C Programming - Chapter Summary

1 Functions and main

A C program consists of one or more functions. These may be written by the programmer, or may be standard library functions.

Program execution starts with the function called `main`, and all C programs must therefore contain a `main` function. It does not need to be at the beginning of the program, but it must exist somewhere.

`main` consists of the name `main`, followed by an open curly bracket `{` known as an open brace. It ends with a close brace,`}`. The statements between the two braces are executed in turn until the final closing brace is encountered and execution stops.

```
main()
{
     printf("Greetings programmers!\n");
}
```

There are two functions in the above example: `main` and `printf`. Note that the name of a function is always followed by brackets, which, depending on the function, may contain arguments", that is to say, information passed to the function by the calling function. In the above example, the function `main` calls the function `printf`, and `printf` has the argument `"Greetings programmers!\n"`.

Note also that the statement calling the `printf` function ends with a semicolon. This must not be left out!

2 Introduction to Variables

As will be explained further in Data Types and Expressions Chapter, any data items used in `main` must be declared at the start of the `main` function. The declaration must specify the type of the variable (e.g. whether it is an integer or a character) and must give the name of the variable, which, by convention, is written in lower-case letters. On compilation, the declaration assigns space in memory for storing that particular variable, and the type specifier indicates the amount of memory required.

The following declarations are for two variables, called `number` and `letter`. The variable `number` will be used to store integer numbers and is therefore of type `int`. The variable `letter` will store characters, so is of type `char.`

```
int number;
char letter;
```

Note again that the declarations for these variables end in semicolons. Except for complex statements, which end with a closing brace `}` and will be described in the chapters on looping and decision making, all C program statements must end in a semicolon.

3 Operators, Operands and Expressions

C programs use operators, such as `+`, `-` and `=`, to modify and test variables, constants and expressions, which, because they are acted on by operators, are sometimes referred to as operands. An expression, such as `a+b`, is simply any combination of operators and operands.

Operators are discussed more fully in the Expressions, Assignments and Operators chapter, but the three mentioned already are used in arithmetic operations. `+` adds the value of the expression on its right to the value of the expression on its left. Similarly, `-` subtracts the value on its right from the value on its left, while `=` assigns the value on its right, the rvalue, to the variable on its left, the lvalue

4 Output: `printf`

There are no input or output functions written into the definition of the C language. C compilers come with standard libraries, however, containing a wide range of useful input and output functions.

`printf` is an output function for displaying text on the screen. The `f` in `printf` stands for formatted print function.

`printf` will print on the screen whatever is enclosed between the double quotation marks within the brackets. It will not necessarily print the message on a new line, however. To start at the left margin on a new line, a backslash followed by an n (for newline) is needed:

```
printf("\nStart on a new line");
```

The above command will print the words **Start on a new line** ranged left on a new line. The characters `'\n'` do not print out. They are purely control characters, and are referred to as an "escape sequence".

There are other escape sequences available for use with `printf`, each starting with the backslash `'\'` character:

`\t`	tab	`\\`	backslash
`\b`	backspace	`\'`	single quotation mark
`\r`	carriage return	`\"`	double quotation mark
`\f`	form feed	`\?`	question mark
`\a`	audible alert	`\v`	vertical tab

One of the most useful features of `printf` is that can include the value of expressions in its output. This is done by including conversion specifications in the control string enclosed between the double quotation marks, and following the control string with the corresponding expression or expressions whose values are to be included in the output.

The general form of the `printf` function is therefore:

```
    printf( control string, expression, expression...);
```

There should be as many expressions, separated by commas, as there are conversion specifications in the control string.

A conversion specification begins with a percentage sign, `%`, and indicates how the value should be printed. Including `%d`, for instance, prints out a decimal integer, while `%c` will print out a character.

If a and `b` are variables of type `int`, the following `printf` statement will print out the current values of a and b, and of the expression a - b:

```
printf("\nThe difference between %d and %d is %d.",a,b,a-b);
```

The following is a list (not complete) of the conversion specifications available for `printf`:

`%d`	print a decimal integer
`%u`	print an unsigned integer
`%o`	print an unsigned octal integer
`%x,%X`	print an unsigned hexadecimal integer
`%c`	print a character
`%s`	print a string
`%f`	print a floating-point number in decimal notation
`%e,%E`	print a floating-point number in exponential notation
`%g,%G`	print a floating-point number in either decimal or exponential notation depending on the size of the number
`%p`	displays a pointer
`%n`	print the number of characters printed so far when the associated argument is an integer pointer to this value
`%%`	print a percentage sign

Modifiers may be placed after the % sign in `printf` conversion specifications:

> a number indicates the width of the field in which the output value will be printed. For example, `%6d` will print an integer in a field of six spaces
>
> a dash, `-`, indicates that the output should be left justified in the field. `%-8u` will print an `unsigned` integer left justified in a field of eight spaces
>
> `l` indicates a `long` value. For example, `%ld` will print a `long` integer. When used with `%e`, `%f` and `%g` indicates a `double` precision value
>
> a number after a decimal point indicates the number of decimal places of a floating-point number that will be printed. `%8.3f`, for instance, will print a floating-point number to three decimal places in a field of eight spaces
>
> an asterisk, `*`, may be used instead of either a field width or a decimal place specification, and the actual values for these modifiers can then be included in the argument list with the names of any other variables to be printed

5 Input: `scanf`

The function `scanf` obtains input from the keyboard and assigns values to variables. The general form of `scanf` is:

```
    scanf(control string, expression, expression...);
```

As with `printf`, conversion specifications in the control string indicate the type of input required, but the corresponding expressions following the control string are the addresses where the values will be stored. These addresses are usually given in the form of the address operator % followed by the name of the variable to which the value is being assigned.

`scanf` cannot be used for printing messages on screen, so a `printf` statement is often used before a `scanf` to prompt the user for input. The following example asks the user for a number, and then uses `scanf` to assign the value of the number entered by the user to a variable called `number`.

```
    printf("\nWhat number would you like to choose?\n");
    scanf("%d",&number);
```

The conversion specification in this example is %d, for a decimal integer. The conversion specifications for `scanf` are similar to those for `printf`, although there is an additional %h specification for reading `short` integers, and there is no %g specification. The %e and `%f` options work in the same way, accepting floating-point numbers in either decimal or exponential notation.

Note that special care needs to be taken when a program includes more than one `scanf` statement. When entering information, the user types the input and then presses the RETURN key. A `scanf` statement will assign the input to the variable or variables indicated, but if no action is taken, the newline character `'\n'`, which was generated when the RETURN key was pressed, will remain in the input buffer. When, subsequently, another `scanf` statement is encountered in the program, it will immediately be triggered by the newline character in the input buffer, as if the user had pressed the RETURN key before typing in any input.

When a `*` modifier is used with a conversion specification in a `scanf` statement, it means ignore this input. `%c` is the character specification, so including `%*c` at the end of a control string will therefore cause `scanf` to discard the newline character generated by the RETURN key, thus ensuring that it does not interfere with any subsequent `scanf` statements, as illustrated in the following example:

```
printf("\nThink of a number - any number!");
scanf("%d%*c", &number1);
printf("\nNow choose another number.");
scanf("%d",&number2);
```

Without the `%*c` in the control string of the first `scanf`, the second `scanf` would have been triggered before the user could enter a number.

6 Comments

Comments may be included anywhere in a source file by using the opening and closing markers `/*` and `*/`, thus:

`/*` A helpful comment describing something in the program `*/`

Everything between the `/*` and the `*/` is removed by the C preprocessor before compilation, so comments do not take up any space in your compiled program.

Use comments to remind you and anyone else reading your code what your program does, and how it does it. Always remember to match each opening marker `/*` with a corresponding closing marker `*/`, however, and never try to nest one comment inside another.

7 Developing and Compiling a C program

The first stage in writing a C program, before any code is written, is to consider what the program is required to do, and how it might best achieve that aim.

Source code for the program is then written in one or more source files, using a suitable editor. Source files are usually given names with the extension `.c`, as in **program.c**.

Source files can then be compiled to produce machine code in object files, whose names have the extension `.obj`, as in **program.obj**. The compilation process often throws up errors in the source code, however, and these must be corrected before compilation can be completed.

When all the source files have been successfully compiled, the link program is then used to join the object files to each other and to functions from the standard C libraries. This produces a final executable file, with a name ending in the extension `.exe`, as in **program.exe**.

To run the program, simply type its name, without any extension. When run, the program still may not do what is required. In this case, the appropriate source code must be altered or rewritten, then compiled and linked once more. The program will then need to be tested again to make sure that the original problem has been solved, and that no new problems have been introduced.