

5

Making statements

This chapter demonstrates how statements can evaluate expressions to determine the direction in which a C program should proceed.

- 72** Testing expressions
- 74** Branching switches
- 76** Looping for a number
- 78** Looping while true
- 80** Breaking out of loops
- 82** Going to labels
- 84** Summary

Testing expressions

The **if** keyword is used to perform the basic conditional test that evaluates a given expression for a boolean value of true or false. Statements within braces following the evaluation will only be executed when the expression is found to be true. So the syntax of an **if** test statement looks like this:

```
if ( test-expression ) { statements-to-execute-when-true }
```

There may be multiple statements to be executed when the test is true but each statement must be terminated by a semi-colon.

Sometimes it may be desirable to evaluate multiple expressions to determine whether following statements should be executed and this can be achieved in two ways. The logical **&&** AND operator can be used to ensure the statements will only be executed when two expressions are both found to be true, with this syntax:

```
if ( ( test-expression ) && ( test-expression ) ) { statements-to-execute }
```

Alternatively multiple **if** statements can be “nested”, one inside another, to ensure following statements will only be executed when both evaluated expressions are found to be true, like this:

```
if ( test-expression )
{
    if ( test-expression ) { statements-to-execute }
}
```

When one or more expressions evaluated by an **if** test statement are found to be false the statements within its following braces are not executed and the program proceeds to subsequent code.

It is often preferable to extend an **if** statement by appending an **else** statement specifying statements within braces to be executed when the expressions evaluated by the **if** statement are found to be false, with this syntax:

```
if ( test-expression )
    { statements-to-execute-when-true }
else
    { statements-to-execute-when-false }
```

This is a fundamental programming technique that offers the program two directions in which to proceed, depending on the result of the evaluation, and is known as “conditional branching”.

...cont'd

- 1 Begin a new program with a preprocessor instruction to include the standard input/output library functions **#include <stdio.h>**
- 2 Add a main function that outputs a statement when a tested expression is found to be true

```
int main()
{
    if( 5 > 1 ) { printf( "Yes, 5 is greater than 1\n" ) ; }
}
```
- 3 Next in the main function block, output a statement when two tested expressions are both found to be true

```
if( 5 > 1 )
{
    if( 7 > 2 )
        { printf("5 is greater than 1 and 7 is greater than 2\n") ; }
}
```
- 4 Now output a default statement after two tested expressions are both found to be false

```
if( 1 > 2 )
{ printf( "1st Expression is true\n" ) ; }
else if( 1 > 3 )
{ printf( "2nd expression is true\n" ) ; }
else
{ printf( "Both expressions are false\n" ) ; }
```
- 5 At the end of the main function block return a zero integer value as required by the function declaration **return 0 ;**
- 6 Save the program file then compile and execute the program to see output following conditional tests



ifelse.c

```

C:\MyPrograms>gcc ifelse.c -o ifelse.exe
C:\MyPrograms>ifelse
Yes, 5 is greater than 1
5 is greater than 1 and 7 is greater than 2
Both expressions are false
C:\MyPrograms>_

```

Don't forget



In C boolean true is represented numerically as 1 and boolean false as 0 (zero). So the expression **(5 > 1)** is shorthand for **(5 > 1 == 1)**.

Hot tip



When the code to be executed is just a single statement the braces may optionally be omitted.

Hot tip



Multiple expressions can be evaluated by combining **if** and **else** statements with **if() { ... } else if () { ... }**

Branching switches

Conditional branching performed by multiple **if else** statements can often be performed more efficiently by a **switch** statement when the test expression just evaluates a single condition.

Hot tip



It's always a good idea to include a **default** statement – even if it's only to output an error message when the **switch** statement fails.

The **switch** statement works in an unusual way. It takes a given value as its parameter argument then seeks to match that value from a number of **case** statements. Code to be executed when a match is found is included in each **case** statement.

It is important to end each **case** statement with a **break** keyword statement so the **switch** statement will then exit when a match is found without seeking further matches – unless that is the deliberate requirement.

Optionally the list of **case** statements can be followed by a single final **default** statement to specify code to be executed in the event that no matches are found within any of the **case** statements. So the syntax of a switch statement typically looks like this:

```
switch ( test-value )
{
    case match-value : statements-to-execute-when-matched ; break ;
    case match-value : statements-to-execute-when-matched ; break ;
    case match-value : statements-to-execute-when-matched ; break ;
    default : statements-to-execute-when-no-match-found ;
}
```

A **switch** statement can have up to at least 257 case statements according to the ANSI C standard, but no two of its **case** statements can attempt to match the same value.

Where a number of match-values are to each execute the same statements only the final **case** statement need include the statements to be executed and the **break** statement to exit the **switch** statement block. For example, to output the same message for match-values of 0, 1, and 2:

```
switch ( num )
{
    case 0 :
    case 1 :
    case 2 : printf( "Less than 3\n" ) ; break ;
    case 3 : printf( "Exactly 3\n" ) ; break ;
    default : printf( "Greater than 3 or less than zero\n" ) ;
}
```

Don't forget



A **default** statement need not appear at the end of the **switch** block but it is logical to place it there where it needs no **break** statement.

...cont'd

- 1 Begin a new program with a preprocessor instruction to include the standard input/output library functions **#include <stdio.h>**
- 2 Add a main function that declares and initializes one integer variable and one character variable


```
int main()
{
    int num = 2 ; char letter = 'b' ; }
}
```
- 3 Next in the main function block, insert a switch statement that attempts to match the integer value


```
switch( num )
{
    case 1 : printf( "Number is one\n" ) ; break ;
    case 2 : printf( "Number is two\n" ) ; break ;
    case 3 : printf( "Number is three\n" ) ; break ;
    default : printf( "Number is unrecognized\n" ) ;
}
```
- 4 Now insert a switch statement that attempts to match the character value


```
switch( letter )
{
    case 'a' : case 'b' : case 'c' :
    printf( "Letter is %c\n" , letter ) ; break ;
    default : printf( "Letter is unrecognized\n" ) ;
}
```
- 5 At the end of the main function block return a zero integer value as required by the function declaration **return 0 ;**
- 6 Save the program file then compile and execute the program to see output from the switch statements



switch.c

Hot tip



In **switch** statements the **case** keyword, match-value, and colon character are regarded as a unique "label".

```
Command Prompt
C:\MyPrograms>gcc switch.c -o switch.exe
C:\MyPrograms>switch
Number is two
Letter is b
C:\MyPrograms>_
```