Семинар #1: Основы C. Ввод/вывод. Операторы. Циклы. Массивы.

Почему C?

- **Язык C это возможно самый влиятельный язык в истории программирования**
  Он появился в 1972 году и повлиял на многие языки, который появились после него, такие как C++, C#, Java и многие другие. После изучения C, вам будет проще изучать другие языки.
- **Язык C популярный**
  Несмотря на "старость язык C сейчас является одним из самых популярных языков программирования. Многие проекты, в том числе новые, до сих пор пишутся на языке C.
- **Язык C компактный**
  Он содержит в себе относительно немного языковых конструкций и большую часть синтаксиса языка вполне можно изучить меньше чем за один семестр.
- **Язык C быстрый**
  Язык позволяет писать код, который будет работать так быстро, насколько это возможно. На других языках сложно написать код, который будет так же быстр, как код на языках C и C++.
- **Язык C является основой для языка C++**
  C++ был создан на основе языка, поэтому первый шаг для полноценного изучения языка C++ это изучение языка C. Несмотря на это, это разные языки. В язык C++ было добавлено так много новых возможностей по сравнению с C, что код на C++ обычно выглядит совершенно по другому. C++ это гораздо более сложный язык, требующий для изучения хорошего знания C.

Функция `main`

Любая программа на языке C должна содержать функцию `main`. Эта функция является точкой входа в программу, с неё программа начинает своё выполнение. Тему *"Функции"* мы будем проходить через несколько занятий. На данный момент единственное, что вы должны знать, это то что весь код нужно писать внутри фигурных скобочек у функции `main`. Вот простейшая программа на языке C:

```
int main() {}
```

Она ничего не делает, так как внутри фигурных скобочек `{}` ничего нет.

Комментарии

Комментарии в языках программирования – это поясняющий текст в исходном коде, который игнорируется компилятором. В языке C однострочные комментарии начинаются с `//`, а многострочные начинаются с `/*` и заканчиваются на `*/`.

```
int main()
{
    //  Это однострочный комментарий

    /*
        Это
        Многострочный
        Комментарий
    */
}
```

Печать на экран. Функция `printf`.

Программа Hello World!

Давайте рассмотрим простейшую программу, которая хоть что-то делает. Данная программа выводит на экран (то есть в терминал) строку `Hello World!`. Чтобы это сделать мы используем функцию `printf` из библиотеки `stdio.h`. Эта функция принимает строку (выражение в кавычках `""` называется строкой) и печатает её на экран. Для подключения необходимой библиотеки `stdio.h` используем директиву `#include`.

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
}
```

Функция `printf`

`printf` это сокращение от *print formatted*, что переводится как форматированная печать. Печать форматированная, потому что с помощью этой функции можно печатать какие-либо объекты в разных форматах. Например, если мы в будущем захотим напечатать с помощью `printf` вещественное число, то можно будет указать количество знаков после запятой и другие параметры. Функция `printf` не является встроенной в язык C, она содержится в библиотеки `stdio.h`, чтобы функция `printf` работала корректно, нужно обязательно подключать эту библиотеку.

Особые символы

Для печати особых символов в функции `printf` используются экранирующие последовательности, которые начинаются с обратного слэша `"\"`.

`\n`	`перенос на новую строку (аналог нажатию Enter в текстовом редакторе)`
`\t`	`табуляция (аналог нажатию Tab в текстовом редакторе)`
`\\`	`один символ обратного слэша (\)`

Например, такая программа:

```
#include <stdio.h>
int main()
{
    printf("One\n\tTwo\n\t\tThree\n");
}
```

Напечатает на экран следующее:

```
One
    Two
        Three
```

Тут, правда, нужно уточнить, что ширина табуляции в разных приложениях может различаться. В примере выше используется табуляция шириной в 4 пробела. В других приложениях ширина табуляции может быть 2 пробела или 8 пробелов.

Целочисленные переменные `int`

Переменная в C – это именованная область памяти компьютера, которая используется для хранения данных. Её значение может изменяться во время выполнения программы.

Переменные могут иметь разный тип. Разные типы переменных хранят разные виды данных. Переменные одного типа могут хранить целые числа, а переменные другого типа могут хранить дробные числа или строки. Переменные типа `int` предназначены для хранения целых чисел (как положительных так и отрицательных). `int` это сокращение от слова *integer*, что в переводе означает целое число.

Объявление переменных

В отличии от многих высокоуровневых языков программирования (таких, как Python), в языке C вы не можете сразу использовать переменную. Прежде чем использовать переменную, вам нужно её объявить. Для объявления переменных используется следующий синтаксис:

```
тип_переменной имя_переменной;
```

Например, чтобы объявить переменную типа `int` под названием `cat`, нужно написать:

```
int cat;
```

Инициализация переменных

Инициализация переменной, это присвоение переменной её начального значения. Инициализацию можно провести вместе с объявлением или после объявления.

```
int cat = 10;    // Объявление и инициализация
int dog;         // Сначала объявили
dog = 20;        // Потом инициализируем
```

Печать значений переменных, с помощью функции `printf`

Для вывода значений переменных на экран используется функция `printf`. Для этого в первом аргументе этой функции – строке форматирования – указывают, где именно должно появиться значение переменной. Например, чтобы вывести число типа `int`, в нужном месте строки пишут `%i`. Тогда вместо `%i` подставится значение переменной. `%i` и другие подобные включения в строку форматирования называются *спецификаторами*.

```
#include <stdio.h>
int main()
{
    int age = 10;
    printf("I am %i years old\n", age);
}
```

Эта программа напечатает на экран:

```
I am 10 years old
```

Для печати нескольких значений, можно использовать несколько спецификаторов `%i`:

```
#include <stdio.h>
int main()
{
    int age = 10;
    int n = 20;
    printf("I'm %i now, I'll be %i tomorrow. I have %i friends\n", age, age + 1, n);
}
```

Эта программа напечатает на экран:

```
I'm 10 now, I'll be 11 tomorrow. I have 20 friends.
```

Для того, чтобы напечатать на экран одно число, просто уберите из строки форматирования лишние слова:

```
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    printf("%i\n", a + b); //  Напечатает 30
}
```

Печать числа с фиксированной шириной поля

В строке форматирования можно задать минимальное число символов, которые будут использованы для печати числа. Например, чтобы ширина поля была минимум 3 символа нужно использовать спецификатор `%3i` вместо `%i`. Если число окажется меньше, то `printf` добавит необходимое количество пробелов перед числом. Если печатаемое число занимает больше символов, чем заданная ширина поля, то оно напечатается полностью.

```
#include <stdio.h>
int main()
{
    printf("%3i\n", 1);
    printf("%3i\n", 12);
    printf("%3i\n", 123);
}
```

Данная программ напечатает:

```
  1
 12
123
```

Если использовать спецификатор `%03i`, то вместо пробелов будут использоваться нули.

```
#include <stdio.h>
int main()
{
    printf("%03i\n", 1);
    printf("%03i\n", 12);
    printf("%03i\n", 123);
}
```

Данная программ напечатает:

```
001
012
123
```

Данный формат печати удобен, если вам нужно красиво напечатать таблицу чисел. Без такого форматирования числа будут "съезжать", так как разные числа будут занимать разное количество символов. Ещё один случай, когда такая печать может понадобиться, это печать некоторых форматов, например времени:

```
#include <stdio.h>
int main()
{
    int h = 12;
    int m = 30;
    printf("%02i:%02i\n", h, m);  //  напечатает 12:30

    h = 9;
    m = 5;
    printf("%02i:%02i\n", h, m);  //  напечатает 09:05
}
```

Адрес и размер переменной

Переменные в памяти

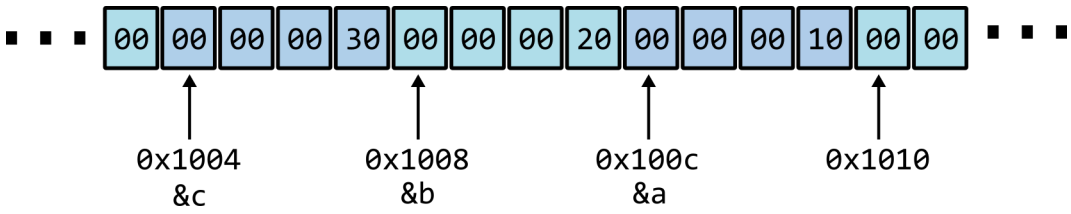
- 1 бит – минимальная единица измерения памяти. В 1 бите может хранится либо `0` либо `1`.
- Вся память делится на ячейки, размером в 8 бит = 1 байт.
- Все эти ячейки (байты) занумерованы, номер ячейки называется адресом. Байт – это минимальная адресуемая единица памяти.
- Переменные размещаются в памяти непрерывными блоками. Размер любой переменной составляет целое число байтов (не менее 1).
- У любой переменной можно получить её адрес. Адрес переменной – это адрес первого байта переменной.
- Чтобы найти адрес переменной, нужно перед ней поставить `&`, например, `&a`.
- Чтобы найти размер переменной в байтах: `sizeof(a)`.
- Переменная типа `int` *обычно* имеют размер 4 байта = 32 бита. Значит в ней может хранится максимум 2^{32} значений. То есть переменные типа `int` могут принимать значения от -2^{31} до $2^{31} - 1$.

Расположение переменных типа `int` в памяти

На большинстве систем переменные типа `int` занимают 4 байта. Соответственно, если вы создадите 3 переменные типа `int` вот так:

```
int a = 10;
int b = 20;
int c = 30;
```

то в памяти это может выглядеть вот так (на самом деле чуть сложнее, но это мы разберём в дальнейшем):



Адрес переменной – это адрес первого байта того участка памяти, который занимает данная переменная. Чтобы получить адрес переменной в языке C нужно написать перед именем переменной символ `&`. Для печати адреса с помощью `printf` используется спецификатор `%p`. В этом случае адрес напечатается в шестнадцатеричной системе счисления.

```
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    printf("Address of a = %p\n", &a);      //  Печатаем адрес переменной a
    printf("Address of b = %p\n", &b);      //  Печатаем адрес переменной b
    printf("Size of a = %zu\n", sizeof(a)); //  Печатаем размер переменной a
    printf("Size of b = %zu\n", sizeof(b)); //  Печатаем размер переменной a
}
```

Важно отметить, что адреса переменных могут быть разными при разных запусках программы. Операционная система рандомизируют адресное пространство для безопасности и гибкости.

Считывание переменных. Функция `scanf`.

Считывание — процесс получения данных из внешнего источника (клавиатура, файлы) и сохранения их в переменные. Для ввода данных с клавиатуры используется функция `scanf` из библиотеки `stdio.h`. При достижении этой функции выполнение программы приостанавливается до ввода пользователем данных. Данные нужно ввести в экран терминала. После ввода данные сохраняются в указанные переменные.

Считывание переменных типа `int`

Для того, чтобы считать значение переменной `a` нужно написать:

```
scanf("%i", &a);
```

Важно! Обратите внимание, что функции `scanf` нужно передавать именно *адрес переменной*, а не само значение переменной. В отличии от функции `printf`, куда нужно передавать саму переменную И это логично, ведь функции `printf` для печати нужно само значение переменной. А функции `scanf` текущее значение переменной не нужно, ей нужен только адрес того места, куда надо записать новое, считанное с экрана значение.
Приведём пример, простой программы, которая считывает число и печатает это число, увеличенное в 2 раза.

```
#include <stdio.h>
int main()
{
    int a;
    scanf("%i", &a);
    printf("%i\n", 2 * a);
}
```

Считывание нескольких переменных типа `int`

Можно считать несколько переменных, используя лишь один вызов функции `scanf`. В следующей программе считываются 2 числа и печатается из сумма.

```
#include <stdio.h>
int main()
{
    int a;
    int b;
    scanf("%i%i", &a, &b);
    printf("%i\n", a + b);
}
```

Форматированное считывание

Форматирование можно использовать не только у функции `printf`, но и у функции `scanf`. Рассмотрим, например, следующую программу:

```
#include <stdio.h>
int main()
{
    int a;
    int b;
    scanf("(%i,%i)", &a, &b);
    printf("%i\n", a + b);
}
```

Если мы на вход этой программе передадим `(10,20)`, то `scanf` корректно распарсит эту строку и запишет в переменную `a` значение `10`, а в переменную `b` – значение `20`. Однако, если вы введёте числа в неверном формате, то `scanf` вернёт ошибку.

Приведём ещё один пример программы, которая работает с форматированным вводом. Пусть на вход подаются 2 времени в формате `hh:mm`. Данная программа считывает эти времена, корректно складывает их и печатает в таком же формате.

```
#include <stdio.h>
int main()
{
    int h1, m1;
    scanf("%i:%i", &h1, &m1);
    int h2, m2;
    scanf("%i:%i", &h2, &m2);

    int total_minutes = 60 * h1 + m1 + 60 * h2 + m2;
    int h = total_minutes / 60;
    int m = total_minutes % 60;
    printf("%02i:%02i\n", h, m);
}
```

Если теперь на вход данной программе мы передадим

```
02:40
06:25
```

то программа напечатает

```
09:05
```

Функция `scanf` и пробельные символы

Пробельные символы играют важную роль при работе с функцией `scanf`. К пробельным символам относятся:

- Пробел (`" "`)
- Табуляция (`"\t"`)
- Перевод строки (`"\n"`)
- Другие пустые символы

Дело в том, что функция `scanf` часто "съедает" эти символы. Чтобы правильно её использовать, важно понимать, когда именно это происходит. Запомните два правила:

- Если `scanf` встречает в строке форматирования пробельный символ (или несколько подряд), она будет считывать все пробелы, табуляции и переводы строк до тех пор, пока не наткнётся на первый непробельный символ. При этом количество и тип пробелов в форматной строке и во вводе могут не совпадать.
- Если `scanf` встречает спецификатор для считывания числа (например, `%i`), она также пропускает все пробельные символы до первого непробельного, и только потом начинает считывать число.

  Например, если написать:

```
scanf("%i", &a);
```

и перед числом ввести в терминале пробелы, табуляции или переводы строк, то число всё равно будет корректно считано. Если использовать:

```
scanf("%i%i", &a, &b);
```

то пробельные символы можно ставить как перед первым числом, так и между числами – ввод всё равно будет обработан правильно. А запись:

```
scanf("%i    %i", &a, &b);
```

работает аналогично: при вводе разрешено любое количество пробелов или других пробельных символов между числами.

Некорректное использование `scanf` при считывании чисел

Одной из самых распространённых ошибок при использовании `scanf` демонстрируется в следующей программе:

```
#include <stdio.h>
int main()
{
    int a;
    scanf("%i\n", &a);
    printf("%i\n", a * a);
}
```

Эта программа должна была считывать число и печатать его квадрат, но программа почему-то работает не так как надо. Вместо того, чтобы просить одно число программа почему-то просит два числа. Но затем правильно печатает квадрат первого введённого числа.
Ошибка заключается в том, что в строке форматирования `scanf` содержится лишний пробельный символ `\n`:

```
scanf("%i\n", &a);
```

А как было сказано выше, как только `scanf` увидит любой пробельный символ в строке форматирования, он будет считывать все пробельные символы, пока не встретит первый непробельный.

Пределы

Максимальное и минимальное число, которое могут хранить переменные типа `int`

Операторы

Арифметические операторы

К целочисленным переменным, таким как переменные типа `int` можно применять арифметические операции, используя следующие операторы:

```
+          сложение
-         вычитание
*         умножение
/   целочисленное деление
%          остаток
```

Приведём пример программы, использующей эти операторы.

```
#include <stdio.h>
int main()
{
    int a = 17;
    int b = 7;

    printf("%i\n", a + b); //  напечатает 24
    printf("%i\n", a - b); //  напечатает 10
    printf("%i\n", a * b); //  напечатает 119
    printf("%i\n", a / b); //  напечатает 2 (так 17 / 7 = 2 (3 в остатке))
    printf("%i\n", a % b); //  напечатает 3 (так 17 / 7 = 2 (3 в остатке))
}
```

Если вы никогда не программировали, принцип целочисленного деления может показаться неочевидным. Поскольку тип `int` предназначен для целых чисел, результат любой операции между ними также будет целым числом. Поэтому при делении дробная часть просто отбрасывается.

Примеры использования оператора остатка %

Приведём примеры использования оператора остатка.

- Программа, которая считывает число и печатает последнюю цифру числа в десятичной записи:

```
#include <stdio.h>

int main()
{
    int a;
    scanf("%i", &a);
    printf("%i\n", a % 10);
}
```

- Программа, которая считывает число и печатает `0`, если число чётное и `1`, если нечётное:

```
#include <stdio.h>

int main()
{
    int a;
    scanf("%i", &a);
    printf("%i\n", a % 2);
}
```

Оператор присваивания

Оператор присваивания(=) используется для присвоения значений переменным. Также, как и другие бинарные операторы он требует двух операндов. Но в отличии от арифметических операторов он изменяет значение одного из своих операндов, а именно, после присваивания значение левого операнда изменится и станет равно значению правого операнда.

```c
#include <stdio.h>
int main()
{
    int a;              //  Объявляем переменную
    a = 10;             //  Задаём значение, используя оператор присваивания =
    printf("%i\n", a);  //  Напечатает 10
}
```

Следует отличать оператор присваивания от инициализации при объявлении переменной.

```c
#include <stdio.h>
int main()
{
    int a = 10;         //  Объявляем и инициализируем переменную
                        //  Тут оператор присваивания не используется

    printf("%i\n", a);  //  Напечатает 10
}
```

Несмотря на использование одинакового символа = и схожего конечного результата, инициализация при объявлении и операция присваивания фундаментально различаются на семантическом уровне. Для компилятора это различные операции, выполняемые на разных этапах работы с переменной.

Возвращаемое значение оператора присваивания

Оператор присваивания не только изменяет значение левого операнда. Он также, как и все другие операторы, возвращает значение. А именно, он возвращает новое значение левого операнда.

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 20;
    printf("%i\n", a = b);  //  Напечатает 20
}
```

Составные операторы присваивания

Помимо обычного оператора присваивания существуют составные операторы, которые объединяют арифметическую операцию с операцией присваивания.

```
=   оператор присваивания                 присвоить левой части правую
+=  оператор присваивания сложения       прибавить к левой части правую
-=  оператор присваивания вычитания      отнять от левой части правую
*=  оператор присваивания умножения      умножить левую часть на правую
/=  оператор присваивания деления        разделить левую часть на правую
%=  оператор присваивания взятия остатка  левая часть становится равна остатку
```

Также как обычный оператор присваивания, все эти операторы возвращают новое значение левого операнда.

Операторы инкремента и декремента

Операторы инкремента и декремента увеличивают или уменьшают значение переменной на 1.

`++`	увеличить на 1
`--`	уменьшить на 1

Это унарные операторы, то есть они применяются только к одно переменной. Различают префиксные операторы инкремента (пишутся перед переменной) и постфиксные (пишутся после переменной).

```c
#include <stdio.h>
int main()
{
    int a = 10;
    ++a;                //  Префиксный инкремент
    a++;                //  Постфиксный инкремент
    printf("%i\n", a);  //  Напечатает 12
}
```

Возвращаемое значение операторов инкремента и декремента

Префиксный и постфиксный операторы инкремента и декремента различаются возвращаемым значением.

- Префиксный (`++a`) возвращает новое значение.
- Постфиксный (`a++`) возвращает старое значение.

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 10;
    printf("%i\n", ++a);  //  Напечатает 11
    printf("%i\n", b++);  //  Напечатает 10
    printf("%i\n", a);    //  Напечатает 11
    printf("%i\n", b);    //  Напечатает 11
}
```

Операторы сравнения

Операторы сравнения в языке C всегда возвращают целые числа типа `int`.

оператор	возвращаемое значение
`a == b`	1 если равны, иначе 0
`a != b`	1 если не равны, иначе 0
`a > b`	1 если больше, иначе 0
`a >= b`	1 если больше или равно, иначе 0
`a < b`	1 если меньше, иначе 0
`a <= b`	1 если меньше или равно, иначе 0

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    printf("%i\n", a < b);  //  Напечатает 1
    printf("%i\n", a == b); //  Напечатает 0
}
```

Булевая алгебра

В булевой алгебре есть два элемента 0 и 1. Часто эти элементы также называют *false* (ложь) и *true* (истина). Также в булевой алгебре есть 3 основные операции:

- **Логическое И** – истинно только тогда, когда оба операнда истинны.

```
0 И 0 = 0
0 И 1 = 0
1 И 0 = 0
1 И 1 = 1
```

- **Логическое ИЛИ** – истинно только тогда, когда хотя бы один из операндов истинен.

```
0 ИЛИ 0 = 0
0 ИЛИ 1 = 1
1 ИЛИ 0 = 1
1 ИЛИ 1 = 1
```

- **Логическое НЕ** – инвертирует значение операнда.

```
НЕ 0 = 1
НЕ 1 = 0
```

Логические операторы

В языке C в логические операторы обозначаются следующим образом:

```
&&   логическое И
||  логическое ИЛИ
 !   логическое НЕ
```

Операнды логических операторов в языке C – это целые числа. При этом они интерпретируются так:

- Число, равное нулю – воспринимается как элемент 0 (ложь).
- Любое число, отличное от нуля – воспринимается как элемент 1 (истина).

Все логические операторы возвращают `0` или `1` – целое число типа `int`.
Пример программы, которая печатает возвращаемые значения логических операторов:

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 0;
    printf("%i\n", a && b);     //  Напечатает 0
    printf("%i\n", a || b);     //  Напечатает 1
    printf("%i\n", !a);         //  Напечатает 0
}
```

Пример программы, которая использует логические операторы, совместно с другими операторами:

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int b = 20;
    printf("%i\n", (a < b) && !(a % 2));        //  Напечатает 1, так как a чётное и меньше b
    printf("%i\n", (a < b) + (a && b) + !!a);   //  Напечатает 3
}
```

Приоритет операторов

Таблица содержит все операторы языка C и уровень их приоритета от наиболее к наименее приоритетным.

приоритет	операторы	описание
1	`++ --`	постфиксный инкремент/декремент
	`()`	оператор вызова функции
	`[]`	оператор индексирования
	`. ->`	оператор точка и стрелочка
2	`++ --`	префиксный инкремент/декремент
	`+ -`	унарный плюс и минус
	`! ~`	логическое НЕ и побитовое НЕ
	`(тип)`	оператор приведения к типу
	`*`	разыменование указателя
	`&`	оператор получения адреса
	`sizeof`	оператор получения размера
	`alignof`	оператор получения выравнивания
3	`* / %`	умножение, деление, остаток
4	`+ -`	сложение и вычитание
5	`<< >>`	побитовый сдвиг влево и вправо
6	`< <= > >=`	операторы сравнения
7	`== !=`	операторы сравнения на равенство/неравенство
8	`&`	побитовое И
9	`^`	побитовое исключающее ИЛИ
10	`|`	побитовое ИЛИ
11	`&&`	логическое И
12	`||`	логическое ИЛИ
13	`?:`	тернарный оператор
14	`=`	присваивание
	`+= -= *= /= %=`	составные присваивания
	`>>= <<= &= ^= |=`	составные побитовые присваивания
15	`,`	оператор запятая

Эта таблица поможет понять вам в каком порядке вычисляется выражение. Например, в следующем выражении:

```
d = a + b * c;
```

сначала выполнится умножение, затем сложение и в конце присваивание. Если нужно изменить порядок вычислений, можно использовать скобки:

```
d = (a + b) * c;
```

в таком случае сначала выполнится сложение, потом умножение и в конце присваивание.

Ассоциативность операторов

Ассоциативность определяет, в каком порядке выполняются операторы с одинаковым приоритетом (слева направо или справа налево). В C операторы делятся на два типа по ассоциативности:

- Лево-ассоциативные – вычисляются слева направо.

  `a - b - c - d` вычисляется как `((a - b) - c) - d`

- Право-ассоциативные – вычисляются справа налево.

  `a = b = c = d` вычисляется как `a = (b = (c = d))`

Инструкции

Условная инструкция `if`

Оператор if принимает число в круглых скобках и если это число не равно 0, то исполняется одна инструкция после оператора if

Инструкция - это выражение, заканчивающееся точкой с запятой

```
if ( условие )
    сделай это
```

```
if ( условие1 )
    сделай это
else if ( условие2 )
    сделай это
else
    сделай это
```

Цикл `while`

```
while ( условие )
{
    делай это
}
```

Цикл `for`

Циклом `for` обычно удобнее пользоваться. Соответствие между циклами `while` и `for`:

```
инициализация
while ( условие )
{
    делай это
    обновление
}
```

-->

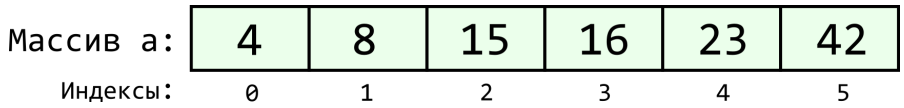
```
for ( инициализация; условие; обновление )
{
    делай это
}
```

Массивы

Массивы - это объекты, которые могут хранить внутри себя большое количество других объектов одного типа. Например, мы можем создать массив, который будет хранить 6 чисел типа `int` вот так:

```
int a[6] = {4, 8, 15, 16, 23, 42};
```

После того, как мы создали массив, мы можем получать доступ к каждому элементу массива по номеру. Номер элемента массива также называется его индексом. При этом нумерация в массиве начинается с нуля.



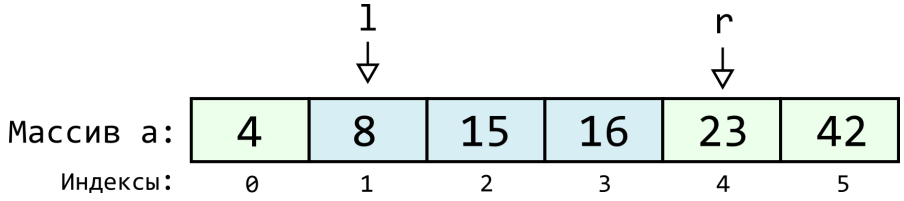
Доступ к элементу по индексу осуществляется через квадратные скобки. Например, если мы хотим поменять в массиве, определённом выше, число 15 на 20 нужно написать:

```
a[2] = 20;
```

Подмассивы

Подмассив - это некоторая последовательная часть массива. В языке C нет никаких специальных средств для работы с подмассивами. Мы будем задавать подмассив в коде как два числа – индексы граничных элементов. Будем обозначать подмассивом `a[l, r]` такую часть массива, элементы которого имеют индекс `i` в диапазоне `l <= i < r`. Обратите внимание, что мы договорились, что элемент `a[r]` не входит в подмассив `a[l, r]`.

Например, в подмассив `[1, 4]` массива `a` входят элементы `8, 15, 16`, а элемент `23` не входит.



Сортировка

Сортировка – это упорядочение элементов по возрастанию, убыванию или по какому-то другому критерию.

Сортировка выбором

Сортировка выбором – это простейший алгоритм сортировки, который заключается в следующем:
Для каждого подмассива `[j, n]` (где `j` последовательно меняется от `0` до `n - 1`) поменять местами первый и минимальный элементы этого подмассива.

```
for (int j = 0; j < n; ++j)
{
    int min_index = j;
    for (int i = j + 1; i < n; ++i)
    {
        if (a[i] < a[min_index])
            min_index = i;
    }

    int temp = a[j];
    a[j] = a[min_index];
    a[min_index] = temp;
}
```

Сортировка пузырьком

Сортировка пузырьком – это простейший алгоритм сортировки, который заключается в следующем:
Для каждого подмассива `[0, n - j]` (где `j` последовательно меняется от `0` до `n - 1`) мы делаем следующую операцию: пробегаем по этому подмассиву и, если соседние элементы стоят неправильно, то меняем их местами.

```
for (int j = 0; j < n; ++j)
{
    for (int i = 0; i < n - 1 - j; i += 1)
    {
        if (a[i] > a[i + 1])
        {
            int temp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = temp;
        }
    }
}
```

Бинарный поиск на отсортированном массиве

Если известно, что массив уже отсортирован, то многие задачи на таком массиве можно решить гораздо проще и/или эффективней. Например, просто найти минимум, максимум и медианное значение. Одной из задач, которая быстрее решается на отсортированном массиве – это задача поиска элемента в массиве. Если массив отсортирован, то решить эту задачу можно гораздо быстрее чем простой обход всех элементов.

Предположим, что массив отсортирован по возрастанию и надо найти элемент `x` в этом массиве или понять, что такого элемента в массиве не существует. Для этого мы мысленно разделим массив на 2 части:

1. Элементы, которые меньше, чем `x`
2. Элементы, которые больше или равны `x`

Затем введём две переменные-индекса `l` и `r`. В начале работы алгоритма индекс `l` будет хранить индекс фиктивного элемента, находящегося до первого (то есть `l = -1`), а индекс `r` будет хранить индекс фиктивного элемента, находящимся после последнего (то есть `r = n`).

На каждом шаге алгоритма мы будем брать середину между индексами `l` и `r` и передвигать к этой середине или индекс `l` или индекс `r`. При этом при изменении индексов должны соблюдаться условия:

```
a[l] < x
a[r] >= x
```

Алгоритм закончится тогда, когда разница между индексами не станет равным 1, то есть не станет `r == l + 1`. И так как `a[l] < x` и `a[r] >= x`, то если элемент `x` в массиве существует, то его индекс равен `r`.

Код для поиска в отсортированном массиве бинарным поиском:

```
#include <stdio.h>

int main()
{
    int n;
    int a[1000];
    scanf("%i", &n);
    for (int i = 0; i < n; ++i)
        scanf("%i", &a[i]);

    int x;
```

```
    scanf("%i", &x);

    int l = -1, r = n;
    while (r > l + 1)
    {
        int mid = (l + r) / 2;

        if (a[mid] >= x)
            r = mid;
        else
            l = mid;
    }

    if (r < n && a[r] == x)
        printf("Element found! Index = %i\n", r);
    else
        printf("Element not found!");
}
```