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

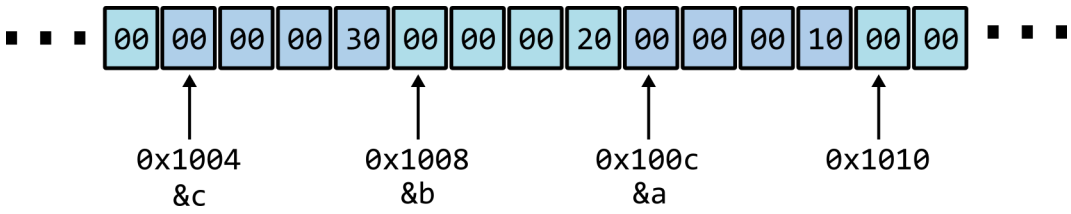
- 1 бит – минимальная единица измерения памяти. В 1 бите может хранится либо `0` либо `1`.
- Вся память делится на ячейки, размером в 8 бит = 1 байт.
- Все эти ячейки (байты) занумерованы, номер ячейки называется адресом. Байт – это минимальная адресуемая единица памяти.
- Переменные размещаются в памяти 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.

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

Определение понятия "литерал"

Литерал – это непосредственно записанное в коде фиксированное значение, которое используется в программе "как есть", а не вычисляется во время выполнения. Например, в строках:

```
int a = 10;
printf("Hello\n");
```

Число `10` является целочисленным литералом, а строка `"Hello\n"` – строковым литералом.

Определение понятия "выражение" в C

Выражение (англ. *expression*) – это последовательность операторов и их операндов, задающих некоторое вычисление. В качестве операндов могут служить имена переменных, имена функций и литералы. Обратите внимание, что вызов функции это тоже оператор (оператор круглые скобочки). Примерами выражений могут быть:

```
a + b - 10 / c
printf("Hello\n")
a = 20
a   // Простейшее выражение, состоящее из одной имени переменной
10  // Простейшее выражение, состоящее из одного литерала
```

Определение понятия "объявление" в C

Объявление (англ. *declaration*) – это конструкция языка, которая добавляет ещё одно имя (идентификатор) в программу и задаёт некоторые свойства этого имени, например такие как тип и начальное значение переменной.

```
int a;
int b = 10;
```

Определение понятия "инструкция" в C

Инструкция (англ. *statement*) – это отдельная команда, которую компилятор воспринимает как законченное действие. Инструкции можно разделить на следующие категории:

- *Пустая инструкция.* Ничего не делает.

  ```
  ;
  ```

- *Инструкции выражения.* Состоит из выражения с точкой с запятой. Просто исполняет это выражение.

  ```
  a + b - 10 / c;
  printf("Hello\n");
  a = 20;
  10;     // Простейшая инструкция выражения, состоит из одного литерала. Ничего не делает
  a;      // Простейшая инструкция выражения, состоит из одного имени. Ничего не делает
  ```

- *Составная инструкция (блок).* Группа из нуля или более инструкций и/или объявлений, заключенная в фигурные скобки `{}`. Пример одной составной инструкции:

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

- *Инструкции управления потоком выполнения.* Это инструкции, которые позволяют изменять порядок выполнения программы: переходить по условиям, повторять части кода или завершать работу функции/-программы. В эту категорию входят условная инструкция `if`, инструкции циклов `for` и `while` и другие.

Блоки и области видимости

Блок – это составная инструкция, то есть группа из нуля или более инструкций и/или объявлений, заключенная в фигурные скобки.
Область видимости – часть программы, где доступно имя переменной или функции.
Блочная область видимости – переменные, объявленные внутри блока, доступны только внутри него.

```c
#include <stdio.h>
int main()
{
    int a = 10;

    {
        int b = 20;         // b не будет видна вне блока, в котором объявлена
        printf("%i\n", a);  // OK, переменная a видна
        printf("%i\n", b);  // OK, переменная b видна
    }

    printf("%i\n", a);      // OK, переменная a видна
    printf("%i\n", b);      // Ошибка, b вне области видимости
}
```

Повторное объявление переменной

В языке C нельзя объявить две переменных с тем же самым именем в одном и том же блоке.

```c
#include <stdio.h>
int main()
{
    int a = 10;
    int a = 20;             // Ошибка, повторное объявление a

    {
        int b = 20;
        int b = 30;         // Ошибка, повторное объявление b
    }
}
```

Затенение переменных в блоке

Если внутри блока объявить переменную с тем же именем, что и внешняя переменная, то внутренняя затенит внешнюю.

```c
#include <stdio.h>
int main()
{
    int a = 10;

    {
        printf("%i\n", a);  // Видит только внешнюю a. Напечатает 10.
        int a = 20;         // Внутренняя переменная a затеняет внешнюю с тем же именем.
        printf("%i\n", a);  // Напечатает 20.
    }

    printf("%i\n", a);      // Тут внутренняя переменная уже не видна. Напечатает 10.
}
```

Инструкции управления потоком выполнения

Условная инструкция `if`

```
if ( условие )
    инструкция
```

Инструкция `if` принимает в круглых скобках условие – выражение, которое вычисляется в целое число.

- Если условие не равно `0`, то следующая после `if(...)` инструкция исполняется.
- Если условие равно `0`, то следующая после `if(...)` инструкция пропускается.

Следующая программа напечатает строки `Cat` и `Mouse`, но не напечатает `Dog`:

```
#include <stdio.h>
int main()
{
    if (1)
        printf("Cat\n");

    if (0)
        printf("Dog\n");

    if (-10)
        printf("Mouse\n");
}
```

Чаще всего инструкция `if` используется вместе с операторами сравнения:

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

    if (a > 0)
        printf("Positive\n");
}
```

Это работает благодаря тому, что операторы сравнения в C возвращают число 1 при истинности условия и 0 при его ложности.

Также часто используются логические операторы:

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

    if (a >= 10 && a < 100)
        printf("Positive two-digit number\n");

    if (a < 0 || a % 2)
        printf("Negative or odd number\n");
}
```

Инструкция `if` и блоки

Нужно помнить, что `if` "применяется" только к одной инструкции, следующий после неё. Например, код:

```
if (a < 0)
    printf("Cat\n");
    printf("Dog\n");
```

напечатает строку `Dog`, даже если переменная `a` больше нуля, так как к `if` относится только строка, печатающая `Cat`. Другими словами, компилятор воспринимает этот код так:

```
if (a < 0)
    printf("Cat\n");
printf("Dog\n");
```

Чтобы к `if` относилось несколько инструкций, нужно объединить эти инструкции с помощью блока:

```
if (a < 0)
{
    printf("Cat\n");
    printf("Dog\n");
}
```

Разновидность условной инструкции `if else`

```
if ( условие )
    инструкция1
else
    инструкция2
```

- Если выражение не равно `0`, то исполняется инструкция, следующая после `if(...)`.
- Если выражение равно `0`, то исполняется инструкция, следующая после `else`.

```
#include <stdio.h>
int main()
{
    int a;
    scanf("%i", &a);
    if (a)
        printf("Non-zero\n");
    else
        printf("Zero\n");
}
```

Если нужно объединить несколько зависимых условий, то можно вспомнить, что `if` и `if else` сами являются инструкциями, поэтому их можно подставить за место инструкции внутри другого `if else` так:

```
if ( условие1 )
    инструкция1
else
    if ( условие2 )
        инструкция2
    else
        if ( условие3 )
            инструкция3
        else
            инструкция4
```

что эквивалентно:

```
if ( выражение1 )
    инструкция1
else if ( выражение2 )
    инструкция2
else if ( выражение3 )
    инструкция3
else
    инструкция4
```

Приведём пример программы, использующей несколько зависимых условий:

```
#include <stdio.h>
int main()
{
    int month;
    scanf("%i", &month);

    if (month == 12 || month == 1 || month == 2)
    {
        printf("Winter\n");
    }
    else if (month >= 3 && month <= 5)
    {
        printf("Spring\n");
    }
    else if (month >= 6 && month <= 8)
    {
        printf("Summer\n");
    }
    else if (month >= 9 && month <= 11)
    {
        printf("Autumn\n");
    }
    else
    {
        printf("Error!\n");
    }
}
```

Инструкция цикла `while`

```
while ( условие )
    инструкция
```

Также как и `if`, инструкция `while` принимает в круглых выражение, которое вычисляется в целое число. Будем называть это выражение условием.

- Если условие равно `0`, то следующая после `while` инструкция не выполняется.
- Если условие не равно `0`, то инструкция выполняется. После этого условие снова проверяется, его значение могло измениться за время выполнения инструкции. Если условие вновь не равно `0`, то инструкция снова выполняется и снова проверяется условие. Так продолжается пока условие не станет равным 0.

Рассмотрим пример использования цикла `while`. В этой программе составная инструкция, следующая после `while` исполнится 5 раз пока значение переменной `a` не станет равным нулю.

```
#include <stdio.h>
int main()
{
    int a = 5;
    while (a)
    {
        printf("%i\n", a);
        a -= 1;
    }
}
```

Бесконечные циклы

Бесконечный цикл – это цикл, который выполняются бесконечно, так как их условие никогда не становится равным нулю. Пример программы, содержащей бесконечный цикл:

```
#include <stdio.h>
int main()
{
    while (1)
        printf("Hello\n");
}
```

Если такую программу запустить в терминале и не предпринимать никаких дополнительных действий по её завершению, то она будет работать вечно, потребляя ресурсы системы. Для принудительного завершения программы в терминале используйте сочетание клавиш `Ctrl-C`.

Тело цикла и итерирование

Чаще всего инструкцией, следующей после `while` является составная инструкция (блок). То есть цикл `while` часто выглядит так:

```
while ( условие )
{
    инструкция1
    инструкция2
    ...
}
```

Набор инструкций внутри блока, следующего после `while`, называется *телом цикла*. Один проход по телу цикла называют *итерацией*, а весь процесс многократных итераций – *итерированием*.

Итерирование до заданного числа

Часто требуется произвести итерирование от нуля до некоторого заданного числа. Разберём этот случай на примере следующей программы, которая считывает число n, а затем n раз печатает на экран строку `"Hello"`:

```
#include <stdio.h>
int main()
{
    int n;
    scanf("%i", &n);
    int i = 0;
    while (i < n)
    {
        printf("Hello\n");
        i += 1;
    }
}
```

Для этой задачи мы завели специальную целочисленную переменную `i`, чьё назначение заключается в том чтобы проверяться в условии перед каждой итерацией и изменяться в конце каждой итерации. Такая переменная называется *счётчик* (англ. *counter*). Общепринято использовать имя `i` для таких переменных. Для увеличения счётчика в конце итерации часто используется оператор инкремента:

```
int i = 0;
while (i < 5)
{
    printf("Hello\n");
    i++;
}
```

Считывание n чисел в цикле

Часто возникает задача обработки последовательности чисел, поступающих на вход. Во многих случаях нет необходимости сохранять все эти числа в массив для последующей обработки (работа с массивами будет рассмотрена несколько позже). Вместо этого можно последовательно считывать каждое число в одну и ту же переменную и немедленно обрабатывать его.

Рассмотрим программу, на вход которой поступает пять чисел, а она печатает эти числа, увеличенные на 1:

```
#include <stdio.h>
int main()
{
    int i = 0;
    while (i < 5)
    {
        int a;
        scanf("%i", &a);
        printf("%i ", a + 1);
        ++i;
    }
    printf("\n");
}
```

Если на вход этой программе передать:

```
10 20 30 40 50
```

то она напечатает:

```
11 21 31 41 51
```

Для обработки последовательности чисел произвольной длины, можно первым числом подавать на вход количество элементов в последовательности, а затем уже подавать сами числа. В самой программе мы сначала используем `scanf` один раз для считывания количества чисел, а затем в цикле считываем сами числа.

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

    int i = 0;
    while (i < n)
    {
        int a;
        scanf("%i", &a);
        printf("%i ", a + 1);
        ++i;
    }
    printf("\n");
}
```

Если этой программе на вход передать:

```
7
10 20 30 40 50 60 70
```

то она напечатает:

```
11 21 31 41 51 61 71
```

Можно передавать произвольное количество чисел без указания их количества. В этом случае программа самостоятельно определяет конец последовательности и количество чисел в ней. Такой будет рассмотрен позднее в этом семинаре.

Примеры программ, использующих цикл `while`

- Программа, которая считывает число n и печатает n звёздочек `*`. Например, если на вход пришло число 5, то программа должна напечатать `*****`.

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

    int i = 0;
    while (i < n)
    {
        printf("*");
        ++i;
    }
    printf("\n");
}
```

- Программа, которая принимает на вход последовательность из n чисел и печатает сумму этих чисел. Например, если на вход приходит:

```
5
10 20 30 40 50
```

то программа должна напечатать `150`.

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

    int i = 0;
    int sum = 0;
    while (i < n)
    {
        int a;
        scanf("%i", &a);
        sum += a;
        ++i;
    }
    printf("%i\n", sum);
}
```

- Программа, которая считывает число и печатает факториал этого числа:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

```c
#include <stdio.h>
int main()
{
    int n;
    scanf("%i", &n);

    int i = 1;
    int fact = 1;
    while (i <= n)
    {
        fact *= i;
        ++i;
    }
    printf("%i\n", fact);
}
```

- Программа, которая считывает число n и печатает n-е число Фибоначчи. Числа Фибоначчи задаются следующими формулами:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_k = F_{k-2} + F_{k-1}$$

```c
#include <stdio.h>
int main()
{
    int n;
    scanf("%i", &n);

    int fcurrent = 0;
    int fnext = 1;

    int i = 0;
    while (i < n)
    {
        int temp = fnext;
        fnext = fnext + fcurrent;
        fcurrent = temp;

        ++i;
    }

    printf("%i\n", fcurrent);
}
```

Инструкция цикла `do while`

```
do инструкция
while ( условие );
```

Цикл `do while` в целом похож на цикл `while`, но отличается от него тем, что условие цикла проверяется в конце каждой итерации, а не в начале, как в `while`. В результате тело цикла `do while` выполнится как минимум один раз.

```
#include <stdio.h>
int main()
{
    // Этот цикл do while напечатает числа от 1 до 5
    int i = 0;
    do
    {
        printf("%i ", i + 1);
        ++i;
    }
    while (i < 5);

    // Этот цикл do while выполнится один раз
    do
    {
        printf("Hello\n");
    }
    while (0);
}
```

Обратите внимание, что инструкция цикла `do while` заканчивается точкой с запятой.

Инструкция цикла `for`

Цикл `for` является самым часто используемым циклом в языках C и C++. Он позволяет записывать итерации с переменной-счётчиком более компактно, чем цикл `while`. Цикл `for` это лучший выбор, когда нужно произвести известное количество итераций, например когда нужно работать с массивом элементов известной длины.

```
for ( инициализация  условие; обновление )
    инструкция
```

В данных обозначениях:

инициализация	инструкция объявления, выражения или пустая, которая выполняется до итерирования
условие	выражение, вычисляемое в целое, также, как и в цикле while
обновление	выражение, вычисляемое после каждой итерации
инструкция	тело цикла, то что вычисляется на каждой итерации

Между циклами `while` и `for` можно провести соответствие. Эти циклы являются взаимозаменяемыми, то есть всегда можно заменить одни цикл на другой. Однако на практике цикл `for` часто оказывается предпочтительнее.

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

Примеры программ, использующих цикл `for`

- Программа, которая 5 раз печатает на экран строку `"Hello"`:

```c
#include <stdio.h>

int main()
{
    for (int i = 0; i < 5; i++)
        printf("Hello\n");
}
```

- Программа, которая печатает числа от 1 до 10:

```c
#include <stdio.h>

int main()
{
    for (int i = 0; i < 10; ++i)
        printf("%i ", i + 1);
    printf("\n");
}
```

- Программа, которая считывает число n и печатает на экран n звёздочек `*`:

```c
#include <stdio.h>

int main()
{
    int n;
    scanf("%i", &n);

    for (int i = 0; i < n; ++i)
        printf("*");

    printf("\n");
}
```

- Программа, которая вычисляет факториал числа:

```c
#include <stdio.h>

int main()
{
    int n;
    scanf("%i", &n);

    int fact = 1;
    for (int i = 1; i <= n; ++i)
        fact *= i;

    printf("%i\n", fact);
}
```

- Программа, на вход которой поступает последовательность из n чисел, а она вычисляет сумму этих чисел:

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

    int sum = 0;
    for (int i = 0; i < n; ++i)
    {
        int a;
        scanf("%i", &a);
        sum += a;
    }
    printf("%i\n", sum);
}
```

- Программа, на вход которой поступает последовательность из n чисел, а она вычисляет максимальный элемент из этих чисел. Предполагаем, что $n > 0$.

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

    int max;
    scanf("%i", &max);

    for (int i = 1; i < n; ++i)
    {
        int a;
        scanf("%i", &a);
        if (a > max)
            max = a;
    }
    printf("%i\n", max);
}
```

Итерирование от 0 или от 1

Предположим, что нам нужно сделать n итераций с помощью цикла `for`. Распространены два способа:

- Итерирование от 0 до $n - 1$:

```
for (int i = 0; i < n; ++i)
```

- Итерирование от 1 до n:

```
for (int i = 1; i <= n; ++i)
```

Обычно предпочитается первый способ, так как он наиболее удобен при работе с массивами.

Другие варианты итерирования

Цикл `for` очень гибок, с помощью него можно итерироваться не только от 0 до n. В следующей программе представлены несколько других вариантов итерирования:

```
#include <stdio.h>
int main()
{
    // Итерируем от 10 до 20 не включительно
    for (int i = 10; i < 20; ++i)
        printf("%i ", i);
    printf("\n");

    // Итерируем от 10 до 0 не включительно, обратный отсчёт
    for (int i = 10; i > 0; --i)
        printf("%i ", i);
    printf("\n");

    // Итерируем от 0 до 70 с шагом 7
    for (int i = 0; i <= 70; i += 7)
        printf("%i ", i);
    printf("\n");
}
```

Область видимости переменной счётчика

Одно из преимуществ цикла `for` перед циклом `while` является то, что область видимости переменной-счётчика в цикл `for` ограничена только самим циклом:

```
#include <stdio.h>
int main()
{
    for (int i = 0; i < 5; ++i)
    {
        printf("%i\n", i);
    }
    // После цикла переменная i не видна
    printf("%i\n", i);  // Ошибка
}
```

Это является преимуществом, так как переменная не засоряет внешнюю область видимости и впоследствии можно использовать переменную с именем `i` в других циклах. Если же необходимо использовать переменную `i` вне области видимости цикла, то можно её объявить до цикла, а в самом цикле только инициализировать:

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 5; ++i)
    {
        printf("%i\n", i);
    }
    // Теперь переменная i видна после цикла
    printf("%i\n", i);  // OK, напечатает 5
}
```

Вложенные циклы

Часто внутри одной итерации цикла требуется выполнить ещё один цикл. Это реализуется с помощью вложенных циклов – когда один цикл размещается внутри тела другого. При этом важно использовать для каждого цикла разные имена переменных-счётчиков (например, `i` для внешнего и `j` для внутреннего), чтобы избежать ошибок.

```
#include <stdio.h>
int main()
{
    for (int i = 0; i < 4; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            printf("%i, %i\n", i, j);
        }
    }
}
```

Инструкции `break` и `continue`

Инструкции `break` и `continue` – это операторы управления циклом в языке C. Они позволяют гибко контролировать поток выполнения внутри циклов.

- `break` – немедленно прерывает выполнение цикла и передает управление за пределы цикла.

  ```
  #include <stdio.h>
  int main()
  {
      for (int i = 0; i < 10; i++)
      {
          if (i == 5)
              break; // цикл прервется, когда i станет равным 5

          printf("%i ", i);
      }
      printf("\n");
  }
  ```

  В результате данная программа напечатает: `0 1 2 3 4`.

- `continue` – немедленно прерывает текущую итерацию цикла и переходит к следующей. Код, находящийся после `continue` внутри тела цикла, выполнен не будет.

  ```
  #include <stdio.h>
  int main()
  {
      for (int i = 0; i < 10; i++)
      {
          if (i == 5)
              continue; // итерация прервется, цикл перейдёт на следующую итерацию

          printf("%i ", i);
      }
      printf("\n");
  }
  ```

  В результате данная программа напечатает: `0 1 2 3 4 6 7 8 9`.

Пример программы, которая печатает все простые числа до 5000.
Простое число – это целое число, большее единицы, которое делится только на само себя и на 1.

```
#include <stdio.h>
int main()
{
    for (int a = 2; a <= 5000; ++a)
    {
        int is_prime = 1;
        for (int i = 2; i * i <= a; ++i)
        {
            if (a % i == 0)
            {
                is_prime = 0;
                break;
            }
        }

        if (is_prime)
            printf("%i ", a);
    }
    printf("\n");
}
```

Завершение двух вложенных циклов

В случае вложенных циклов оператор `break` обеспечивает выход только из текущего (внутреннего) цикла, тогда как выполнение внешнего цикла продолжается. Для одновременного выхода из обоих циклов необходимо использовать дополнительную переменную-флаг, которая сигнализирует о необходимости завершения.

```
#include <stdio.h>
int main()
{
    int flag = 0;
    for (int i = 0; i < 5 && !flag; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            printf("%i, %i\n", i, j);

            if (i == 2 && j == 3)
            {
                flag = 1;
                break;
            }
        }
    }
}
```

Пропуски в цикле `for`

Цикл `for` в круглых скобочках содержит 3 части и любую из них можно пропустить:

```
#include <stdio.h>
int main()
{
    // Пропуск инициализации в круглых скобках:
    int i = 0;
    for (; i < 3; ++i)
    {
        printf("%i\n", i);
    }

    // Пропуск условия в круглых скобках:
    for (int i = 0;; ++i)
    {
        if (i >= 3)
            break;
        printf("%i\n", i);
    }

    // Пропуск обновления в круглых скобках:
    for (int i = 0; i < 3;)
    {
        printf("%i\n", i);
        ++i;
    }

    // Пропуск условия и обновления в круглых скобках:
    for (int i = 0;;)
    {
        if (i >= 3)
            break;
        printf("%i\n", i);
        ++i;
    }
}
```

Бесконечный цикл `for`

Как и цикл `while`, цикл `for` может быть бесконечным. Простейший бесконечный цикл `for` можно написать, если пропустить все части цикла `for` в круглых скобочках:

```
#include <stdio.h>
int main()
{
    // Пропуск всего в круглых скобках - бесконечный цикл
    for (;;)
    {
        printf("Hello\n");
    }
```

В результате данная программа будет бесконечно печатать строку `"Hello"`. Чтобы принудительно завершить программу нужно воспользоваться комбинацией клавиш `Ctrl-C`.

Инструкция выбора `switch`

Инструкция `switch` выполняет код в соответствии со значением целочисленного аргумента. Используется, когда требуется выполнить тот или иной код в соответствии со значением целочисленного аргумента. Обычно эта инструкция имеет следующий вид:

```
switch ( выражение )
{
    case константа1:
        код1
        break;
    case константа2:
        код2
        break;
    //...
    default:
        код, если ни один case не подошел
}
```

В теле инструкции `switch` можно использовать специальные *метки* `case` и `default`.
Пример программы, которая считывает число и если это число 1, 2 или 3, то печатает `One`, `Two` или `Three` соответственно, в ином случае печатает `Other`.

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

    switch (a)
    {
    case 1:
        printf("One\n");
        break;
    case 2:
        printf("Two\n");
        break;
    case 3:
        printf("Three\n");
        break;
    default:
        printf("Other\n");
        break;
    }
}
```

Особенности инструкции `switch`:

- Выражение в круглых скобках после `switch` может быть только целочисленным.
- Числа после `case` должны быть целочисленными константными выражениями, известными на этапе компиляции.
- Метки не должны дублироваться.
- Благодаря ограничением, накладываемым на `switch` по сравнению с `if else`, он является более производительным. Чем больше условий, тем больше выигрыш от использования `switch` по сравнению с `if else`.

Проваливание в `switch`

Проваливание – это ситуация, когда после выполнения кода внутри одного `case` программа продолжает исполнять следующий `case` без дополнительной проверки условия. Такое происходит, если в конце блока, связанного с меткой `case`, отсутствует оператор `break`. Рассмотрим пример проваливания в `switch`. Эта программа похожа на предыдущую, но из неё удалены все инструкции `break`.

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

    switch (a)
    {
    case 1:
        printf("One\n");
    case 2:
        printf("Two\n");
    case 3:
        printf("Three\n");
    default:
        printf("Other\n");
    }
}
```

В этом случае, при достижение конца участка кода, соответствующего некоторому `case`, исполнение не будет выходить из `switch`, а вместо этого будет переходить к участку кода следующей метки. Поэтому, если этой программе передать на вход число 2, то она напечатает:

```
Two
Three
Other
```

Проваливание может возникать либо как ошибка (при пропуске `break`), либо как намеренный приём для группировки логики, как в примере ниже:

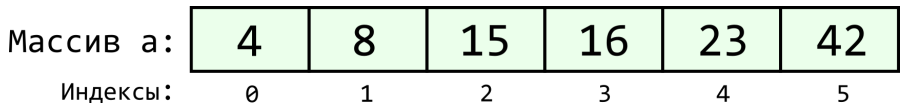
```
    int month;
    scanf("%i", &month);
    switch (month)
    {
    case 1: case 2: case 12:
        printf("Winter\n");
        break;
    case 3: case 4: case 5:
        printf("Spring\n");
        break;
    case 6: case 7: case 8:
        printf("Summer\n");
        break;
    case 9: case 10: case 11:
        printf("Autumn\n");
        break;
    default:
        printf("Error!\n");
        break;
    }
```

Массивы

Массив – это непрерывная область памяти, содержащая последовательность элементов одного типа. Размер массива (количество элементов) фиксируется при создании и не может быть изменён. Массив позволяет хранить любое фиксированное количество разных объектов одного типа и работать с ними.
Чтобы создать массив, который будет хранить 6 элементов типа `int` и сразу задать все значения элементов:

```
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;`

Инициализация массива

Объявить и инициализировать массив можно несколькими разными способами:

- Без инициализации. Такой массив обязательно нужно будет инициализировать в дальнейшем.

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

- Полная инициализация. Задаём все элементы.

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

- Неполная инициализация. В данном примере, мы задаём два элемента, а остальные автоматически зануляются. В результате массив `a` будет содержать следующие элементы: `1, 2, 0, 0, 0`.

  ```
  int a[5] = {1, 2};
  ```

- Инициализация нулями. Все элементы зануляются. Частный случай неполной инициализации.

  ```
  int a[10] = {0};
  ```

- Автоматическое определение размера, если он не указан. Размер этого массива будет равен пяти.

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

Границы массива

Выходить за границы массива опасно. Любой доступ за границы массива является очень грубой ошибкой – неопределённым поведением.

```
#include <stdio.h>
int main()
{
    int a[5] = {10, 20, 30, 40, 50};

    printf("%i\n", a[2]);   // OK, напечатает 30

    printf("%i\n", a[7]);   // Грубая ошибка - неопределённое поведение.
    printf("%i\n", a[-2]);  // Грубая ошибка - неопределённое поведение.
}
```

Наличие неопределённого поведения делает всё программу невалидной. При этом программа на вашем компьютере может не выдавать никаких сообщений об ошибке и даже делать что-то адекватное. Но это не значит, что ошибки в программе нет. Из-за того, что в программе присутствует неопределённое поведение, ошибки могут возникнуть на других системах или даже на вашей системе при следующем запуске.

Печать массива на экран

К сожалению в языке C нет стандартного способа распечатать массив на экран одной командой. Один вызов функции `printf` также не может распечатать весь массив сразу. Поэтому единственный вариант напечатать массив, это просто просто напечатать все его элементы, используя цикл, как это сделано в следующей программе:

```
#include <stdio.h>
int main()
{
    int a[5] = {10, 20, 30, 40, 50};

    for (int i = 0; i < 5; ++i)
        printf("%i ", a[i]);
    printf("\n");
}
```

Присваивание массива

К сожалению в языке C нет стандартного способа присваивания одного массива другому. Для присваивания также не работает синтаксис с фигурными скобочками, который работает для инициализации при объявлении. Поэтому единственный способ присвоить один массив другому – это поэлементное присваивание в цикле.

```
#include <stdio.h>
int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int b[5] = {90, 80, 70, 60, 50};

    // Ошибка, нельзя присваивать один массив другому
    a = b;

    // Ошибка, фигурные скобки для задания массива работают только при объявлении
    a = {11, 12, 13, 14, 15};

    // Единственный способ:
    for (int i = 0; i < 5; ++i)
        a[i] = b[i];
}
```

Считывание массива с экрана

Также как и для печати, в языке C нет одной стандартной функции, которая бы считывала весь массив сразу. Поэтому считывать нужно каждый элемент массива по отдельности в цикле. Пример программы, которая считывает 5 элементов и записывает из в массив, а затем печатает эти элементы, увеличенные на 1:

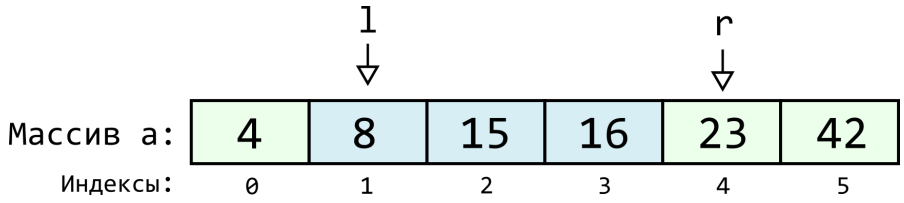
```
#include <stdio.h>
int main()
{
    int a[5];
    for (int i = 0; i < 5; ++i)
        scanf("%i", &a[i]);

    for (int i = 0; i < 5; ++i)
        printf("%i ", a[i] + 1);
    printf("\n");
}
```

Считывание n чисел с экрана в массив

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!");
}
```

Дополнительный материал

Объявление несколько переменных в одной строке

Возвращаемое значение функции `printf`

Возвращаемое значение функции `scanf`

Считывание последовательности чисел неизвестной длины

Независимость потоков `stdin` и `stdout`

Буферизация `scanf`

Особенность логических операторов

Тернарный оператор

Оператор запятая