
    return 0;

}
```

Рассмотрим ещё один пример работы с глобальными переменными. В нём мы продемонстрируем потенциальные проблемы со скрытием глобальной переменной, из-за наличия локальной переменной с таким же названием.

```
#include <iostream>
using namespace std;
int A = 10;
void SetA() {
    A = 99;
}

void Show() {
    cout << A << endl;
}
/* Внутри функции глобальная переменная будет не
   видна, так как её скрывает локальная переменная A
*/
void SetASecond(){
    int A = 77;
    cout << A <<endl;
}

int main()
{
    /* на экране отобразится значение 10 из
       глобальной переменной A
    */
    Show();
    /* записывам в глобальную переменную значение 99 */
    SetA();
    /* на экране отобразится значение 99 из
       глобальной переменной A
    */
    Show();
    /* изменяем значение локальной переменной A
       на экране будет значение 77
    */
    SetASecond();
    /* на экране отобразится значение 99 из
       глобальной переменной A
    */
```

```
    Show();
    return 0;
}
```

Если вам всё-таки необходимо обратиться к глобальной переменной при наличии локальной переменной с таким же именем, используйте оператор расширения области видимости «::»

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

int A = 10;

/* Внутри функции глобальная переменная будет не
   видна, так как её скрывает локальная переменная A.
   Однако, если нам всё-таки нужно обратиться к
   глобальной переменной, будем использовать
   оператор ::
*/
void Test() {
    int A = 77;

    // обращаемся к локальной переменной
    // на экране 77
    cout << A << endl;

    // обращаемся к глобальной A
    // на экране 10
    cout << ::A << endl;

    // меняем значение в глобальной переменной
    ::A = 333;
}
```

```
int main()
{
    Test();

    // отображается значение 333
    cout << A << endl;
    return 0;
}
```

В этом примере мы продемонстрировали, как просто обратится к глобальной переменной при наличии локальной с таким же именем.

5. Аргументы (параметры) по умолчанию

Формальному параметру функции может быть задан аргумент по умолчанию. Это означает, что в данный аргумент значение при вызове можно не передавать. В этом случае будет использовано значение по умолчанию.

Общий синтаксис для реализации такого подхода имеет следующий вид:

```
тип_возвращаемого_значения имя_функции(тип_арг имя_арг =
                                    значение_по_умолчанию)
```

Здесь значение_по_умолчанию и есть значение, присваиваемое аргументу, если он опущен при вызове. Разумеется, аргументов по умолчанию может быть несколько:

```
тип_возвращаемого_значения имя_функции(арг1=значение,
                                       арг2=значение)
```

Аргументами по умолчанию могут быть аргументы, начиная с правого конца списка параметров функции и далее последовательно слева направо без перерывов. Например:

```
void foot (int i, int j = 7); // допустимо
void foot (int i, int j = 2, int k); // недопустимо
void foot (int i, int j = 3, int k = 7); // допустимо
// допустимо
void foot (int i = 1, int j = 2, int k = 3);
void foot (int i=- 3, int j); // недопустимо
```

Рассмотрим пример на работу с параметрами по умолчанию.

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

// рисует линию из звездочек длиной count
void Star(int count=20){
    for(int i=0;i<count;i++)
        cout<<'*';
    cout<<"\n\n";
}

int main(){
    Star(); // показ линии из 20 звездочек
    Star(5); // показ линии из пяти звездочек
    return 0;
}
```

Вот и все на сегодня. А теперь — дерзайте! Вам обязательно нужно выполнить домашнее задание, для закрепления материала. Удачи!

Статическая переменная

Как мы уже знаем локальные переменные при выходе из их области видимости уничтожаются. Например:

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

void SomeFunc() {
    int a = 0;
    a++;
    cout << a << endl;
}
```

```
int main()
{
    SomeFunc();
    SomeFunc();
    SomeFunc();
    return 0;

}
```

В консоли будет выведено:

```
1
1
1
```

Каждый раз, когда функция SomeFunc будет доходить до места объявления переменной a, она будет создаваться и инициализироваться 0. При завершении работы функции переменная будет уничтожаться. В нашем случае переменная создавалась и уничтожалась три раза. А что, если нам нужно сделать так, чтобы переменная a не уничтожалась при выходе из функции и сохраняла своё значение между вызовами?

Для решения этой задачи нам нужно использовать **статическую локальную переменную**. Этот вид локальных переменных не уничтожается при выходе за пределы области видимости. Такие переменные создаются один раз при первом обращении и уничтожаются, когда программа заканчивает своё исполнение. Для создания локальной статической переменной необходимо использовать ключевое слово static.

Рассмотрим на примере его использование.

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

void SomeFunc() {
    /*
       Переменная создаётся при первом вызове функции
       После выхода из функции она не уничтожается
       и сохраняет своё значение
    */
    static int a = 0;
    a++;
    cout << a << endl;
}

int main()
{
    // 1
    SomeFunc();

    // 2
    SomeFunc();

    // 3
    SomeFunc();
    return 0;

}
```

В нашем примере переменная a статическая, так как она объявлена с помощью ключевого слова static. Она будет создана при первом обращении к функции SomeFunc. По завершении работы функции она не будет уничтожена и сохранит своё значение. При новом вызове переменная

не будет пересоздаваться. При обращении к ней мы будем использовать значение, полученное при прошлом вызове. Именно поэтому, мы увидим на экране последовательность: 1 2 3.

Сегодня мы познакомились впервые с ключевым словом static в дальнейшем вы узнаете о других вариантах его использования.

6. Домашнее задание

1. Написать функцию, которая получает в качестве аргументов целое положительное число и систему счисления, в которую это число должно переводится (системы счисления от 2 до 36). Например, при переводе числа 27 в систему счисления 16 должно получиться 1B; 13 в 5-ю — 23; 35 в 18-ю — 1H.
2. Игра «кубики». Условие: имеется два игральных кубика со значениями от 1 до 6. Игра происходит с компьютером, кубики бросаются поочередно. Побеждает тот, у кого сумма выпавших очков по итогам пяти бросков больше. Предусмотрите возможность получения первого хода человеком или компьютером. Кубики отображаются с помощью символов. В конце игры необходимо выводить среднюю сумму по броскам для обоих участников.
3. Написать функцию, выводящую на экран прямоугольник с высотой N и шириной K.
4. Написать функцию, вычисляющую факториал переданного ей числа
5. Написать функцию, которая проверяет, является ли переданное ей число простым? Число называется простым, если оно делится без остатка только на себя и на единицу.
6. Написать функцию, определяющую минимум и максимум (значение и номер) элементов передаваемого ей массива
7. Написать функцию, меняющую порядок следования элементов передаваемого ей массива на противоположный.