

Лекция II

27 сентября 2017

2. Управление ходом выполнения программы

Цикл

Под **циклом** в программировании понимается обособленный набор повторяющихся N раз инструкций. Причём, $N \geq 0$

Дополнительные определения

- **Тело цикла** - набор повторяющихся инструкций
- **Итерация** (или проход цикла) - однократное выполнение всех инструкций из тела цикла

2. Управление ходом выполнения программы

Циклы **while** и **do ... while**

```
while (<логическое выражение>) {  
    [инструкции];  
}
```

```
do {  
    [инструкции];  
} while (<логическое выражение>);
```

2. Управление ходом выполнения программы

Цикл **while**

```
1 int counter = 10;  
2  
3 while ( counter > 0 ) {  
4     printf("%d\n", counter * counter);  
5     counter--;  
6 }
```

2. Управление ходом выполнения программы

Цикл **do ... while**

```
1 int counter = 0;  
2  
3 do {  
4     printf("%d\n", counter * counter);  
5     counter++;  
6 } while ( counter < 10 );
```

2. Управление ходом выполнения программы

Бесконечный цикл на примере **while**

```
1 while ( 1 ) {  
2     printf( "*" );  
3 }
```

2. Управление ходом выполнения программы

Цикл **for**

```
for (    [инициализация];  
      [условие выхода];  
      [итеративное изменение]  
    ) {  
    [инструкции];  
}
```

- ❶ **Инициализация:** инструкции, которые выполняются до начала работы цикла. Как правило, задание различных счётчиков, используемых внутри цикла.
- ❷ **Условие выхода:** логическое выражение, проверяемой **до первой** и после каждой следующей итерации.
- ❸ **Итеративное изменение:** задание выражений (изменение счётчика цикла, как пример), которые выполняются **после каждой итерации**.

2. Управление ходом выполнения программы

Цикл **for**

```
1 #include <math.h>
2
3 // Бесконечный цикл
4 for (;;) {
5     // что-то полезное
6 }
7
8 // Вывод значений квадратного корня для ←
   чисел от 0 до 9
9 for (int i = 0; i < 10; i++) {
10     printf("%lf\n", sqrt(i));
11 }
```


2. Управление ходом выполнения программы

Управление циклами **continue** и **break**

```
1 for (int counter = 0; ; ++counter) {  
2     if ( (counter % 2) == 0 ) {  
3         printf("%d\n", counter);  
4         continue;  
5     }  
6  
7     if ( (counter > 15) ) {  
8         break;  
9     }  
10  
11     printf("Итерация закончена\n");  
12 }
```

Управляющие конструкции закончились: кроме трёх циклов, условного перехода да конструкции **switch** больше нечем влиять на логику программы (99% правда)

Переходим к рассмотрению функций в C

3. Функции. Определение

Функция - обособленный набор инструкций, пригодный для повторного использования, который может принимать произвольное количество значений и возвращать значение.

Определение функции в C

```
[...] <тип_возвращаемого_значения>  
      <имя_функции> ( <список_аргументов> )  
{  
    [инструкции;]  
    return <значение>;  
}
```

, где **список_аргументов** - это набор пар *тип данных* и *символьное обозначение* каждого параметра, перечисляемых через запятую. Троекотия - специальные указания компилятору, возможно рассмотрятся на практике.

3. Функции. Определение

- Каждая функция должна быть уникальна
- Уникальность функции в языках программирования определяется её **сигнатурой** - набором отличительных признаков
- В С сигнатура функции определяется **её именем** (строго говоря - идентификатором)
- Определение функции внутри другой функции стандартом языка запрещено (но некоторые компиляторы позволяют это делать)

3. Функции. Определение

$$n! = 1 * 2 * 3 * 4... * n$$

```
1 long long factorial(unsigned n)
2 {
3     long long result = 1;
4
5     for (unsigned i = 2; i <= n; ++i) {
6         result *= i;
7     }
8
9     return result;
10 }
11
12 ...
13 long long result = factorial(8);
14 printf("Факториал 8 равен: %lld", result);
```

3. Функции. Множественный return

Число операторов возврата **return** - неограничено.

```
1 long long factorial(unsigned n)
2 {
3     if ( n < 2) {
4         return 1;
5     }
6
7     long long result = 1;
8
9     for (unsigned i = 2; i <= n; ++i) {
10         result *= i;
11     }
12
13     return result;
14 }
```

3. Функции. Определение

number^n , n – целое

```
1 double degree_nth(double num, int n)
2 {
3     double result = 1;
4     int is_negative = n < 0;
5     unsigned degree = is_negative ? -n : n;
6
7     while ( degree > 0 ) {
8         result *= num;
9         --degree;
10    }
11
12    return is_negative ?
13           (1 / result) : result;
14 }
```

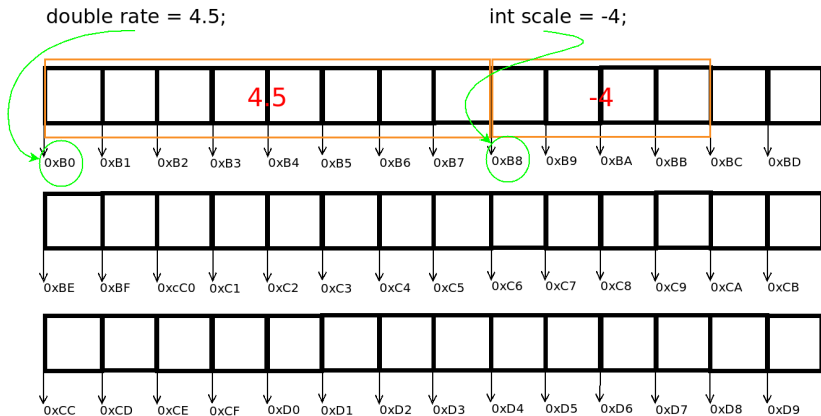
3. Функции. Передача аргументов по значению

```
1 double degree_nth(double num, int n)
2 { ... }
3
4 double rate = 4.5;
5 int scale = -4;
6
7 double rate_in_scale = degree_nth( rate, ←
    scale );
```

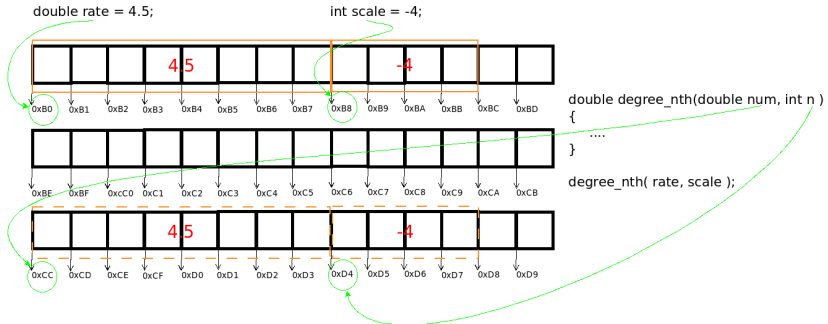

Адрес переменной

Каждая переменная любого типа связана с блоком в памяти, который состоит из какого-либо количества байт. Каждый байт в модели памяти языка C пронумерован - ему присвоен порядковый номер: 0, 1, 2, 3, Номер первого байта блока называется **адресом переменной**. Адреса в литературе приводят в *шестнадцатиричном* виде.

3. Функции. Передача аргументов по значению



3. Функции. Передача аргументов по значению



3. Функции. Специальный тип void

Функция может не возвращать никакого значения. Для этого в качестве возвращаемого типа используется специальный тип - **void**. Вызов оператора **return** не обязателен.

```
1 void print_pair(double first, double second)
2 {
3     printf("<< %lf, %lf >>", first, second);
4
5     // В тоже время, нельзя создать переменную ←
6     // void v_var; // Ошибка компиляции!
7 }
```

3. Функции. Специальный тип void

Однако, оператор **return** может присутствовать для немедленного выхода из подобной функции (тип возвращаемого значения - **void**).

```
1 void declare_party(size_t alcohol_litres)
2 {
3     if ( alcohol_litres == 0 ) {
4         return;
5     }
6
7     if ( alcohol_litres < 10 ) {
8         printf("Party good: :)\n");
9         return;
10    }
11
12    printf("Party very good: :D!\n");
13 }
```

3. Функции. Объявление и определение

Аналогично переменным, для функций существуют понятия **объявления** и **определения**. Определение - это все примеры выше, а под **объявлением** понимается указание *типа возвращаемого значения, названия функции и перечисление типов всех аргументов*. Причём, давать имена аргументам - не обязательно

```
1 #include <stdio.h>
2
3 double my_random(unsigned); // Объявление!
4
5 int main()
6 {
7     printf("Случайное число: %lf\n", my_random(567));
8 }
9
10 double my_random(unsigned seed) // Определение!
11 { return 4.5688 / seed; }
```

Основные сведения о функциях законились.

На очереди - массивы

Область видимости идентификатора

Место в файле с исходным кодом, где идентификатор доступен для операций (использование значения в случае переменной, вызов - в случае функции). В языке C различают две области - **глобальная** и **локальная**.

- Все функции имеют **глобальную** область видимости
- Переменная имеет **локальную** область видимости, если её объявление или определение находится внутри пары фигурных скобок ({})
- Иначе переменная получает **глобальную** область видимости
- Локальная переменная видна во всех вложенных областях

Прежде, чем идти дальше

Область видимости идентификатора

```
1 // Переменная с глобальной областью видимости
2 const double PI = 3.14159265358;
3
4 void do_something(double x)
5 {
6     // Локальная область видимости
7     double rate = 0.5;
8
9     if (x > 10.0) {
10         // Ещё одна. PI и rate тут доступны
11         double num = PI * x * rate;
12         printf("Результат: %lf", num);
13     }
14
15     // Доступа к num уже нет:
16     // num += 3.05;
17     rate *= 9.5;
18 }
```

Время жизни переменной

- Переменные с локальной областью видимости автоматически удаляются по достижении конца области видимости
- Глобальные переменные существуют до тех пор, пока выполняется программа

Массив - структура данных, содержащая набор проиндексированных элементов. В языке С массивы являются типизированными, то есть могут содержать элементы только одного типа.

```
<тип> <имя_массива>[<размер>];
```

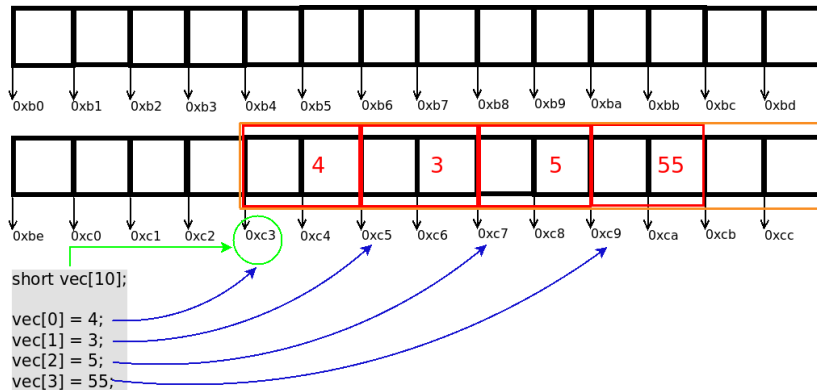
В С все элементы одного массива располагаются в памяти последовательно.

Статический массив

Индексация элементов в массиве **начинается с нуля**.

```
1 short vec[10];  
2  
3 // Задаём значение первого элемента  
4 vec[0] = 4;  
5 vec[3] = 55;  
6  
7 printf("Первый элемент равен %d", vec[0]);
```

Точка зрения памяти



Инициализация массива - задание начальных значений каждому элементу

```
1 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
2
3 for (unsigned i = 0; i < 8; ++i) {
4     printf("%u ", vec[i]);
5 }
6 printf("\n");
7
8 // Незаданные элементы будут равны 0.0
9 double real_arr[5] = { 3.4, 5.5, 77.11 };
10
11 // При явной инициализации размер массива ←
    можно пропускать
12 int another_vec[] = { 1, 2, 3, 4 };
```

Многомерные массивы

```
1 int matrix1[10][10];
2
3 for (int i = 0; i < 10; ++i) {
4     for (int j = 0; j < 10; ++j) {
5         matrix[i][j] = i + j;
6     }
7 }
8
9 // Инициализация
10 int matrix2[3][3] = { {1, 2, 3}, {4, 5, 6}, ↵
    {7, 8, 9} };
```

Многомерные массивы: в числе размерностей никто не ограничен. Гарантируется работа до 31 уровня вложенности.

```
1 int monstr[3][4][5][3][4][5];  
2 monstr[0][0][0][0][0][0] = 5;
```


Передача массивов в функции.

```
1 void print_array(int arr[], size_t count)
2 {
3     std::cout << "\nПереданный массив:\n";
4     for (size_t i = 0; i < count; ++i) {
5         printf("%d ", arr[i]);
6     }
7     printf("\n");
8 }
9
10 ...
11
12 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
13 print_array(vec, 8);
```

Никакой проверки размерности массива не происходит.

```
1 void print_array(int arr[55], size_t count)
2 {
3     printf("\nПереданный массив:\n");
4     for (size_t i = 0; i < count; ++i) {
5         printf("%d ", arr[i]);
6     }
7     printf("\n");
8 }
9
10 ...
11
12 int vec[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
13 print_array(vec, 8);
```

Массивы в C

Многомерные массивы: нужно указать все размерности, кроме первой.

```
1 const size_t COLS = 8;
2
3 void print_2D_array(int matr[][COLS], size_t ←
    rows_count)
4 {
5     printf("\nПереданный двумерный массив:\n") ←
        ;
6     for (size_t i = 0; i < rows_count; ++i) {
7         for (size_t j = 0; j < COLS; ++j) {
8             printf("%d ", matr[i][j]);
9         }
10        printf("\n");
11    }
12    printf("\n");
13 }
```

В современном стандарте С (**C11**) определены **массивы переменной длины** (VLA - variable length array)

```
1 size_t my_size = 15;
2
3 int arr[my_size];
4
5 for (size_t i = 0; i < my_size; ++i) {
6     arr[i] = 15 - i;
7 }
```

Другой пример на массивы переменной длины

```
1 size_t rows, cols;
2
3 printf("Введите число строки и"
4        " столцов матрицы: ");
5 scanf("%lu %lu", &rows, &cols);
6
7 // Определяем двумерный массив
8 // заданных размеров
9 double real_matrix[rows][cols];
10
11 // Используем по своему усмотрению
12 for (size_t i = 0; i < rows; ++i) {
13     for (size_t j = 0; j < cols; ++j) {
14         real_matri[i][j] = 35.678 / (1 + i + j);
15     }
16 }
```

Ограничения на **массивы переменной длины**

- Массивы переменной длины не могут быть глобальными (должны использоваться только внутри функций)
- Массив переменной длины невозможно проинициализировать начальными значениями при объявлении