



# Lesson 3

Массивы. Двумерные массивы данных. Указатели и ссылки. Функции. Понятие Рекурсия.  
Примеры кода.



# Массивы.

**Массив** — это набор переменных одного типа. Массивы делятся на одномерные (для доступа к элементам используется один индекс) и многомерные (используется несколько индексов).

Общий синтаксис одномерного массива: `тип_данных имя_массива[размер_массива];`

```
int a[100];  
char m[12];
```

В результате в памяти компьютера выделяется **100 ячеек типа int** и **12 ячеек типа char**. Доступ к элементам массива осуществляется по индексу через квадратные скобки `[]`, а инициализация массива (задание значений по-умолчанию массиву данных) — через фигурные скобки `{ }`. Для примера:

```
a[2] = 5;  
a[17] = 8;
```

```
int b[5] = { 0, 3, 4, 3, 2 };
```

## Одномерный массив



0	1			
1	2	3	4	5

array[0] -> 1

array[1] -> 2

```
#include <iostream>
```

```
int main() {
```

```
    // Размер массива
```

```
    const int size = 5;
```

```
    // Определение и инициализация статического  
    одномерного массива
```

```
    int array[size] = {1, 2, 3, 4, 5};
```

```
    // Вывод элементов массива
```

```
    std::cout << "Элементы массива: ";
```


```
    for (int i = 0; i < size; ++i) {
```

```
        std::cout << array[i] << " ";
```

```
    }
```

```
    return 0;
```

```
}
```



```
#include <iostream>

int main() {
    // Размер массива
    const int size = 5;
    // Определение и инициализация статического
одномерного массива
    int массив[size];
    // Ввод значений от пользователя
    std::cout << "Введите " << size << " целых чисел
для заполнения массива:\n";
    for (int i = 0; i < size; ++i) {
        std::cout << "Элемент " << i + 1 << ": ";
        std::cin >> массив[i];
    }
    std::cout << "Элементы массива: ";
    for (int i = 0; i < size; ++i) {
        std::cout << массив[i] << " ";
    }
    return 0;
}
```



# Двумерные массивы.

**Двумерный массив** — это список одномерных массивов (*фактически это матрица, многомерный уже массив данных*). Для доступа к элементам двумерного массива необходимо указать два индекса.

```
int matrix[5][5];  
matrix[0][0] = 1;  
matrix[2][1] = 3;
```

Пример заполнения **двумерного массива** и вывода его в консоль:

```
for ( int i = 0; i < 5; i++ )  
    for ( int j = 0; j < 5; j++ )  
        std::cin >> matrix[i][j];
```

```
for ( int i = 0; i < 5; i++ ){  
    for ( int j = 0; j < 5; j++ ){  
        std::cout << matrix[i][j];  
    }  
    std::cout << "\n";  
}
```

# Введение в указатели/ссылки.

При выполнении любой программы, все необходимые для ее работы данные должны быть загружены в оперативную память компьютера. Для обращения к переменным, находящимся в памяти, используются специальные адреса, которые записываются в шестнадцатеричном виде, например 0x100 или 0x200.

Если переменных в памяти потребуется слишком большое количество, которое не сможет вместить в себя сама аппаратная часть, произойдет перегрузка системы или ее зависание.

Если мы объявляем переменные статично, так как мы делали в предыдущих уроках, они остаются в памяти до того момента, как программа завершит свою работу, а после чего уничтожаются.

Такой подход может быть приемлем в простых примерах и несложных программах, которые не требуют большого количества ресурсов. Если же наш проект является огромным программным комплексом с высоким функционалом, объявлять таким образом переменные, естественно, было бы довольно не умно.

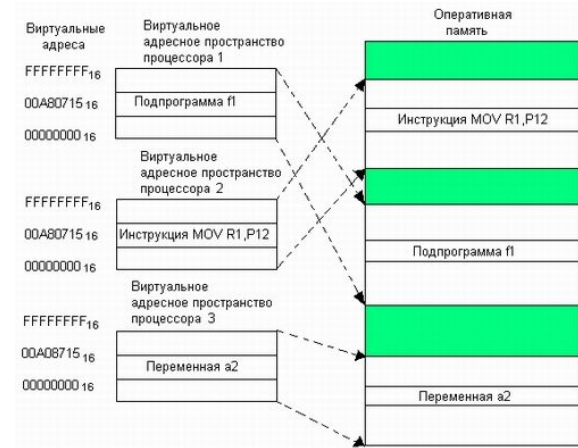


**Адресная шина в оперативной памяти (RAM)** - это канал, который используется для передачи адресов между процессором и памятью.

# RAM.

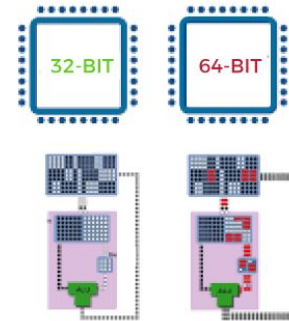
**Оперативная память** - это место в компьютере, где хранятся данные, с которыми работает программа в момент ее выполнения. Каждая программа имеет свою "область" в этой памяти, где хранятся ее данные. ОС управляет этой памятью, выделяя ее программам по мере необходимости. Если программе требуется больше памяти, чем доступно, ОС может использовать хитрости, чтобы все работало как нужно.

**Виртуальное адресное пространство** - это способ, с помощью которого каждая программа "думает", что у нее есть вся память компьютера для использования, хотя на самом деле память физически ограничена.



## Размер памяти для 32/64 битных систем.

На **32-битной системе** максимально доступное виртуальное адресное пространство составляет  $2^{32}$  байт, что равно 4 гигабайтам (или 4,294,967,296 байт). Эта память распределяется между всеми процессами и операционной системой. Однако фактический объем доступной физической памяти может быть меньше из-за ограничений настройки системы и резервирования памяти для других нужд, таких как драйверы и системные процессы.



32 bit vs 64 bit



# Указатели и ссылки.

**Указатель** — это переменная, содержащая адрес другой переменной в памяти. Для примера, если переменная **a** содержит адрес переменной **b**, это означает, что переменная **a** указывает на переменную **b**. В C++ поддерживаются **три типа** указателей:

- 1) управляемые указатели (вы не можете назначить адрес вне среды выполнения)
- 2) неуправляемые указатели (вы можете назначить любой адрес памяти)
- 3) указатели на неуправляемые функции

Синтаксис управляемого указателя ( keyword **ref**): **ref** **description**;

Синтаксис неуправляемого указателя (традиционные указатели C++): **тип\_данных\_переменной\*** **именование**;  
Чтобы получить адрес переменной в памяти, вы используете операцию **&**.

```
int* a;
```

\*Все операции, работающие с указателями, такие же, как и с обычными переменными.

*разыменование исп-ся для изменения значения.*

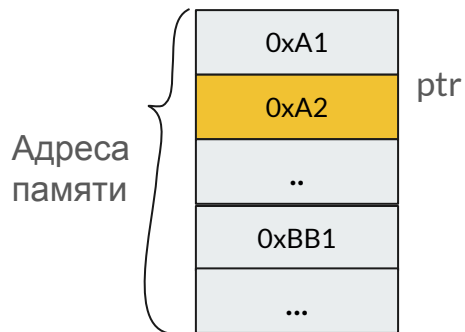
```
int x = 5;
int* a = &x;
*a = 10; // now x = 10
```



## Задачи

1. Просто создайте какую-то переменную (типа данных `float` или `int`) и выведите ее на экран.
2. Создать переменную (`float`) `x`, создать указатель `ptr` с адресом этой переменной, изменить значение по адресу и вывести `x`.
3. Создайте массив целых чисел (`int`) размером 5 и заполните его значениями. Затем создайте указатель на массив и выведите на экран значения массива через указатель.
4. Создайте две переменные типа `double`, затем создайте указатель, который будет указывать на переменную с максимальным значением. Выведите значение через указатель.

## неуправляемый указатель



Смена значения по указателю ptr:

```
*ptr = 5;
```

**Полный код:**

```
int x = 100; // исходное значение 100
int* ptr = &x; // передаем в ptr ссылку на переменную x
std::cout << ptr << std::endl; // ptr хранит адрес 0x7ff7bab476e8
*ptr = 5; // разыменование указателя с изменением значения по адресу
std::cout << x; // вывод нового значения x (5)
```



# Функции.

**Функция** — это часть программы, которая имеет следующие свойства или атрибуты:

- 1) является логически независимой частью программы;
- 2) имеет имя, на основании которого вызывается функция.
- 3) может содержать список параметров, которые передаются ей для обработки или использования. Если функция **не содержит** списка параметров, такая функция называется функцией без параметров;
- 4) может возвращать (*не обязательно*) некоторое значение. Если функция не возвращает никакого значения, то указывается ключевое слово **void** (*что с означает пустота*);
- 5) имеет собственный программный код, который находится между фигурными скобками **{ }** (*тело функции*). Программный код решает задачу, возложенную на эту функцию.

# Синтаксис Функции.

тип данных возвращаемого  
элемента (если *ничего не*  
возвращаем то ставим `void`)

имя функции

передаваемые параметры в функцию  
(если параметров нет то здесь  
пустые круглые скобки будут)

```
type function_name ( options .. )  
{  
    function body;  
    [return] (expression);  
}
```

оператор **return** служит для  
возврата какого либо  
значения согласно типу  
возвращаемого элемента

**тело функции** (важная часть  
функции), здесь собственно  
совершаются различные операции  
такие как проход по массиву  
используя цикл, операции  
присваивания, сложения между  
переменными и много др.



## Небольшой пример создания и вызова функции.

В этом примере, переменная `x` передается функции, а затем выполняются различные операции в теле функции, а затем переменная `val` возвращается (с помощью `return`).

```
int get_value(int x) {  
    int val = 0;  
    val = x * 2315;  
    return val;  
}
```

Вызов функции будет следующий `get_value(5);`



## Аргументы по-умолчанию и перегрузка функций.

**Аргумент по умолчанию (параметры по-умолчанию)** — это аргумент функции, который программист может не указывать при вызове функции. Аргумент по умолчанию добавляется компилятором автоматически. Проще говоря, мы просто присваиваем некоторое значение параметру функции (по умолчанию).

**Перегрузка функции** — это объявление функции с тем же именем несколько раз. Чтобы компилятор мог отличать «перегруженные» функции друг от друга, эти функции должны различаться списком входных параметров.

```
int Max(int a, int b) {  
    // ...  
}  
  
int Max(int a, int b, int c) {  
    // ...  
}
```



## Аргументы по-умолчанию. Пример.

```
#include <iostream>

// Пример функции с параметрами по умолчанию
int add(int a, int b = 0) {
    return a + b;
}

int main() {
    // Вызов функции с двумя аргументами
    int result1 = add(5, 3);
    // Вызов функции с одним аргументом,
    // второй аргумент использует значение по умолчанию
    int result2 = add(5);

    std::cout << result1 << std::endl; // 8
    std::cout << result2 << std::endl; // 5
    return 0;
}
```





## Перегрузка функций. Пример.

```
#include <iostream>

// Перегрузка функции для сложения разных типов данных
int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

int main() {
    // int add(int, int);
    int result1 = add(5, 3);
    // double add(double, double);
    double result2 = add(3.5, 2.1);
    return 0;
}
```



## Понятие рекурсии.

**Рекурсия** — это способ определения функции, при котором результат возврата из функции для данного значения аргумента основан на результате возврата из той же функции для предыдущего (*меньшего или большего*) значения аргумента. Если функция (метод) вызывает саму себя, это называется рекурсивным вызовом функции.

**Рекурсия** - это когда функция вызывает саму себя, решая похожие, но более простые задачи.

Рекурсия может быть сравнима с циклом, но она использует вызов функции вместо итераций. Давайте представим задачу вычисления факториала числа ( **$n!$** ):

$$2! = 1 * 2 = 2$$

$$3! = 1 * 2 * 3 = 6$$

# Понятие рекурсии. Пример (n!).

с рекурсией

```
#include <iostream>

int factorial(int n) {
    // n = 0 значит возвращаем сразу 1
    if (n == 0)
        return 1;
    else
        // n! = n * (n-1)!
        return n * factorial(n - 1);
}

int main() {
    int result1 = factorial(2);
    int result2 = factorial(5);
    std::cout << "2! = " << result1 << std::endl; // 2
    std::cout << "5! = " << result2 << std::endl; // 120
    return 0;
}
```

без рекурсии

```
#include <iostream>

int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
    int result1 = factorial(2);
    int result2 = factorial(5);
    std::cout << "2! = " << result1 << std::endl; // 2
    std::cout << "5! = " << result2 << std::endl; // 120
    return 0;
}
```



Thank you for your attention!