Лекция 3. Массивы, указатели, функции

План лекции:

- Одномерные массивы.
- Многомерные массивы.
- Указатели. Связь массивов и указателей.
- Динамические переменные и массивы.

МАССИВЫ

Массив — это последовательность объектов одного и того же типа, которые занимают смежную область памяти.

Массивы данных могут быть одномерными (векторами), двухмерными (матрицами) или многомерными. В языках C/C++ индексация массива начинается с нуля. Например, если размер массива определен величиной 10, то в нем можно хранить 10 элементов с индексацией 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Доступ к отдельному элементу массива осуществляется с помощью индекса. Индекс описывает позицию элемента внутри массива.

```
int b = a[0];
a[1] = b*2;
```

Все массивы занимают смежные ячейки памяти, т.е. элементы массива в памяти расположены последовательно друг за другом. Ячейка памяти с наименьшим адресом относится к первому элементу массива, а с наибольшим – к последнему.

Выход за границы массивов должен отслеживать только сам программист.

Массивы

Объявление массива имеет два формата:

спецификатор-типа описатель [константное - выражение];

спецификатор типа описатель [ ];

Массивы определяются так же, как и переменные:

```
int a[100];
char b[20];
float d[50];
```

В первой строке объявлен массив **a** из 100 элементов целого типа: a[0], a[1], ..., a[99] (индексация всегда начинается с нуля). Во второй строке элементы массива **b** имеют тип char, а в третьей - float.

Описатель - это идентификатор(имя) массива

Память под такие массивы выделяется в стеке.

Выделение памяти под массив

Стек — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек элемент будет первым в очереди на вывод из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.
При создании статического массива для указания его размера может использоваться только константа. Размер выделяемой памяти определяется на этапе компиляции и не может изменяться в процессе выполнения.
Если требуется, чтобы массив был слишком большим для выделения в стеке или его размер не мог быть известен во время компиляции, можно выделить его в куче. Выделение памяти в процессе выполнения возможно при работе с динамическими массивами. Но о них немного позже.

Выделение памяти под массив

Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. Из-за динамической природы кучи ЦП не принимает участия в контроле над ней; в языках без сборщика мусора

(C, C++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу кучи.

В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности — если вам нужно работать с динамическими или глобальными переменными, пользуйтесь кучей.

Рассмотрим пример: задано число n — количество элементов последовательности, а затем n целых чисел array — это члены последовательности. Необходимо вывести заданную последовательность чисел в обратном порядке
(в этом примере память под массив выделяется в стеке)

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

int main() {
    int n; // size of array
    int array[100];
    cout << "Size: ";
    cin >> n; // number of elements no more than 100

    cout << "Array: ";
    for (int i = 0; i < n; i++) { // filling the array

        cin >> array[i];
        }

    for (int i = n-1; i >= 0; i--){
        cout << array[i] << " ";
        }

    return 0;
}
```

Рассмотрим пример: задано число n — количество элементов последовательности, а затем n целых чисел array — это члены последовательности. Необходимо вывести заданную последовательность чисел в обратном порядке
(в этом примере память под массив выделяется в куче)

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

int main() {
    int n; // size of array

    cout << "Size: ";
    cin >> n;

    int array[n];
    cout << "Array: ";
    for (int i = 0; i < n; ++i){ // filling the array
        cin >> array[i];
    }

    for (int i = n-1; i >=0; i--){
        cout << array[i] << " ";
        }

    return 0;
}
```

ВЕКТОРЫ

Традиционные массивы в стиле С являются источником многих ошибок, но по-прежнему еще применяются, как это показано в предыдущем примере. В современных С++ версиях предпочтительно использовать std:: Vector вместо массивов в стиле С.
С помощью этой стандартной библиотеки элементы массива тоже хранятся в непрерывном блоке памяти, но обеспечивается гораздо большая безопасность типов вместе с итераторами, которые гарантированно указывают на допустимое расположение в последовательности.
Чтобы обратиться к отдельным элементам вектора, нужно указать номер в квадратных скобках после его имени. Внутри скобок допускается любое арифметическое выражение, а с отдельным элементом вектора можно обращаться как с обычной переменной. Если указать в скобках номер, который превышает размер вектора (или отрицательное число), то программа скомпилируется, но будет работать неправильно или вовсе сломается. Это связано с тем, что такое обращение попадает в область памяти, которая не относится к нашему вектору.

Векторы

Пример создания вектора

```cpp
#include <iostream>
#include <vector>

int main()
{
    // Вектор из 10 элементов типа int
    std::vector<int> v1(10);

    // Вектор из элементов типа float
    // С неопределенным размером
    std::vector<float> v2;

    // Вектор, состоящий из 10 элементов типа int
    // По умолчанию все элементы заполняются нулями
    std::vector<int> v3(10, 0);

    return 0;
}
```

(в этом примере память под массив выделяется в векторе)

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

int main() {
    int n; // size of array

    cout << "Size: ";
    cin >> n;

    vector <int> array(n);
    cout << "Array: ";
    for (int i = 0; i < n; ++i){ // filling the array
        cin >> array[i];
    }

    for (int i = n-1; i >=0; i--){
        cout << array[i] << " ";
        }

    return 0;
}
```

Рассмотрим еще одну похожую задачу: нужно считать последовательность и вывести в обратном порядке только положительные элементы. Задачу можно решить несколькими способами — например, считать всю последовательность и при выводе печатать только положительные элементы. Мы будем решать задачу другим способом, запоминая только положительные элементы на этапе считывания данных.

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

int main() {
    int n; // size of array

    cout << "Size: ";
    cin >> n;

    vector <int> array(n);
    cout << "Array: ";
    for (int i = 0; i < n; i++) { // filling the array
        int temp;
        cin >> temp;
        if (temp > 0) {
            array.push_back (temp);
        }
    }

    for (int i = array.size()-1; i >=0; i--){
        cout << array[i] << " ";
        }

    return 0;
}
```

Методы класса vector

В этой программе при создании вектора мы не указывали количество элементов в нём — он создался пустым.

Добавление нового элемента в конец вектора делается с помощью метода **push_back()**, при этом в скобках указано то, что и требуется добавить. Метод — это почти то же самое, что функция, применяемая к объекту. Сначала указывается имя объекта (в нашем случае это вектор array), затем ставится точка, пишется имя метода и в скобках указываются параметры. Аналогично push_back() используется и метод **size()**, который не принимает параметров и возвращает количество элементов в векторе.

Для доступа к элементам вектора можно использовать квадратные скобки [], также, как и для обычных массивов.

pop_back() — удалить последний элемент

clear() — удалить все элементы вектора

empty() — проверить вектор на пустоту

Еще пример: Составить и вывести на экран новый массив с номерами элементов исходного массива, которые равны заданному значению.

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

int main() {
    int size; // size of array

    cout << "Size: ";
    cin >> size;

    int array[size];
    cout << "Array: ";
    for (int i = 0; i < size; ++i){ // filling the array
        cin >> array[i];
    }

    int value;
    cout << "Value: ";
    cin >> value;

    vector<int> index; // vector indexes of value

    for (int i = 0; i < size; ++i){
        if (array[i] == value){
            index.push_back(i);
        }
    }

    cout << "Index(es) of value: ";
    for (auto x: index){
        cout << x << " ";
    }

    return 0;
}
```

12:01

Пример: поменять местами минимальный элемент массива с первым

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

int main() {
    int n;

    cin >> n;
    vector <int> a;
//считывание
for (int i = 0; i < n; i++) {
        int temp;
        cin >> temp;
        a.push_back(temp);
}
//обработка
int num_min = 0; //номер минимального элемента
for (int i = 0; i < n; i++) {
        if (a[i] < a[num_min]) {
                num_min = i;
                }
}
//обмен значений элементов a[0] и a[num_min]
int temp;
temp = a[0];
a[0] = a[num_min];
a[num_min] = temp;
//вывод
for (auto now : a) {
        cout << now << " ";
}
}
```

Пример: сортировка массива методом выбора

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

int main() {
  int n;

  cin >> n;
  vector <int> a;
//считывание
for (int i = 0; i < n; i++) {
    int temp;
    cin >> temp;
    a.push_back(temp);
}
//обработка
for (int k = 0; k < n; k++){ // просмотры массива
int num_min = k; //номер минимального элемента
for (int i = k; i < n; i++) {
    if (a[i] < a[num_min]) {
        num_min = i;
        }
}

//обмен значений элементов a[k]и a[num_min]
int temp;
temp = a[k];
a[k] = a[num_min];
a[num_min] = temp;
}
//вывод
for (auto now : a) {
    cout << now << " ";
}
}
```

Обсуждение

В этих решениях используется ещё более короткая запись при выводе всех значений вектора. Цикл for (auto now : a) будет поочередно подставлять в переменную now все значения из вектора a. Здесь auto – это тип переменной, «автоматический». Поскольку вектор состоит из целых чисел, то переменная now автоматически будет определена как целое число.

МНОГОМЕРНЫЕ МАССИВЫ

В языке C++ определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы.

Они формализуются списком константных-выражений следующих за идентификатором массива, причем каждое константное-выражение заключается в свои квадратные скобки, поэтому двумерный массив представляется как одномерный, элементами которого так же являются массивы.

Многомерные массивы

Пусть задан массив:

```
int array[4][4];
```

Тогда элементы массива array будут размещаться в памяти следующим образом:

	0	1	2	3
0	array[0][0]	array[0][1]	array[0][2]	array[0][3]
1	array[1][0]	array[1][1]	array[1][2]	array[1][3]
2	array[2][0]	array[2][1]	array[2][2]	array[2][3]
3	array[3][0]	array[3][1]	array[3][2]	array[3][3]

Многомерные массивы

- Имя массива - это константа, которая содержит адрес его первого элемента (в данном примере переменная a содержит адрес элемента array[0][0]).
- Предположим, что array = 1000. Тогда адрес элемента array[0][1] будет равен 1004 (элемент типа int занимает в памяти 4 байта), адрес следующего элемента array[0][2] - 1008 и т.д.
- Что же произойдет, если выбрать элемент, для которого не выделена память? К сожалению, компилятор не отслеживает данной ситуации. В результате возникнет ошибка и программа будет работать неправильно.

Инициализация массива

Язык C++ позволяет инициализировать массив при его определении. Для этого используется следующая форма:

```
тип имя_массива[...] ... [...] = {список значений};
```

Примеры:

```
int a[5] = {0, 1, 2, 3, 4};
char ch[3] = {'d', 'e', '9'};
int b[2][3] = {1, 2, 3, 4, 5, 6};
```

В последнем случае: b[0][0] = 1, b[0][1] = 2, b[0][2] = 3, b[1][0] = 4, b[1][1] = 5, b[1][2] = 6.

Многомерные массивы

В объявлениях многомерных массивов, имеющих список инициализаторов, константное выражение, задающее границы для первого измерения, может быть опущено. Например:

```
const int cMarkets = 4;
double TransportCosts [ ] [cMarkets] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09,   9.76, 22.55 }
};
```

В показанном выше объявлении определяется массив, состоящий из трех строк и четырех столбцов.

Массивы C++ размещаются в памяти по срокам. Построчный порядок означает, что быстрее всего изменяется последний индекс.

УКАЗАТЕЛИ

Указатели - это переменные, показывающие место, или адрес памяти, где расположены другие объекты (переменные, функции и др.).

- При объявлении переменной типа указатель, необходимо определить тип объекта данных, адрес которого будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек).

Формат объявления указателя

спецификатор типа [модификатор] * описатель

Адреса, указатели

Так как указатель содержит адрес некоторого объекта, то через него можно обращаться к этому объекту. Унарная операция **&** дает адрес объекта, поэтому оператор

```
y = &x;
```

присваивает адрес переменной x переменной y. Операцию & нельзя применять к константам и выражениям; конструкции вида &(x+7) или &28 недопустимы.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     int x=2, *p;
6     p = &x;
7     cout << "x=" << x << endl;
8     cout << "adress of x =" << p <<endl;
9
10    return 0;
11  }
```

```
x=2
adress of x =0x7fff16e83698
```

Адреса, указатели

Унарная операция * воспринимает свой операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого, поэтому оператор

```
z = *y;
```

присваивает z значение переменной, записанной по адресу y.

```
y = &x;
z = *y;
```

Равнозначно **`z = x;`**

```
int *a, *b, *c;
char *d;
```

Адреса, указатели

Над указателями определено 5 основных операций:

- **Определение адреса указателя**: **&p**, где p – указатель (&p – адрес ячейки, в которой находится указатель).
- **Присваивание**. Указателю можно присвоить адрес переменной **p=&q**, где p – указатель, q – идентификатор переменной.
- **Определение значения**, на которое ссылается указатель: ***p** (операция косвенной адресации).Т.е. для того, чтобы получить **значение**, которое находится **по адресу**, на который ссылается указатель, **используется префикс *** . Данная операция называется **разыменованием указателя**.
- **Увеличение (уменьшение) указателя**.
  - Увеличение выполняется как с помощью операции сложения (+), так и с помощью операции инкремента (++).
  - Уменьшение – с помощью операции вычитания (−) либо декремента (−−).

Адреса, указатели

```
*y = 7;
*x *=5;
(*z)++;
```

В последнем случае круглые скобки необходимы, так как операции с одинаковым приоритетом выполняются справа налево. В результате если, например, *z = 5, то (*z)++ приведет к тому, что *z = 6, а *z++ всего лишь изменит сам адрес **z** (операция ++ выполняется над адресом **z**, а не над значением ***z** по этому адресу).

Адреса, указатели

- Указатели можно использовать как операнды в арифметических операциях.
- Если **y** - указатель, то унарная операция **y++** увеличивает его значение; теперь оно является адресом следующего элемента.
- Указатели и целые числа можно складывать. Конструкция **y + n** (y - указатель, n - целое число) задает адрес n-го объекта, на который указывает y. Это справедливо для любых объектов (int, char, float и др.); транслятор будет масштабировать приращение адреса в соответствии с типом, указанным в определении объекта.

Адреса, указатели

Любой адрес можно проверить на равенство (==) или неравенство (!=) со специальным значением NULL, которое позволяет определить ничего не адресующий указатель.

Разность двух указателей. Пусть p1 и p2 – указатели одного и того же типа. Можно определить разность p1 и p2, чтобы найти, на каком расстоянии друг от друга находятся элементы массива.

УКАЗАТЕЛИ И МАССИВЫ

В языке C++ существует сильная взаимосвязь между указателями и массивами. Любое действие, которое достигается индексированием массива, можно выполнить и с помощью указателей, причем последний вариант будет работать быстрее

Определение

```
int a[5];
```

задает массив из пяти элементов a[0], a[1], a[2], a[3], a[4].

Указатели и массивы

Если объект ***y** определен как

```
int *y;
```

то оператор **y = &a[0]**; присваивает переменной **y** адрес элемента **a[0]**. Если переменная **y** указывает на текущий элемент массива **a**, то **y+1** указывает на следующий элемент, причем здесь выполняется соответствующее масштабирование для приращения адреса с учетом длины объекта (для типа int - 4 байта, long - 8 байт, double - 8 байт и т.д.).

Указатели и массивы

Так как само имя массива есть адрес его нулевого элемента, то оператор **y = &a[0]**; можно записать и в другом виде: **y = a**. Тогда элемент **a[1]** можно представить как ***(a+1)**. С другой стороны, если **y** - указатель на массив **a**, то следующие две записи: **a[i]** и ***(y+i)** - эквивалентны.

Между именем массива и соответствующим указателем есть одно важное различие. ***Указатель - это переменная*** и **y = a**; или **y++**; - допустимые операции. ***Имя же массива - константа***, поэтому конструкции вида **a = y; a++;** использовать нельзя, так как значение константы постоянно и не может быть изменено.

Массивы указателей

В языке допускаются массивы указателей, которые определяются, например, следующим образом:

char *m[5];

Здесь m[5] - массив, содержащий адреса элементов типа char

Двумерный массив можно инициализировать, например, так:

```
int b[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Массивы

Пример, поиск суммы элементов одномерного массива (на С)

```c
#include <stdio.h>
#include <conio.h>
#include <locale.h>

int main (void)
{
        int i, size, S=0;
        int A[] = {3, 5, 2, 8, 12, 0, 7, -3, -21};
        setlocale(LC_ALL,"russian");

        size = sizeof(A)/sizeof(A[0]);
        printf("Размер массива A=%d\n", size);

        for (i = 0; i < size; ++i) S += A[i];
        printf("Сумма элементов: %d\n", S);

        printf("\n\n Нажмите любую клавишу: ");
        _getch();
        return 0;
}
```

Пример использования статических переменных

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

int main()
{
    int a; // Объявление статической переменной
    int b = 5; // Инициализация статической переменной b

    a = 10;
    b = a + b;
    cout << "b is " << b << endl;
    return 0;
}
```

Пример использования динамических переменных

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

int main()
{
    int *a = new int; // Объявление указателя для переменной типа int
    int *b = new int(5); // Инициализация указателя

    *a = 10;
    *b = *a + *b;

    cout << "b is " << *b << endl;

    delete b;
    delete a;

    return 0;
}
```

Динамические переменные

В первом примере мы объявляем/инициализируем статические переменные **a** и **b**, после чего выполняем различные операции напрямую с ними.

Во втором примере мы оперируем динамическими переменными посредством указателей. **Выделение памяти** осуществляется с помощью оператора **new** и имеет вид:

тип_данных *имя_указателя = new тип_данных;

например **int *a = new int;**

После удачного выполнения такой операции, в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа **int**.

Инициализация значения, находящегося по адресу указателя выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением:

тип данных *имя_указателя = new тип_данных (значение);

В нашем примере это

int *b = new int(5);

Для того, чтобы получить **адрес** в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом **&**. перед ним.

Динамические переменные

Например, чтобы вывести на экран адрес ячейки памяти, на который ссылается указатель **b** во втором примере, мы пишем

cout << "Address of b is " << &b << endl;

Для того, чтобы получить **значение**, которое находится **по адресу**, на который ссылается указатель, **используется префикс ***. Напомним, что данная операция называется **разыменованием указателя**.

Динамические переменные

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку, например, как во втором примере:

```
*b = *a + *b;.
```

Когда мы оперируем **данными**, то используем знак *****

Когда мы оперируем **адресами**, то используем знак **&**

Создание динамического массива

Синтаксис выделения памяти для массива имеет вид
указатель = new тип [размер];
В качестве размера массива может выступать любое целое положительное значение.

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

int main()
{
    int num; // размер массива
    cout << "Enter integer value: ";
    cin >> num; // получение от пользователя размера массива

    int *p_darr = new int[num]; // Выделение памяти для массива
    for (int i = 0; i < num; i++) {
        // Заполнение массива и вывод значений его элементов
        p_darr[i] = i;
        cout << "Value of " << i << " element is " << p_darr[i] << endl;
    }
    delete [] p_darr; // очистка памяти
    return 0;
}
```

Указатель на указатель

Возможно объявление переменной, которая содержит адрес другой переменной, которая, в свою очередь, также является указателем. Такая переменная может быть необходима, если в функции нужно изменить адрес какого-либо объекта. Однако наличие более двух звёздочек в объявлении переменной говорит, скорее всего, о плохом проектировании.

```
int **ppi;                    // Объявляем указатель на указатель на целое

void f(int **ppi)
 { int *pi = *ppi;            // Указателю на целое присваивается значение, хранящееся по адресу,
   ...                        // содержащемуся в указателе на указатель на целое
 }
```

ССЫЛКИ

Ссылка — это объект, указывающий на определенные данные, но не хранящий их. **Ссылка** (reference) не является указателем, а просто является другим именем для объекта. Для определения ссылки применяется знак амперсанда &:

```
1 int number = 5;
2 int &refNumber = number;
```

В данном случае определена ссылка refNumber, которая ссылается на объект number. При этом в определении ссылки используется тот же тип, который представляет объект, на который ссылка ссылается, то есть в данном случае int.

В языках программирования ссылка может быть реализована как переменная, содержащая адрес ячейки памяти.

После установления ссылки мы можем через нее манипулировать самим объектом, на который она ссылается. Изменения по ссылке неизбежно скажутся и на том объекте, на который ссылается ссылка.

```
1  #include <iostream>
2
3  int main()
4  {
5      int number = 5;
6      int &refNumber = number;
7      std::cout << refNumber << std::endl; // 5
8      refNumber = 20;
9      std::cout << number << std::endl;   // 20
10
11     return 0;
12 }
```

```
int  a = 10;
int &r = a;                          // Объявляем и инициализируем ссылку
r++;                                 // Значение переменной a становится 11

void f(double &a) { a += 3.14; }
double d = 0;
f(d);                                // d = 3.14

int v[20];
int& f(int i) {  return v[i]; }
f(3) = 7;                            // Элементу массива v[3] присваивается 7
```

На первый взгляд, ссылка является удобной заменой указателю, но она затрудняет понимание программы из-за несовпадения синтаксиса и семантики ссылки. Однако ссылки могут быть полезны для того, чтобы не передавать по значению (и не копировать) параметр функции, который имеет большой размер. В том случае, если мы не собираемся менять этот параметр внутри функции, можно объявить ссылку с модификатором const. В этом случае мы будем гарантированы, что параметр не изменится, вместо большого объекта будет передаваться его адрес, а для пользователя всё будет выглядеть как передача параметра по значению.

Функции

- Подпрограммой называется именованная логически законченная группа операторов языка, которую можно вызвать для выполнения по имени любое количество раз из различных мест программы.
- В языке C++ все подпрограммы являются **функциями**. Функции — это блоки кода, выполняющие определенные операции. Если требуется, функция может определять входные параметры, позволяющие вызывающим объектам передавать ей аргументы. При необходимости функция также может возвращать значение как выходное.
- Функция может быть вызвана из любого количества мест в программе. Значения, которые передаются функции, *являются ее аргументами,* чьи типы должны быть совместимы с типами формальных параметров параметров в определении функции.

Функции

- **Функцию** можно рассматривать как операцию, определенную пользователем. В общем случае она задается своим именем.
- Операнды функции, или *формальные параметры*, задаются в *списке параметров*, через запятую. Такой список заключается в круглые скобки.
- Результатом функции может быть значение, которое называют *возвращаемым*. Об отсутствии возвращаемого значения сообщают ключевым словом **void**.
- Переменные, объявленные в теле функции, называются локальными. Они исчезают из области видимости при выходе из функции, поэтому функция никогда не должна возвращать ссылку на локальную переменную.

Минимальное *объявление* функций состоит из типа возврата, имени функции и списка параметров (которые могут быть пустыми), а также дополнительных ключевых слов, которые предоставляют дополнительные инструкции компилятору.

```
int sum(int a, int b);
```

Определение функции состоит из декларации (объявления) , плюс *тело*, которое заключает весь код между фигурными скобками:

```
int sum(int a, int b)
{
    return a + b;
}
```

Функции

Прототипы функций

Особенностью стандарта ANSI является то, что для генерации правильного машинного кода функции до её первого вызова необходимо сообщить тип возвращаемого результата, а также количество и типы формальных параметров. Оператор объявления типа функции (прототип) имеет следующую общую форму

type_specifier function_name (type _argument);

где **type_specifier** - это спецификатор типа возвращаемого значения, **function_name** - имя функции, **type_argument** – список аргументов, который состоит из перечня типов, разделенных запятыми.

Если тип возвращаемого функцией значения не указан, то по умолчанию считается, что функция возвращает целое.

Функции

Аргументы в подпрограммы можно передавать одним из двух способов.

- ***Первый способ*** называется *вызовом-значением*. Этот способ копирует значение аргумента в формальный параметр подпрограммы. Поэтому изменения, которые вы делаете в параметрах подпрограммы, не влияют на переменные, которые вы используете при ее вызове.
- ***Второй способ***, которым вы можете передать аргументы в подпрограмму, называется *вызовом-ссылкой.* В этом случае в параметр копируется адрес аргумента. Внутри подпрограммы этот адрес используется для доступа к фактическому параметру, использованному в этом вызове. Это означает, что изменения, которые вы делаете в параметре, будут влиять на переменную, которая используется в вызове программы.

Пример: функция, вычисляющая наибольший общий делитель

```
1   #include <iostream>
2   using namespace std;
3
4   int gcd (int a, int b) {
5     while ( b != 0) {
6       int c = a % b;
7       a = b;
8       b = c;
9     }
10    return a;
11  }
12
13  int main() {
14    int m, n;
15    cin >> m >> n;
16    cout << gcd (m, n);
17    return 0;
18
19  }
```

Пример: Сокращение дроби (с помощью функции поиска наибольшего общего делителя)

Для сокращения дроби нужно найти наибольший общий делитель числителя и знаменателя, а затем разделить их на него

```
1   #include <iostream>
2   using namespace std;
3
4   int gcd (int a, int b) {
5     while ( b != 0) {
6       int c = a % b;
7       a = b;
8       b = c;
9     }
10    return a;
11  }
12  void reduce (int &a, int &b) {
13    int c = gcd (a, b);
14    a /= c;
15    b /= c;
16  }
17  int main() {
18    int m, n;
19    cin >> m >> n;
20    reduce (m, n);
21    cout << m << " " << n;
22    return 0;
23  }
```

Функции

- Рассмотрим функцию ***swap()***, которая изменяет значения двух своих целочисленных аргументов:

```
swap(int *x, int *y)
{
    int temp;
    temp = *x; /* сохранить значение из адреса x */
    *x = *y;   /* поместить y в x */
    *y = temp; /* поместить x в y */
}
```

Функции

Правильный способ вызова функции ***swap()***.

```
main()
{
    int x, y;
    x = 10;
    y = 20;
    swap(&x, &y);
}
```

Массивы и функции

- В том случае, когда в качестве аргумента функции используется массив, то передается только адрес этого массива, не его полная копия.
- Когда вы вызываете функцию с именем массива, вы передаете в эту функцию указатель на первый элемент в этом массиве. Это означает, что объявление параметра должно быть типа, сходного с указателем.
- Есть три способа объявить параметр, который будет получать указатель массива.

Массивы и функции

Во-первых, вы можете объявить его как массив:

```
1   #include <iostream>
2   using namespace std;
3
4    void display ( int num [10])
5   {
6     int i;
7     for (i = 0; i < 10; i++)
8       cout << num [i];
9   }
10
11
12  int main() {
13    int i;
14    int t [10];
15    for (i = 0; i < 10; i++)
16      t[i] = i;
17      display (t);
18    return 0;
19  }
```

Массивы и функции

Второй способ объявить параметр массива как безразмерный массив:

```
void display ( int num [])
{
  int i;
  for (i = 0; i < 10; i++)
    cout << num [i];
}
```

Массивы и функции

Третим способом, которым вы можете объявить параметр массива, и наиболее предпочтительной формой в C++ программах, является указатель, как показано ниже:

```
display ( int *num) {
  int i;
  for (i = 0; i < 10; i++)
    cout << num [i];
}
```

Массивы и функции

Двумерные массивы как аргументы функции

Если в вызывающей программе матрица описана как 2-мерный массив, то обращение к элементам в приведенной функции *display()* осуществляется с помощью индексных выражений.

```
1  #include <iostream>
2  using namespace std;
3
4  void display ( int *num) // указатель на начало массива
5  {
6      int i, j;
7      cout << "индексированный указатель" << endl;
8      for ( i =0; i < 6; i++)
9          cout << num [i];
10     cout << "адресное выражение" << endl;
11     for (i = 0; i < 2; i++)
12         for (j = 0; j < 3; j++)
13             cout << *(num + i*3 + j);
14 }
```

Массивы и функции

Обратите внимание на вызов функции и передачу массива в качестве фактического параметра (один из трех способов):

```
15  int main() {
16    int i, j;
17    int t [2][3];
18    for (i = 0; i < 2; i++)
19        for (j = 0; j < 3; j++)
20        t[i][j] = i+j;
21        display (&t[0][0]);
22        // display ((int *)t);
23        //display (*t);
24    return 0;
25  }
```

```
// Магический квадрат
#include <iostream>
using namespace std;

int main()
{
		setlocale(LC_ALL, "Russian");
		int n;
		cout << "Размер квадрата - ";
		cin >> n;

		int **square = new int *[n];
		for (int i = 0; i < n; ++i)  square[i] = new int[n];

		int sqr = n * n;
		int i = 0, j = n / 2;

		for (int k = 1; k <= sqr; ++k) {
			square[i][j] = k;
			i--;
			j++;
			if (k % n == 0) {i += 2;  --j;}
			else {
				if (j == n) j -= n;
				else if (i < 0) i += n; }
	}
		cout << "\n\nМагический квадрат размерностью - " << n << endl;
		for (int i = 0; i < n; i++){
			for (int j = 0; j < n; j++)
				cout << square[i][j] << "\t";
			cout << endl; }

		for (i = 0; i < n; i++)
			delete[] square[i];
		delete[] square;
		system("pause");
   return 0;
}
```

Рекурсия

В языке программирования C++ функции могут вызывать сами себя.

Функция является **рекурсивной**, если оператор в теле этой функции вызывает самого себя.

Работая с функциями, нужно различать две сущности: последовательность команд, которые выполняются в функции, и локальные переменные конкретного экземпляра функции. При рекурсивном запуске последовательность команд у функций одинаковая, но локальные переменные у каждого экземпляра свои. Кроме этого каждый экземпляр помнит, куда он должен вернуться после завершения работы. Например, при рекурсивном запуске нужно возвращаться в вызвавший эту функцию другой экземпляр той же функции.

Рекурсия

(пример из курса на stepik)

Представим себе, что у нас есть миллиард человек (это будущие экземпляры функции), сидящих в ряд, и у каждого из них есть листочек для записи (это его локальная память). Нам нужно произносить числа и написать инструкцию для людей так, чтобы они в итоге сказали все числа из последовательности в обратном порядке. Пусть каждый из них будет записывать на своем листочке только одно число. Тогда инструкция для человека будет выглядеть так:

1) Запиши названное число

2) Если число не последнее – потереби следующего за тобой человека, пришла его очередь работать

3) Когда следующий за тобой человек сказал, что он закончил – назови записанное число

4) Скажи тому, кто тебя теребил (предыдущий человек), что ты закончил

Формализуем задачу. Пусть задается последовательность натуральных чисел, заканчивающаяся нулем. Необходимо развернуть ее с помощью рекурсии.

Развернуть последовательность, оканчивающуюся нулем

```
1   #include <iostream>
2   using namespace std;
3    void rec() {
4     int n;
5     cin >>n;
6     if (n != 0) {
7       rec();
8       cout << n;
9     }
10    return;
11  }
12  int main() {
13    rec ();
14    return 0;
15  }
```

Рекурсия

```
fact(int n) /* рекурсивная */
{
        int answer;
        if(n==1) return(1);
        answer=fact(n-1)*n;
        return(answer);
}
```

```
fact(int n) /* нерекурсивная */
{
        int t,answer;
        answer=1;
        for(t=1; t<=n; t++) answer=answer*(t);
        return(answer);
}
```

Значения параметров по умолчанию

- Значение параметра по умолчанию – это значение, которое разработчик считает подходящим в большинстве случаев употребления функции, хотя и не во всех. Оно освобождает программиста от необходимости уделять внимание каждой детали интерфейса функции. Значения по умолчанию для одного или нескольких параметров функции задаются с помощью того же синтаксиса, который употребляется при инициализации переменных.
- Функция, для которой задано значение параметра по умолчанию, может вызываться по-разному. Если аргумент опущен, используется значение по умолчанию, в противном случае – значение переданного аргумента.

Значения параметров по умолчанию

- Фактические аргументы сопоставляются с формальными параметрами позиционно (в порядке следования), и значения по умолчанию могут использоваться только для подстановки вместо отсутствующих последних аргументов.
- При разработке функции с параметрами по умолчанию придется позаботиться об их расположении. Те, параметры для которых значения по умолчанию вряд ли будут употребляться, необходимо поместить в начало списка.

Значения параметров по умолчанию

```
int func( int a, int b, int c = 0 );
```

- В приведенном примере, если третий параметр при вызове функции будет опущен, то ему будет присвоено значение 0.

Полный пример

```
int func( int a, int b, int c = 0 )
{
            printf("c = %d\n",c);
}
main()
{
            func(1,2,3);
            func(1,2);
}
```

Многоточие

- Иногда нельзя перечислить типы и количество всех возможных аргументов функции. В этих случаях список параметров представляется многоточием (...), которое отключает механизм проверки типов. Наличие многоточия говорит компилятору, что у функции может быть произвольное количество аргументов неизвестных заранее типов.
- Многоточие употребляется в двух форматах:

```
void func( parm_list, ... );
void func( ... );
```

Многоточие

Примером вынужденного использования многоточия служит функция **printf()** стандартной библиотеки C. Ее первый параметр является C-строкой:

```
int printf( const char* ... );
```

```
printf( "hello, world\n" );
```

```
printf( "hello, %s\n", userName );
```

Многоточие

Большинство функций с многоточием в объявлении получают информацию о типах и количестве фактических параметров по значению явно объявленного параметра. Следовательно, первый формат многоточия употребляется чаще.

Отметим, что следующие объявления неэквивалентны:

```
void f();
void f( ... );
```

В первом случае f() объявлена как функция без параметров, во втором – как имеющая ноль или более параметров.

Практические задания

- Напишите функцию min(a, b), вычисляющую минимум двух чисел. Затем напишите функцию min4(a, b, c, d), вычисляющую минимум 4 чисел с помощью функции min. Считайте четыре целых числа и выведите их минимум.
- Даны четыре действительных числа: x_1, y_1, x_2, y_2. Напишите функцию distance(x1, y1, x2, y2), вычисляющую расстояние между точкой (x_1. y_1) и (x_2, y_2). Считайте четыре действительных числа и выведите результат работы этой функции.
- Дано натуральное число n > 1. Выведите его наименьший делитель, отличный от 1. Решение оформите в виде функции MinDivisor(n). Количество операций в программе должно быть пропорционально корню из n.
  Указание. Если у числа n нет делителя, меньшего n , то число n — простое и ответом будет само число n.
- Напишите функцию fib(n), которая по данному целому положительному n возвращает n-е число Фибоначчи. В этой задаче нельзя использовать циклы - используйте рекурсию.

Интернет-ресурсы

Учебник C++ http://computersbooks.net/index.php?id1=4&category=language-programmer&author=podelskiy-vv&book=2003

курс по C++ https://stepik.org/course/363/promo

курс C++ с нуля https://code-live.ru/tag/cpp-manual/

курс C++ https://www.intuit.ru/studies/courses/17/17/info

курс C https://www.intuit.ru/studies/courses/43/43/info

курс основ программирования на C/C++ https://stepik.org/course/55918/promo

курс, позволяющий изучать все даже в мобильном и иметь компилятор в телефоне https://www.sololearn.com/