Конструкция if

Зачастую нам необходимо выполнить не все инструкции в программе. Если некоторое условие истинно, то мы выполняем одни инструкции, иначе – другие. Для этого предназначена конструкция if, вот два примера её использования:

```
// выведет Hello!


if (42)    {
   printf("Hello!");
} else {
   printf("Not good");
}


// выведет Not good
if (42-42) {
   printf("Hello!");
} else {
   printf("Not good");
}
```

Выделяются три части:

- Условие – это выражение; в примерах выше это "42" и "42 - 42".
- Что делать если условие "верно" (ненулевое). Этот код называют *веткой then*.
- Что делать если условие "неверно" (равно нулю). Этот код называют *веткой else*.

Почему "ветки"? Потому что конструкция if подобна дереву с двумя ветвями: можно залезть на одну, а можно на другую, но не на обе сразу.

```
if (42)
//  ^^ условие
{                       // ветка then
   printf("Hello!");  // ветка then
```

```
}                       // ветка then
else
{                       // ветка else
   printf("Not good");// ветка else
}                       // ветка else
```

Шаблон для написания конструкции if выглядит так:

```
if ( <expr> )
{
   <branch-then>
}
else
{
   <branch-else>
}
```

Ветку else можно опустить, если мы не хотим ничего делать когда условие неверно.

```
if (42-42) {
   printf("Hello!");
}


// ничего не выведет, т.к. 42-42 == 0
```

Условия

Для кодирования логических условий нам необходимы операции сравнения и логические связки между ними, так же, как в математической логике. Начнём со сравнений, а связки между ними рассмотрим позднее.

В коде операции сравнения пишутся между двумя выражениями. Они сравнивают результаты подсчёта выражений и возвращают 0 или 1.

- Сравнение чисел с помощью >, <, <= (меньше или равно), >= (больше или равно)
  Значение `42 < 0` равно 0, значение `99 >= 99` равно 1.

- Проверки на равенство с помощью `==` (равно), `!=` (не равно)
  Значение `44 == 42 + 2` равно 1, значение 8 != 8 равно 0.

Распространённая ошибка. Нельзя сравнивать значения с помощью одинарного знака равенства `=`, можно только с помощью двойного `==`.

Последовательные проверки

Если написать несколько if'ов последовательно, то может показаться, что только один из них выполнится:

```
if (x < 5) {

    printf("less than five \n");

}

if (x > 0) {

    printf("greater than zero \n");

}
```

На самом деле программа по прежнему выполняется последовательно. Если в примере выше `x == 1`, то:

- сначала проверится первое условие: `1 < 5` верно, программа выведет `"less than five"`
- затем мы выполняем следующий statement: вторую конструкцию if.
- проверяем второе условие: `1 > 0` верно, поэтому программа выведет `"greater than zero"`.

Вложенные условия

Можно писать множество вложенных if'ов, например:

```
void not_zero(int x) {

   if (x > 0)

   {

      printf("yes");

   }

   else
```

```
    {
        if (x < 0)
        {
            printf("yes");
        }
        else
        {
            printf("no");
        }
    }
}
```

Составные условия

Порой нам необходимо описывать достаточно сложные условия, такие, как попадание числа в интервал. В следующем примере мы хотели бы вывести `Yes` если для z выполняется неравенство $3 < z < 7$, и `No` в противном случае.

```
void in_range_3_7(int z) {
    if ( /* z больше 3 и z меньше 7*/ ) {
        printf("Yes");
    }
    else {
        printf("No");
    }
}
```

Для кодирования составных логических условий нам необходимы не только операции сравнения, но и логические связки между ними, так же, как в математической логике.

Логические связки пишутся между двумя логическими значениями и возвращают 0 или 1.

- Логическое И `&&`. Вернет 1 если оба числа ненулевые.
  `x && y` равно 1 если `x != 0` и `y != 0`, иначе 0.
- Логическое ИЛИ `||`. Вернет 1 если хотя бы одно число ненулевое.
  `x || y` равно 0 если `x == 0` и `y == 0`, иначе 1.
- Логическое НЕ `!`
  `!x` равно 1 если `x == 0`, иначе 0

Помните, что логические связки можно писать и между числами. Например, `8 && 2 == 1` т.к. и 8 и 2 являются истинными значениями.

Теперь мы знаем, как закодировать условие в примере выше:

```
void in_range_3_7(int z) {

   if (z > 3 && z < 7) { // z больше 3 и z меньше 7

       printf("Yes");

   }

   else {

       printf("No");

   }

}
```

Когда в выражении несколько операторов, они выполняются в определённом порядке, в соответствии с приоритетами. Например, у умножения приоритет больше, чем у сложения, поэтому 1 + 2 * 3 равносильно 1 + (2*3) а не (1+2) * 3 . Полный список операторов с приоритетами достаточно велик, но для начала нам достаточно запомнить несколько простых правил:

1. Умножение и деление перед сложением и вычитанием.
2. && перед ||
3. Сравнения (<, >, >= и т.д.) перед && и ||

В любых хоть сколько-нибудь неочевидных выражениях расставляйте вручную столько скобок, сколько необходимо чтобы не ошибиться при его чтении.

Вернёмся к примеру со второго слайда.

```
void in_range_3_7(int z) {

   if (z > 3 && z < 7) { // z больше 3 и z меньше 7

       printf("Yes");
```

```
    }

    else {

        printf("No");

    }

}
```

Распространённая ошибка: в C не имеет смысла двойное неравенство. Например, выражение `3 < z < 7` всегда равно 1. Почему? Расставим в нём скобки: `(3 < z) < 7`. Рассмотрим два случая когда подвыражение `3 < z` равно или 0 или 1:

- для z < 3 всё выражение посчитается как 0 < 7, что всегда верно;
- для z >= 3 всё выражение посчитается как 1 < 7 , что тоже всегда верно.

Вернемся к изначальному примеру и перепишем его красивее: отделим логику принятия решений от вывода на экран.

```
int in_range_3_7(int z) {

    if (z > 3 && z < 7) { return 1; }

    else { return 0; }

}
```

Тогда чтобы проверить, верно ли $3 < 42 < 73<42<7$, мы можем написать:

```
if ( in_range_3_7(42) ) {

    printf("Yes");

}

else {

    printf("No");

}
```

Наконец, вспомним, что все операции сравнения и логические операции возвращают 0 или 1. Значит, значение выражения `z > 3 && z < 7` и так будет или 0 или 1. Можно вернуть прямо это значение:

```
int in_range_3_7(int z) {

    return z > 3 && z < 7;
```

```
}
```

Ошибка control reaches the end of non-void function

Рассмотрим такой пример. Можете ли вы предположить, в чём ошибка?

```
int f( int a ) {

   if ( a > 0 ) { return 1; }

}
```

У этой функции есть возвращаемое значение типа `int` и мы ожидаем, что как бы функция ни выполнилась, но мы обязательно выполним `return` с каким-то значением. Однако если `a <= 0`, мы не выполним ни одного `return`'а и совершенно непонятно, какое число эта функция должна вернуть.

Компилятор откажется работать с этой программой и выведет сообщение об ошибке. Вот сообщение об ошибке компилятора GCC:

```
control.c: In function ‘f’:

control.c:3:1: error: control reaches end of non-void function [-Werror=return-type]

    3 | }

      | ^
```

А вот что выведет компилятор clang:

```
control.c:3:1: error: non-void function does not return a value in all control paths [-Werror,-Wreturn-type]

}

^

1 error generated.
```

... но я же покрыл с помощью if'ов все возможные случаи!

Компилятор часто не может понять, что какой-то `return` обязательно будет выполнен. Например, вот так не получится:

```
int f (int x)  {

  if (x > 0) { return 1; }
```

```
  if (x <=0) { return 2; }
}
```

Нам кажется, что каким бы ни был `x`, один из if'ов выполнится. Но компилятор недостаточно умён, чтобы это понять, для него останется вопрос: "а что, если не выполнится ни первый, ни второй if?"

В общем случае для произвольных условий задачу "покрывают ли условия все возможные случаи" решить невозможно, даже теоретически (см алгоритмическую неразрешимость). Поэтому компилятор и не пытается.

Чтобы избежать ошибки, последнее условие обычно пишут внутри ветки `else`:

```
int f (int x)  {
  if (x > 0) { return 1; }
  else { return 2; }
}
```

В функциях, которые возвращают `void`, такой проблемы не возникает: они не обязаны ничего возвращать, поэтому если такая функция никогда не дойдёт до `return`, компилятор это не смутит.

```
void f (int x)  {
  if (x > 0) { printf("1"); }
  if (x <=0) { printf("2"); }
}
```

Упражнение. Перепишите код в задании так, чтобы он компилировался. В тестах вводятся два неотрицательные числа: первое проверяется с помощью `is_single_digit`, второе с помощью `is_double_digit`.

Sample Input:

```
1 2
```

Sample Output:

```
yes no
```

1task – program

Ленивость в логических условиях

Вспомним, как вычисляются конструкции `&&` и `||`. Представим, что мы действуем на выражения `x` и `y`, которые могут вычисляться к истинным или ложному значению; выражения `x` и `y`, над которыми производятся действия, называются *операндами*.

Таблица истинности для логических И и ИЛИ

x	y	x && y	x || y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Мы можем описать смысл этих операций так:

- логическое И **ложно** если **хотя бы один из операндов ложен**;
- логическое ИЛИ **истинно** если **хотя бы один из операндов истиннен.**

Значит, если первый операнд логического И равен нулю, то результат вычисления выражения уже известен. В выражении `0 && f()` нет смысла запускать функцию `f`: что бы она ни вернула, значение выражения всё равно будет равно 0. Аналогично и для выражения `1 || f()`.

```
int print1() { printf("Hello!"); return 1; }
```

```
...
```

```
0 || print1() ; // напечатает Hello!
```

```
1 || print1() ; // ничего не напечатает: print1 не был запущен
```

```
print1() || print1() ; // напечатает Hello! один раз; второй print1 не будет запущен.
```

```
int print0() { printf("Hello!"); return 0; }


...

1 && print0() ; // напечатает Hello!

0 && print0() ; // ничего не напечатает: print0 не был запущен



print0() && print0() ; // напечатает Hello! один раз; второй print0 не будет запущен.
```

Итак, операторы `&&` и `||` в C сделаны *ленивыми:* это значит, что они вычисляют ровно столько аргументов слева направо, сколько необходимо для получения итогового значения, а остальные игнорируются.

Заполните тело функции `is_sorted3` которая принимает 3 аргумента и возвращает 1 если они в возрастающем порядке, -1 если они в убывающем порядке и 0 в остальных случаях.

Не нужно писать полную программу, если об этом явно не сказано в задании. Если требуется написать функцию, то нужно написать только её (и можно добавлять вспомогательные функции); также не подключайте заголовочные файлы, даже если вы уже знаете, что это такое.

Про ошибку control reaches the end of non-void function.

Sample Input:

```
1 2 3
```

Sample Output:

```
1
```

2task.c – program

Заполните тело функции `max3` которая возвращает максимальный из своих аргументов.

Не нужно писать полную программу, если об этом явно не сказано в задании. Если требуется написать функцию, то нужно написать только её (и можно добавлять вспомогательные функции); также не подключайте заголовочные файлы, даже если вы уже знаете, что это такое.

[Про ошибку control reaches the end of non-void function.](#)

Sample Input:

```
1 2 3
```

Sample Output:

```
3
```

3task – program

Напишите функцию `fizzbuzz`, которая принимает один аргумент типа `int` и напечатает на экран в точности одну из следующих строчек:

- `fizz`, если аргумент делится на 3
- `buzz`, если аргумент делится на 5
- `fizzbuzz`, если аргумент делится и на 3 и на 5
- `no`, если аргумент меньше или равен нулю, вне зависимости от делимости.

Подсказка: чтобы проверить делимость числа. используйте операцию % (остаток от деления).

Не нужно писать полную программу, если об этом явно не сказано в задании. Если требуется написать функцию, то нужно написать только её (и можно добавлять вспомогательные функции); также не подключайте заголовочные файлы, даже если вы уже знаете, что это такое.

Sample Input:

```
3
```

Sample Output:

```
fizz
```

4task.c – program

Итоги

- Специального булевого типа данных для значений "истины" и "ложь", отличимого от целых чисел, в C нет.
- "Истинными" считаются любые значения кроме нуля.
- "Ложным" считается ноль.
- Выражения, которые считают логические значения, с помощью сравнений, булевых операций и т.д. при вычислении дают или число 0 (ложь), или число 1 (истина). Поэтому выражение (5>6) > 3 и подобные ему вычисляются неочевидно: 5>6 равно 0, а 0 не больше трёх.
- В конструкции `if (<expr>) { <branch1> } else { <branch2> }` ветку else можно опустить.

- Расставляйте скобки в сложных условиях.