Making Decisions - Chapter Summary

1 Looping in C

In C, the flow of control moves sequentially through the statements of a program. The first statement is executed, then the second, then each statement in turn, until all statements have been executed.

Many programming tasks, however, require parts of a program to be repeated. This can be achieved by using a loop, which will cause the required series of statements to be executed repeatedly, usually until a predetermined condition is met. The loop then ends, and the statements after the loop are executed in the normal manner.

The number of times the statements in a loop are executed is referred to as the number of iterations of the loop.

2 The `while` Loop

The while loop has the general form:

```
while (expression)
     loopbody
```

The `while` loop is an entry-condition loop. First, the expression is evaluated as either true (i.e. non-zero) or false (i.e. zero). This determines whether or not the statements in the loopbody are executed.

Note that if the expression is false, the loopbody is never executed.

If the expression is true, the loopbody is executed, then the expression is evaluated again. If it is still true, the loopbody is executed once more.

This process of evaluation of the expression followed by execution of the loopbody continues until the expression evaluates to false.

Simple and Compound Statements

The following program contains two `while` loops. The first has a loopbody that consists of a simple statement, and the second has a loopbody that consists of a compound statement:

```
main()
{
     int shrink = 100, grow =0;

     while (shrink > 50)
          printf("%d is a shrinking integer!\n", shrink--);

     while (shrink > grow)
     {
          printf("%d is bigger than %d.\n", shrink, grow);
          shrink--;
          grow++;
     }
     printf("%d and %d are the same size!\n", shrink, grow);
}
```

The statement that forms the loopbody in the first `while` loop ends with a statement terminator `;`. Note that the statement is indented on a new line for clarity, so that it stands out from the `while` and its accompanying expression.

In the second `while` loop, the start and end of the loopbody are marked by opening and closing braces, `{` and `}`. The individual statements within the loopbody must still end in semicolons, however, and are once again indented for clarity.

3 The do while Loop

The general form of the do while loop is:

```
do
     loopbody
while (expression);      /* Note the ; */
```

The `do while` loop is an exit-condition loop. First, the loopbody is executed. It is only after all the statements in the loopbody have been executed once that the test expression in the final line of the loop is evaluated.

If the expression evaluates to false (i.e. zero), control is passed to the next statement in the program.

If the test expression evaluates to true (i.e. non-zero), the loopbody is executed again. Every time the loopbody has been completed, the test expression is evaluated once more to determine whether to leave the loop or to execute loopbody again.

Remember, whereas a `while` loop may be executed zero or more times, a `do while` loop is always executed at least once.

As with the `while` loop, the loopbody of a `do while` loop may consist of a simple or a compound statement (see above example for the difference in the way that these are written). The following program contains an example of a `do while` loop with a loopbody consisting of a simple statement:

```
main()
{
     int times = 1;

     do
          printf("I'll do anything once!\n");
     while (++times <= 1);
}
```

This `do while` loop is executed only once, because the value of the variable `times` is greater than one the first time the test expression is evaluated.

4 The `for` Loop

The general form of the `for` loop is:

```
for (initialise; test; update)
     loopbody
```

4.1 Initialisation

When a `for` loop is encountered, the initialise substatement is executed first. Note that this is the only time that the initialising part of the `for` statement is executed.

4.2 Test

Once the loop has been initialised, the test expression is evaluated. If it is false, control passes to the first statement after the for loop, and the loopbody is not executed at all.

4.3 Loopbody

If the test expression evaluates to true, the statements in the loopbody are then executed in turn.

4.4 Update

When they have all been executed once, the update substatement is executed. After that, the test expression is evaluated once more. If it is false, the loop ends, and control passes on to the first statement after the loop. If it is true, the loop is iterated once more.

The order in which the parts of the for loop are executed is therefore: initialise, test, loopbody, update. Remember that although the sequence (test, loopbody and update)

may be repeated many times, the expression in the initialise substatement is executed only once.

The loopbody in a `for` loop can be a simple or a compound statement, and these are written as described in the section on `while`..

An example of a `for` loop with a compound statement would be:

```
for (count = 0; count < 2; count++)
{
     printf("Listen very carefully!\n");
     printf("I shall say this only once.\n");
}
```

(It is quite deliberate that this loop is iterated twice!)

4.5 A Compact `while` Loop

The `for` loop may be regarded as a particularly compact and easily-readable kind of `while` loop. Like the `while` loop, it is an entry-condition loop, so the loopbody may be executed zero or more times. However, the `for` loop draws together in one place information that could be widely scattered in an equivalent program using a `while` loop.

The following two programs carry out the same programming task: they output the Fahrenheit equivalent of temperatures from 0 to 100 degrees Celsius. Note, however, that the program with the `for` loop is not only shorter than the one using a `while` loop, but it conveniently puts together on one line the initial value for the variable `celsius`, the test that is evaluated to decide whether to execute the loopbody, and the step that increments `celsius` each time the loop is iterated.

`/*` Celsius converted to Fahrenheit: `while` version `*/`

```
main()
{
     double celsius = 0.0, fahrenheit;

     while (celsius < 101)
     {
          fahrenheit = (celsius * 9/5) + 32;
          printf("%d C is %5.2f F.\n",
                    celsius, fahrenheit);
          celsius++;
     }
}
```

```
/* Celsius converted to Fahrenheit: for version */
     main()
     {
          double celsius, fahrenheit;

          for (celsius = 0; celsius < 101; celsius++)
          {
               fahrenheit = (celsius * 9/5) + 32;
               printf("%d C is %5.2f F.\n",
                         celsius, fahrenheit);
          }
     }
```

The `%5.2f` in the `printf` statements in the two programs ensures that the temperatures in Fahrenheit are printed to two decimal places.

4.6 Multiple Initialisers and Updaters

The following is an example of a for loop that initialises three variables: `x`, `y` and `z`.

```
for (x = 1, y = 35, z = 113; x < z/y; y++)
     x = y + z;
```

Note that the initialise, test and update parts of the `for` loop are separated by semicolons, but the three variables to be initialised are separated from each other by commas.

How about a `for` loop which increments more than one variable?

```
for (old = 0, wise = 4; old < 70; old++, wise += 2)
{
     printf("A little bit older,");
     printf("a little bit wiser.\n");
}
```

The individual expressions in the update are separated from each other by a comma, but the updating section as a whole is preceded by a semicolon to separate it from the test expression.

4.7 The Comma Operator

The comma used to separate assignment statements in the initialising part of a `for` loop, or expressions in the updating part of the loop, is the comma operator.

An important difference between the comma operator and the comma separator used, for instance, in the declaration of multiple variables of the same type (e.g. `int a, b;`)

is that the comma operator guarantees that the expressions it separates will be evaluated in the order from left to right.

If, therefore, you were to write an updating section such as `old++, wise = 1/old` in a `for` loop, the comma operator would guarantee that `old++` would be evaluated first. This means that the expression that assigns a value to the variable `wise` would always use the incremented value of the variable `old`.

5 The null Statement

The null statement is a statement that contains no statements at all!

C allows you to write a statement that does nothing at all, and which consists simply of a semicolon. This is referred to as the null statement. You can use a null statement anywhere in a program where you would expect to use a normal statement.

It is, therefore, quite possible in C to write loops that have loop bodies consisting of a null statement, or that have null statements instead of initialising, testing or updating sub-statements.

The following `for` loop would be the top-level menu in a real-time program, and would run until switched off, since a test condition consisting of a null statement is always true:

```
for (;;)
     loopbody;
```

A while loop with a null loopbody looks like this:

```
while (expression)
     ;
```

Null-bodied `while` loops are often used in handling character strings and arrays. These topics are covered in later chapters, but the following example illustrates a typical construction:

```
while (*p++ = *q++)
     ;
```

This loop copies the string beginning at the address pointed to by `q` into successive locations beginning at the address pointed to by `p`. The loop continues until the null terminator is encountered (the null terminator being the character `'\0'`, which is used in C to mark the end of a string).

A Word of Caution!

A null-bodied loop will execute faster than an equivalent loop with the relevant statements in the loopbody. However, try to avoid the temptation of cramming all the contents of the loopbody into the test expression of a `while` loop, or into the initialise, test and update parts of a `for` loop. Although null-bodied loops have their uses and are very natural in some contexts, they can also be very hard to read!

One of the many strengths of C is that its looping statements are so clear and elegant. This clarity is a virtue that is far from trivial, as it greatly contributes to the ease with which programs can be debugged or modified. It is, therefore, unwise to sacrifice clarity for a small increase in speed, which in many applications will go unnoticed by the user.

6 Which Kind of Loop?

In deciding which kind of loop to use in a program, the first thing to establish is whether you want the body of the loop always to be executed at least once. If you do, then you need an exit-condition loop, and C has only one: the `do while` loop.

An exit-condition loop should only be needed in about 5% of loops. Most programming tasks require an entry-condition loop, so that execution of the statements in the loop depends on an initial test condition being met. If the test condition is not met, the loop is not executed at all.

The fact that the test comes at the start of the loop also makes entry-condition loops easier to read than exit-condition loops.

If you want an entry-condition loop, you can use either a `while` or a `for` loop. Which is best depends on the context.

The `for` loop is very convenient for any process that involves initialising and updating a variable and that is executed a predetermined number of times.

The `while` loop is ideal for loops in which the controlling variable needs to be changed within the loop body.

7 Nested Loops

Loops inside other loops are said to be nested. Nesting is a very useful programming technique, but can easily produce code that is confusing and hard to read.

Two helpful rules for writing nested loops are:

(a) increase the level of indentation for each nested loop
(b) avoid using more than four loops one within another

If you follow rule (a), you may well find that your program code disappears off the right-hand side of the page or screen if you fail to follow rule (b)!

You can often avoid the need for deeply-nested loops by rewriting nested code as function calls (which are described in a coming chapter).

8 Some Potential Problems and Common Mistakes

Warning!

There are some programming applications that require an infinite loop. The rest of the time, the infinite loop is a hazard to be avoided at all costs!

If you do not want to enter an infinite loop, remember, when using a `while` loop, that you must include some instruction which changes the value of the variable in the test expression in such a way that the expression will eventually evaluate as false.

The value of the integer variable `x` is changed in the following program fragment, but not in the correct direction. Once the expression `x` is less than `100` becomes true, it remains true (ultimately the value of `x` will sink below the range for negative integers on your computer, and then, by default, become a very large positive number and end the loop).

```
while (x < 100)
     x--;
```

Impossible Conditions

It is possible to have a `while` loop in which the expression is never true, so the loopbody is never executed. Although less disastrous than an infinite loop, it is equally pointless.

The Assignment Operator and the Test for Equality

Do not confuse the assignment operator = with the logical test for equality == !!

Commas and Semicolons

A common mistake is to use semicolons instead of commas between multiple entries in the initialising part of a `for` loop, or between multiple updating expressions.

The opposite mistake is to use commas instead of semicolons to separate the initialising expressions from the test expression or the test expression from the updating expressions.

Never put a semicolon at the end of the `while`, `do` or `for` line of a loop, before the loop body starts:

```
while (test);            /* Wrong! */
     loopbody;
```

Initialising, Updating and Ending Values

Sometimes, a program will compile correctly, but will output incorrect results. When checking the source code, pay particular attention to the initial values assigned to the variables and the way in which the test conditions of loops are specified.

Also check to make sure that variables are incremented or decremented when you need them to be, whether before or after a test.