

1.4. Принятие решений. Логические выражения.

Friday, March 20, 2020 23:19

Мы уже успели немного познакомиться с логическими выражениями, когда знакомились с циклами, — они используются в описании условия повторения цикла. В этом уроке мы рассмотрим логические выражения более глубоко и ознакомимся операторами ветвления:

- ♦ `if`
- ♦ `if-else`
- ♦ `else-if`
- ♦ `switch-case`
- ♦ тернарный оператор

Оператор `if`

В языке C для принятия решений используется оператор `if`. Синтаксис выглядит следующим образом:

```
1 if ( logic_expression )
2     program statement
```

Например, можно выводить сообщения при выполнении какого-то условия. Строки

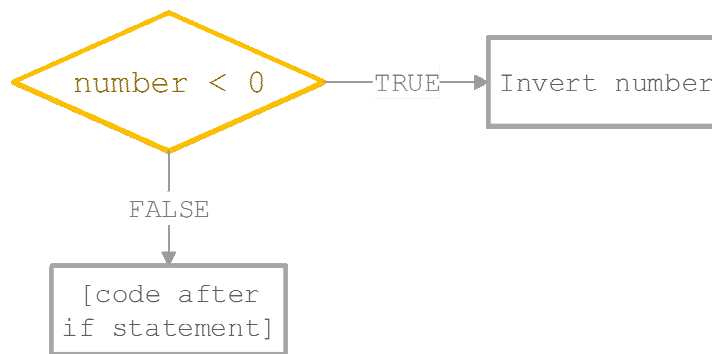
```
1 if ( count > COUNT_LIMIT )
2     printf ( "Count limit exceeded\n" );
```

выведут сообщение `Count limit exceeded` только когда значение переменной `count` будет больше значения переменной `COUNT_LIMIT`.

Напишем программу, выводящую число по модулю.

```
1 // Program to calculate the absolute value of an integer
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     int number;
8
9     printf ( "Enter your number: " );
10    scanf ( "%i", &number );
11
12    if ( number < 0 )
13        number = -number;
14
15    printf ( "The absolute value is %i\n", number );
16
17    return 0;
18 }
```

Мы сравниваем число с нулем, и если оно меньше нуля, то инвертируем его знак унарным минусом. Также составим логическую диаграмму, чтобы лучше понимать происходящее и учиться работать с такими диаграммами:



Напишем еще одну программу; которая подсчитает средний балл и покажет количество непроходных баллов.

```

1  /* Program to calculate the average of a set of grades
2   and count the number of failing test grades          */
3
4  #include <stdio.h>
5
6  int main (void)
7  {
8      int    numberOfGrades, passingGrade, i, grade;
9      int    gradeTotal = 0;
10     int    failureCount = 0;
11     float  average;
12
13     printf ("How many grades will be entered? ");
14     scanf ("%i", &numberOfGrades);
15     printf ("Enter the minimum passing grade: ");
16     scanf ("%i", &passingGrade);
17     printf ("\n");
18
19     for (i = 1; i <= numberOfGrades; i++) {
20         printf ("Enter grade #%i: ", i);
21         scanf ("%i", &grade);
22         gradeTotal += grade;
23
24         if (grade < passingGrade)
25             failureCount++;
26     }
27
28     average = (float) gradeTotal / numberOfGrades;
29
30     printf ("\nAverage grade: %.2f\n", average);
31     printf ("Failure count: %i\n", failureCount);
32
33     return 0;
34 }
  
```

Здесь при вычислении среднего значения пришлось снова вспомнить об отсечении чисел после запятой при делении целочисленных переменных. Поэтому для получения правильного результата нужно дописать (float) перед gradeTotal. Параметр %.2f выводит число в формате 00.00, потому что при подсчете оценок нам обычно не нужно больше чисел после запятой.

Будет довольно интересный результат, если ввести 0 при запросе количества оценок. Что при этом произойдет, будет зависеть от того, как система обработает деление на ноль. Windows 8.1 x64 возвращает NaN (Not a Number).

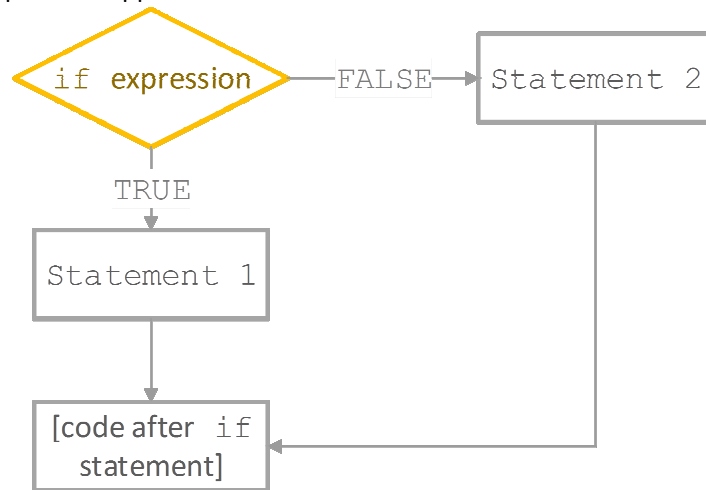
Конструкция if-else

Бывает, что нужно выполнить какое-то действие не только когда условие выполняется, но и когда оно не

выполняется. Да, можно дважды использовать `if`, но в таком коде будет сложнее разобраться. Гораздо удобнее использовать конструкцию `if-else`. Синтаксис выглядит следующим образом:

```
1 if ( expression )
2     program statement 1
3 else
4     program statement 2
```

То есть если выражение возвращает `TRUE`, будет выполнено `statement 1`, а если `FALSE`, то `statement 2`. Логическая диаграмма будет выглядеть так:



Напишем программу, которая будет проверять, четность числа.

```
1 // Program to determine if a number is even or odd
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     int number, remainder;
8
9     printf ("Enter your number: ");
10    scanf ("%i", &number);
11
12    remainder = number % 2;
13
14    if (remainder == 0)
15        printf ("The number is even.\n");
16    else
17        printf ("The number is odd.\n");
18
19    return 0;
20 }
```

Не забывать о том, что `=` это присваивание, а `==` логический оператор сравнения!

Сложные логические выражения (Compound Relational Tests)

В предыдущих примерах мы всегда пользовались простыми логическими выражениями из одного неравенства. Давайте рассмотрим более сложные выражения с операторами булевой алгебры. Например, выражение

```
if ( grade > 70 && grade < 79 )
```

использует логический оператор `AND` и данное выражение сработает только когда переменная `grade` больше 70 **И** меньше 79. А выражение

```
if ( index < 0 || index > 99 )
```

использует логический оператор `OR` и данное выражение сработает когда переменная `index`

меньше 0 **ИЛИ** больше чем 99.

Можно составлять и более сложные выражения, но не рекомендуется их делать слишком длинными, чтобы код было удобнее читать.

Напишем программу, которая будет определять, високосный введенный год или нет. Год должен делиться на 4, но если он делится и на 100, то нужно дополнительно проверить, делится ли он на 400. Составим логическое уравнение:

```
if (div_4 == 0 && div_100 != 0 || div_400 == 0)
```

То есть, если год делится на 4 и делится на 100, операция AND выдаст FALSE, т.к. если год делится на 100, то неравенство `div_100 != 0` выдаст FALSE. Но, если год делится на 400, операция OR выдаст TRUE и условие выполнится.

Напишем программу целиком:

```
1 // Program to determine if a year is a leap year
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     int year, div_4, div_100, div_400;
8
9     printf ("Enter the year: ");
10    scanf ("%i", &year);
11
12    div_4    = year % 4;
13    div_100  = year % 100;
14    div_400  = year % 400;
15
16    if ((div_4 == 0 && div_100 != 0) || div_400 == 0)
17        printf ("This is a leap year.\n");
18    else
19        printf ("This is not a leap year.\n");
20
21    return 0;
22 }
```

Кстати, компилятору не понравилось, что `div_4 == 0 && div_100 != 0` не было взято в скобки и он мне посоветовал их поставить. Не знаю почему, но на всякий случай скобки поставил. В интернете пишут, типа компилятор может не понять, в каком порядке производить логические операции, `(a && b) || c` или `a && (b || c)`. Очень странно, но ладно.

Логическая диаграмма:

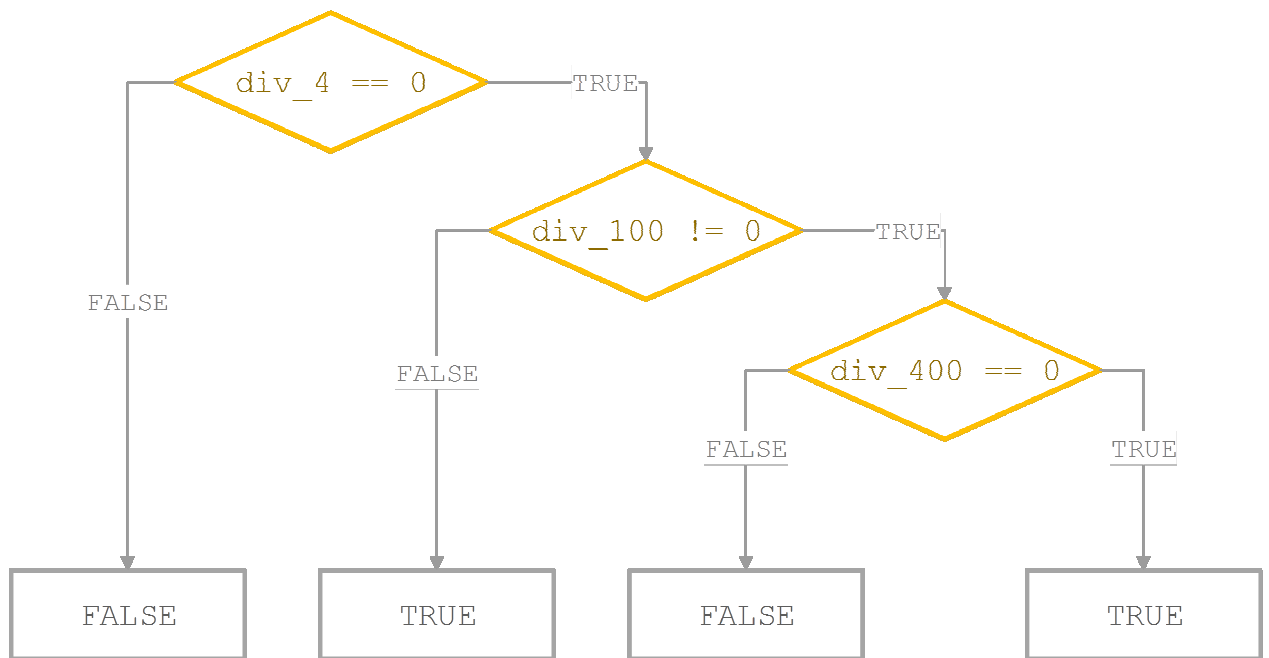


Таблица истинности:

div_4 == 0	div_100 != 0	div_400 == 0	(a && b) c
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Вложенные конструкции if

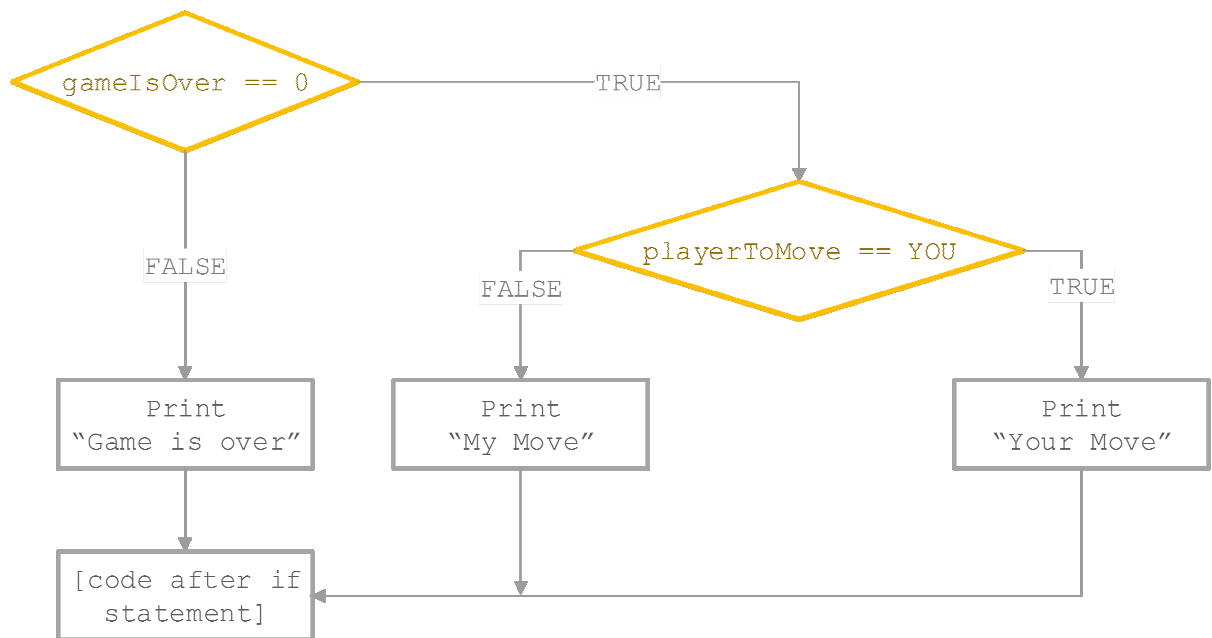
Нам ничто не мешает каскадировать операторы if и else. Например, код

```

1 if ( gameIsOver == 0 )
2     if ( playerToMove == YOU )
3         printf ( "Your Move\n" );
4     else
5         printf ( "My Move\n" );
6 else
7     printf ( "The game is over\n" );
  
```

будет проверять переменную `playerToMove` только если переменная `gameIsOver` равна нулю. И после проверки будут выведена строка в зависимости от значения `playerToMove`.

Диаграмма будет выглядеть так:



...надоело составлять диаграммы, хватит.

Конструкция `else if`

Эту конструкцию удобно использовать, когда нужно принять небинарное решение (четное / нечетное число, високосный / не високосный год), а множественное решение. Например, рассмотрим программу, которая на вход получает число и должна вывести, больше ли оно или меньше нуля, или равно ему (по-нормальному это называется функцией определения знака числа). Конструкция `if-else` здесь не будет работать. Да, можно использовать три отдельных оператора `if`, но такая конструкция далеко не всегда работает, особенно когда проверяемые условия не взаимоисключаемые.

Лучше всего использовать конструкцию `else if`, — дописав оператор `if` после `else`:

```

1 if ( expression 1 )
2     program statement 1
3 else
4     if ( expression 2 )
5         program statement 2
6     else
7         program statement 3
  
```

которая расширяет количество возможных исходов с двух до трех. Можно еще добавлять операторы `if` после операторов `else`, если нужно. Такая конструкция так часто используется, что ее форматируют иначе, чтоб сразу можно было увидеть, что здесь принимаются множественные решения:

```

1 if ( expression 1 )
2     program statement 1
3 else if ( expression 2 )
4     program statement 2
5 else
6     program statement 3
  
```

Напишем программу для определения знака числа, о которой мы говорили в начале главы, используя конструкцию `else if`.

```

1 // Program to implement the sign function
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     int number;
  
```

```

8
9     printf ("Enter your number: ");
10    scanf ("%i", &number);
11
12    if (number < 0)
13        printf ("It's a negative number.\n");
14    else if (number == 0)
15        printf ("It's a zero.\n");
16    else
17        printf ("It's a positive number.\n");
18
19    return 0;
20 }

```

Напишем что-нибудь покруче. Например, программу, определяющую тип символа: буква алфавита, цифра или специальный символ.

```

1 // Program to categorize a single character
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     char c;
8
9     printf ("Enter your character: ");
10    scanf ("%c", &c);
11
12    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
13        printf ("It's an alphabetic character.\n");
14    else if (c >= '0' && c <= '9')
15        printf ("It's a digit.\n");
16    else
17        printf ("It's a special character.\n");
18
19    return 0;
20 }

```

Такую программу можно использовать, например, для реализации ввода математических выражений с клавиатуры: можно распознать, где цифры, где математические операторы и где переменные.

Но лучше помнить, что такое распознавание символов нацелено на работу только с кодировкой ASCII. На практике, для надежной работы в любой кодировке, лучше использовать операторы `islower()` и `isupper()` из библиотеки `ctype.h`.

Также обратите внимание, что при сравнении цифр мы сравниваем не целочисленные 0..9, а символы '0'..'9'. Это большая разница с точки зрения их цифрового представления в программе.

Раз уж мы упомянули реализацию ввода математических выражений, напомним-ка программу, которая будет рассчитывать простейшие выражения из двух чисел и одного оператора.

```

1 // Program to evaluate simple expressions
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     float value1, value2;
8     char operator;
9
10    printf ("Enter your expression.\n");
11    scanf ("%f %c %f", &value1, &operator, &value2);
12
13    if (operator == '+')
14        printf (".2f\n", value1 + value2);

```

```

15     else if (operator == '-')
16         printf (".2f\n", value1 - value2);
17     else if (operator == '*')
18         printf (".2f\n", value1 * value2);
19     else if (operator == '/')
20         if (value2 == 0)
21             printf ("Division by zero.\n");
22         else
23             printf ("%g\n", value1 / value2);
24     else
25         printf ("Invalid expression.\n");
26
27     return 0;
28 }

```

Операторы `%f` `%c` `%f` в функции `scanf()` написаны через пробел. Это допускает ввод одного пробела между числами и оператором. Именно что допускает, а не делает обязательным. Ввод `2+2` будет работать так же правильно, как и `2 + 2`. Я здесь еще немного забежал наперед и добавил защиту от ввода неправильного оператора и от деления на ноль. Это хорошая практика, когда программист предусматривает возможные ошибочные состояния программы и реализует их обработку.

Оператор `switch`

Мы уже пользовались конструкцией `else if`, при помощи которой мы по очереди сравнивали переменную с несколькими значениями. Специально для этого был создан оператор `switch`. Синтаксис выглядит следующим образом:

```

1  switch ( expression )
2  {
3      case value1:
4          program statement
5              ...
6          break;
7      case value2:
8          program statement
9              ...
10         break;
11         ...
12     case valueN:
13         program statement
14             ...
15         break;
16     default:
17         program statement
18             ...
19         break;
20 }

```

Здесь оператор `default` срабатывает только если ни один из операторов `case` не сработал. По сути как последний `else` в конструкции `else if`.

Давайте переделаем нашу программу для расчета выражений, используя оператор `switch`.

```

1  // Program to evaluate simple expressions
2
3  #include <stdio.h>
4
5  int main (void)
6  {
7      float value1, value2;
8      char operator;
9
10     printf ("Enter your expression.\n");
11     scanf ("%f %c %f", &value1, &operator, &value2);

```



```

12
13     switch (operator)
14     {
15     case '+':
16         printf (".2f\n", value1 + value2);
17         break;
18     case '-':
19         printf (".2f\n", value1 - value2);
20         break;
21     case '*':
22     case 'x':
23         printf (".2f\n", value1 * value2);
24     case '/':
25         if (value2 == 0)
26             printf ("Division by zero.\n");
27         else
28             printf ("%g\n", value1 / value2);
29         break;
30     default:
31         printf ("Invalid expression.\n");
32         break;
33     }
34
35     return 0;
36 }

```

Обратите внимание, что теперь умножение выполняется не только при помощи оператора *, но и при помощи оператора x. То есть можно задавать одну и ту же функцию для разных значений переменной, если просто их писать друг за другом в строчку или в столбик.

Логический тип переменных (Boolean Variables)

Рассмотрим задачу составления списка простых чисел, т.е. таких, которые делятся только на 1 и сами на себя. Существует множество решений этой задачи, но мы рассмотрим самое простое решение в лоб: для каждого числа p проверять делимость на числа в диапазоне от $d = 2$ до $p-1$.

```

1 // Program to generate a table of prime numbers
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     int p, d;
8     _Bool isPrime;
9
10    for (p = 2; p <= 50; p++) {
11        isPrime = 1;
12
13        for (d = 2; d < p; d++)
14            if (p % d == 0)
15                isPrime = 0;
16
17        if (isPrime != 0)
18            printf("%i ", p);
19    }
20
21    printf ("\n");
22    return 0;
23 }

```

Здесь сделаны два вложенных цикла, первый цикл выбирает число p и задает `isPrime = TRUE`. Второй цикл проверяет, делится ли число p на числа, которые меньше него. Если находится число, которое на него делится нацело, то флаг сбрасывается: `isPrime = FALSE`. Если флаг был сброшен, то мы не

выводим число `p`, при проверке которого флаг `isPrime` был сброшен.

Давайте теперь вспомним урок 1.2, где мы говорили о том, что значение переменной считается `TRUE`, если она ненулевая. То есть нам ничто не мешает написать, например

```
1 if (100)
2     printf ("This will always be printed.\n");
```

и оператор `printf()` всегда будет выполняться. То есть можно опустить оператор сравнения и писать просто

```
if (isPrime)
```

что будет эквивалентно

```
if (isPrime != 0)
```

Если надо проверить, равна ли переменная нулю (`FALSE`), можно использовать логический оператор `NOT`

```
if (! isPrime)
```

который инвертирует значение переменной (она станет 0, если была ненулевая и станет 1, если была 0).

Еще один вариант использования оператора `NOT`:

```
myMove = !myMove;
```

Его также можно применять к выражениям:

```
! (x < y)
```

На самом деле это выражение можно было записать как `x >= y`, но я не придумал лучшего примера.

Также из урока 1.2 вспомним, что тип переменной `bool` со значениями `true` и `false` можно подключить из библиотеки `stdbool.h`. Перепишем нашу предыдущую программу с использованием этого типа переменной.

```
// Program to generate a table of prime numbers

#include <stdio.h>
#include <stdbool.h>

int main (void)
{
    int p, d;
    bool isPrime;

    for (p = 2; p <= 50; p++) {
        isPrime = true;

        for (d = 2; d < p; d++)
            if (p % d == 0)
                isPrime = false;

        if (isPrime != false)
            printf ("%i ", p);
    }

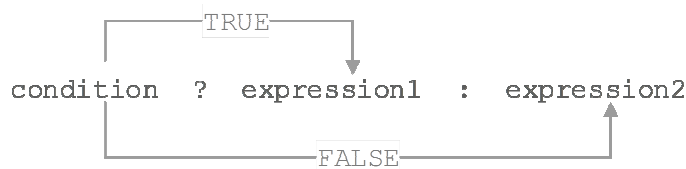
    return 0;
}
```

Всё работает. Да, это просто косметические изменения, смысл никак не изменился, просто стало проще читать код.

Тернарный условный оператор (Ternary conditional operator)

Довольно странная штука. Называется тернарным (третичным), потому что имеет три операнда, разделенные знаком вопроса (`?`) и двоеточием (`:`). Синтаксис и логика работы выглядит следующим

образом:



То есть, если условие (`condition`) выполняется, то будет выполнено первое выражение (`expression1`), а если не выполняется, то второе выражение (`expression2`). Не знаю, какой садист это придумал, но зачем-то это существует. Аналогичная конструкция через `if-else`:

```
1 if (condition)
2     expression1
3 else
4     expression2
```

Тернарный оператор чаще всего используется, когда нужно присвоить переменной два разных значения в зависимости от некоторого условия. Представим, что у нас есть целочисленные переменные `x` и `s` и нужно присвоить переменной `s` значение `-1`, если `x < 0`, а во всех остальных случаях присвоить ей `x2`. Реализация через тернарный оператор будет выглядеть вот так:

```
s = (x < 0) ? -1 : x * x;
```

Здесь `x < 0` взято в скобки просто для надежности, в простых выражениях не обязательно их использовать.

Приведем еще один пример, где переменной `maxValue` присваивается значение большей из переменных `a` и `b`:

```
maxValue = (a > b) ? a : b;
```

Как и с конструкциями `else if`, ничто не мешает каскадировать тернарные операторы:

```
sign = ( number < 0 ) ? -1 : ( ( number == 0 ) ? 0 : 1 );
```

Здесь, если `number < 0`, то `sign = -1`, а если `number = 0`, то `sign = 0`, во всех остальных случаях `sign = 1`. Кстати, это реализация определения знака переменной. Одной короткой строкой. Охренеть.

Если что, можно без скобок. Операторы обрабатываются слева направо. То есть выражение

```
e1 ? e2 : e3 ? e4 : e5
```

будет обработано как

```
e1 ? e2 : ( e3 ? e4 : e5 )
```

Не знаю, кому я это пишу, правда, лично мне всегда удобнее лишний раз скобки поставить. Ну, чужой код смогу правильно прочесть по крайней мере.

Также не обязательно чтоб тернарный оператор был частью функции присваивания какой-то переменной. Ничто не мешает его использовать, например, в функции `printf` ()

```
printf ("Sign = %i\n", ( number < 0 ) ? -1 : ( number == 0 ) ? 0 : 1 );
```

Здесь нам не пришлось использовать переменную `sign` для хранения знака числа, а мы сразу вывели его в строку.

Упражнения

2. Здесь достаточно простейшей конструкции `else if`.

```
/* Program to test if number is evenly
   divisible by other number */
#include <stdio.h>

int main (void)
{
    int num1 = 0, num2 = 0;

    printf ("Enter your numbers, divided by space: ");
    scanf ("%i %i", &num1, &num2);

    if (num2 == 0)
        printf ("Can't divide by zero.\n");
    else if (num1 % num2 == 0)
        printf ("%i is evenly divisible by %i.\n", num1, num2);
    else
        printf ("%i isn't evenly divisible by %i.\n", num1, num2);

    return 0;
}
```

3. Непонятно что тут делает это упражнение, как-то слишком просто.

```
// Program to divide two integers

#include <stdio.h>

int main (void)
{
    int num1, num2;

    printf ("Enter your numbers, divided by space: ");
    scanf ("%i %i", &num1, &num2);

    if (num2 == 0)
        printf ("Can't divide by zero.\n");
    else
        printf ("%i / %i = %.3f\n", num1, num2,
            (float) num1 / num2);

    return 0;
}
```

4. Не так уж и сложно, как показалось на первый взгляд. Но громоздко.

```
1 // "Logging" calculator
2
3 #include <stdio.h>
4 #include <stdbool.h>
5
6 int main (void)
7 {
8     bool run = true;
9     float num, acc = 0;
10    char oper;
```

```

11
12     printf ("Begin Calculations\n");
13
14     do {
15
16         scanf ("%f %c", &num, &oper);
17
18         switch (oper) {
19
20             case 'S': case 's':
21                 acc = num;
22                 break;
23
24             case '+':
25                 acc += num;
26                 break;
27
28             case '-':
29                 acc -= num;
30                 break;
31
32             case '*':
33                 acc *= num;
34                 break;
35
36             case '/':
37                 if (num == 0)
38                     printf("Division by zero.\n");
39                 else
40                     acc /= num;
41                 break;
42
43             case 'E': case 'e':
44                 run = false;
45                 break;
46
47             default:
48                 printf("Invalid expression.\n");
49                 break;
50             }
51
52         printf("= %g\n", acc);
53     }
54     while(run);
55
56     printf("End of Calculations.\n");
57
58     return 0;
59 }

```

Исходник:



main

5. Вернемся к упражнению 10 урока 1.3. Очевидно, что нужно проверить знак числа, и если оно отрицательное, вывести минус, инвертировать знак числа и переворачивать как раньше.

```

1 // Reverse number of any sign
2
3 #include <stdio.h>
4
5 int main (void)
6 {

```

```

7         int num;
8
9         printf ("Enter your number: ");
10        scanf ("%i", &num);
11
12        if (num < 0) {
13            num = -num;
14            printf("-");
15        }
16
17        do {
18            printf("%i", num % 10);
19            num /= 10;
20        }
21        while (num != 0);
22
23        return 0;
24    }

```

6. Эту задачу явно нужно решать через массивы, но я "типа" не умею с ними еще работать. Забавно. Потратив чуть больше времени чем было нужно, я решил сделать следующим образом:
- считаем количество нулей в конце числа, ибо основной алгоритм не будет способен их распознать
 - отрезаем от числа цифры, пока не останется одна цифра (число < 10)
 - с каждой отрезаемой цифрой домножаем переменную множителя на 10
 - оставшуюся от числа цифру выводим текстом в консоль, домножаем на множитель и вычитаем ее из изначального числа
 - дописываем столько нулей, сколько насчитали в начале

То есть, например, у числа 5600 мы насчитаем два нуля, а после первой итерации его обработки останется цифра 5 и множитель будет равен 1000.

И тогда в консоль будет выведено `five`, и от числа 5600 будет вычтено $5 \times 1000 = 5000$.

Таким образом мы выбросим первую цифру исходного числа. Останется 600, которое отправится на повторную обработку. Затем выбросим цифру 6. После этого цикл завершится, потому что останутся только нули, которые будут дописаны сразу после завершения цикла ровно столько раз, сколько мы насчитали в начале.

Если же была введена одна цифра, она будет выброшена все равно, т.к. стандартное значение множителя равно 1 и введенное число будет вычтено само из себя, что даст 0, который и является условием завершения программы.

```

1 //  Number to text conversion
2
3 #include <stdio.h>
4
5 int main (void)
6 {
7     int num, buffer, multiplier, zerocount = 0;
8
9     printf ("Enter your number: ");
10    scanf ("%i", &num);
11
12    buffer = num;
13
14    if (buffer != 0) // can't divide by zero
15        while (buffer % 10 == 0) { // count trailing zeros
16            buffer /= 10;
17            zerocount ++;
18        }
19
20    do {
21        multiplier = 1;

```

```

22         buffer = num;
23                                     // count number of digits
24         while (buffer > 9) {         // for each iteration
25             buffer /= 10;
26             multiplier *= 10;
27         }
28
29         num -= buffer * multiplier;
30
31         switch (buffer) {
32         case 0:
33             printf("zero ");
34             break;
35         case 1:
36             printf("one ");
37             break;
38         case 2:
39             printf("two ");
40             break;
41         case 3:
42             printf("three ");
43             break;
44         case 4:
45             printf("four ");
46             break;
47         case 5:
48             printf("five ");
49             break;
50         case 6:
51             printf("six ");
52             break;
53         case 7:
54             printf("seven ");
55             break;
56         case 8:
57             printf("eight ");
58             break;
59         case 9:
60             printf("nine ");
61             break;
62         }
63     }
64     while(num != 0);
65
66     for ( ; zerocount != 0; zerocount--)
67         printf("zero ");           // print trailing zeros
68
69     printf("\n");
70
71     return 0;
72 }

```

Исходник:



main

P.S. Анон дал мне задание вывести число по-нормальному, как в синтезаторах речи. Сделал без особого труда. Единственное что, программа поддерживает числа только до 9999. Дальше было лень делать.

Заодно научился работать в дебаггере, потому что долго не понимал, почему цикл

`while (buffer > 10)` выходит из цикла при `buffer = 10` (подсказка: 10 не больше 10).

Исходник:



main

7. Даже не знаю, как прокомментировать. Просто задача на операторы ветвления. Постарался задействовать максимум выученного.

```
1 // Quicker prime number list generator
2
3 #include <stdio.h>
4 #include <stdbool.h>
5
6 int n, p;
7 bool isPrime;
8
9 int main (void)
10 {
11     for (n = 2; n <= 50; n++) {
12         if (n != 2 && (n % 2 == 0))
13             continue;
14
15         isPrime = false;
16
17         for (p = 2; p < n; p++) {
18
19             if (p != 2 && (p % 2 == 0))
20                 continue;
21             else if (n % p == 0)
22                 isPrime = true;
23         }
24
25         if (!isPrime)
26             printf("%i ", n);
27     }
28
29     return 0;
30 }
```

Еще более оптимальный вариант: зачем проверять делимость на 2, когда можно вообще исключить возможность появления четных чисел.

```
1 // Quicker prime number list generator v1.1
2
3 #include <stdio.h>
4 #include <stdbool.h>
5
6 int n, p;
7 bool isPrime;
8
9 int main (void)
10 {
11     printf("2 ");
12
13     for (n = 3; n <= 50; n += 2) {
14         isPrime = true;
15
16         for (p = 3; p < n; p += 2)
17             if (n % p == 0)
18                 isPrime = false;
19
20         if (isPrime)
```



```
21         printf("%i ", n);  
22     }  
23  
24     return 0;  
25 }
```

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 5
2. <https://sites.google.com/a/jcu.edu/cs128fall2012/daily-lessons/10-26/boolean-logic-example-leap-year>
3. <https://ci-plus-plus-snachala.ru/?p=85>