1

Getting Started With C

In this chapter, we introduce many of the basic features of C. These include the data types, operators and statements most commonly used in C programs. The intent is not to be too formal or complete, but rather to get the reader to the point where he/she becomes familiar with the nature of a C program and can construct non-trivial programs in the shortest possible time. Topics which are glossed over in the interest of brevity and/or simplicity are treated in greater detail in other chapters of the book.

The C programming language is built around a few basic concepts. These include:

(1) **keywords**, e.g., `if`, `while`, `do`, `for`, `int`, `char`, `float`. Each keyword has a specific meaning in the context of a C program, and may not be used for any other purpose. For this reason, they are usually called **reserved words**. Appendix A gives the complete list of C keywords.

(2) a small number of **data types**, e.g. character (designated by `char`), integer (`int`), floating-point (`float`). Each data type defines constants of that type, for example, 't' is a character constant and 23 is an integer constant. Data types are discussed in more detail later in this chapter (Section 1.3).

(3) **variables**, e.g., `sum`, `count`, `numStudents`. Variables are used to hold data of different types as well as for the names of functions. Variables are discussed in Section 1.4.

(4) **operators**, e.g., + (addition), == (test for equality), || (logical or), < (less than), = (assignment). One of the hallmarks of C is that it has a rich set of operators, many of which are not found in other languages. An operator specifies the action to perform on its operand(s). An operand can be a constant or a variable. Operators are discussed in Section 1.5.

(5) **expressions**, e.g., `a + b * c, x = sum / n, (a == b) || (b == c).`

Expressions are formed by combining operators, constants and/or variables according to prescribed rules. Expressions are discussed in Section 1.5.

(6) **statements**, e.g., `x = sum / n;`
`if (a < b) small = a; else small = b;`

Statements specify actions to be performed by the program. The commonly used statements are discussed starting from Section 1.6.

(7) **functions** – in C, as in other languages, functions are the building blocks from which larger programs are constructed. In a well-written program, a function performs a single, well-defined task. The dividing of a large job into smaller tasks, each of which is implemented by a function, is a feature of the top-down, modular approach to programming. Functions are discussed in detail in Chapters 3 and 5.

The C language does not contain, among other things, built-in input/output statements. Input/output facilities are provided by means of 'standard' functions provided in the 'standard' library. An integral part of any C programming system is the provision of this library of functions. In addition to I/O functions, the library normally includes

- functions for character and string handling;
- mathematical functions;
- time, date and other system-related functions;
- functions for the dynamic allocation of storage;
- constant definitions for implementation-defined limits, for example, the maximum and minimum values of `int` variables.

In addition, many libraries on specific systems contain functions for screen-handling or graphics. You will need to refer to your specific compiler manual for details of the library functions provided.

Almost any program one writes will need to perform some input/output. Any such program will therefore need to use at least one of the standard I/O functions. These functions use variable (and other) declarations contained in a special **header** file. (Each class of functions has its own header file). For a user's program to compile and run properly, it is necessary to specify the header file(s) required by the program. The header file containing the declarations for the I/O functions is denoted by `⟨stdio.h⟩`. Since it is almost always needed, most users' programs must

be preceded by the statement

```
#include <stdio.h>
```

This informs the C compiler to include the declarations in the file `stdio.h` in the user's program.

Other commonly used header files are:

`ctype.h` – contains declarations for character–handling functions (Chapter 3).
`limits.h` – contains definitions for implementation–dependent limits, for example, minimum and maximum values of various data types.
`math.h` – contains declarations used by the mathematical functions; these include trigonometric, hyperbolic, exponential and logarithmic functions.
`stdlib.h` – contains declarations for miscellaneous functions, for example, random number functions and functions for searching and sorting.
`string.h` – contains declarations for string functions (Chapter 4).
`time.h` – contains declarations for system time and date functions.

You will need to consult your compiler manual for the complete list of available header files and library functions.

1.1 The first example

Write a C program to print the message

```
Welcome to the World of C
```

One solution is program P1.1.

Program P1.1

```
#include <stdio.h>

  main()
  {
    printf("Welcome to the World of C");
  }
```

A C program consists of one or more 'functions' (or subprograms), one of which must be called `main`. Our solution consists of just one function, so it must be called `main`. The (round) brackets after `main` are necessary because, in C, a function name must be followed by a list of 'arguments' enclosed in brackets. If there are no 'arguments', the brackets must still be present; `main` has no 'arguments' so the brackets alone are present.

Next comes a left brace (curly bracket); this indicates the start of the 'body' of the function. A matching right brace indicates the end of the 'body' of the function. The braces { and } enclose the statements which comprise the body of the function. The braces are equivalent to `begin` and `end` of languages like Pascal or Algol. The body of our solution consists of a single statement (actually a 'call' to the standard output function `printf`). The 'argument' to `printf` in our example is a **string constant** (also called a **character string**); this is simply a set of characters enclosed in double quotes ("). The effect is that the string (without the quotes) is printed. In general, a statement in C is terminated by a semicolon (`;`). In the example, a semicolon follows the right bracket (of the call to `printf`) to satisfy this requirement.

In C, a program begins execution at the first statement of `main` and terminates when the right brace (indicating the end of `main`) is encountered. Execution also terminates if a `return` statement (see Section 3.1) is encountered in `main`.

1.1.1 Running the program

Having written the program on paper, the next task is to get it running on a real computer. How this is done varies somewhat from one system to the next but, in general, the following steps must be performed:

(1) type the program to a file. Some compilers require that the name of the file end in '`.c`', so we could name our file `welcome.c`. Even if the compiler does not require it, it is still good practice to append `.c` to the file name to remind us that a C program is stored in it. Since this program uses the standard output function `printf`, the program is preceded by the line

```
#include <stdio.h>
```

(2) invoke the C compiler to compile the program in the file `welcome.c`. For example, on the VAX/VMS operating system, the command for

doing this is:

```
cc welcome
```

where `cc` specifies that the C compiler is required; it expects to find a file `welcome.c` containing a C program.

Some systems allow you to choose 'compile' from a menu of choices. (You will need to find out the specific command for your compiler).

Assuming there are no errors, the compiler will produce an 'object file', typically called `welcome.obj`, containing the machine code equivalent of the C program. This object file will contain 'place holders' for functions or variables **used** in the program but not **defined** in the program. References to these functions or variables will be resolved in the next step.

(3) invoke the **linker** to resolve references to functions or variables not defined in the program. For example, on the VAX/VMS operating system, the command for doing this is:

```
link welcome
```

Since it only makes sense to link an **object** file, the linker will look for a file `welcome.obj`. But where does it find the functions or variables that need to be resolved? Many systems automatically search the standard library. Others require that the link command specify the library or libraries to be searched. The result of linking is that an 'executable file', typically called `welcome.exe`, is produced. This file contains code which can be executed by the computer. In this example, the linker will find the function `printf` in the standard library and include its code in the executable file being produced.

(4) run the program. For example, on the VAX/VMS operating system, the command for doing this is:

```
run welcome
```

Since only executable files can be 'run', the operating system looks for a file `welcome.exe`. When this program is run,

```
Welcome to the World of C
```

will be printed on the screen.

1.1.2 A word on program layout

C does not require the program to be laid out as in the example. An equivalent program is

```
#include <stdio.h>
main() {printf("Welcome to the World of C"); }
```

For this small program, it probably does not matter which version we use. However, as program size increases, it becomes imperative that the layout of the program highlight the logical structure of the program, thus improving its readability. Indentation and clearly indicating which right brace matches which left brace can help in this regard. We will see the value of this principle as our programs become more substantial.

The newline character (written as `\n`*, read as backslash n)*

Suppose we wanted to write a program to print the following lines:

```
Welcome to the World of C
Hope you enjoy it
```

Our initial attempt might be

```
#include <stdio.h>

main()
{
   printf("Welcome to the World of C");
   printf("Hope you enjoy it");
}
```

However, this does not quite give us what we want. When this program is executed, it will print

```
Welcome to the World of CHope you enjoy it
```

Note that the two strings are joined together. This happens because `printf` does not place output on a new line, unless this is specified explicitly. Put another way, `printf` does not automatically supply a **newline** character after printing its argument(s). (A newline character would

cause subsequent output to begin at the left margin of the next line). In the example, a newline character is not supplied after 'C' is printed so that 'Hope...' is printed on the same line as 'C' and immediately after it.

To get the desired effect, we must tell `printf` to supply a newline character after printing '...C'. We do this by using the character sequence `\n` (backslash n) as in program P1.2:

Program P1.2

```
#include <stdio.h>

main()
{
   printf("Welcome to the World of C\n");
   printf("Hope you enjoy it\n");
}
```

The first `\n` says to terminate the current output line; subsequent output will start at the left margin of the next line. Thus 'Hope...' will be printed on a new line. The second `\n` has the effect of terminating the second line. If it were not present, the output will still come out right, but only because there is no more output to follow. (This is also the reason why our first program worked without `\n`).

As an embellishment, suppose we wanted to leave a blank line between our two lines of output. Each of the following sets of statements will accomplish this:

```
(1) printf("Welcome to the World of C\n\n");
    printf("Hope you enjoy it\n");

(2) printf("Welcome to the World of C\n");
    printf("\nHope you enjoy it\n");

(3) printf("Welcome to the World of C\n");
    printf("\n");
    printf("Hope you enjoy it\n");
```

The backslash (\) signals that a special effect is needed at this point. The character following the backslash specifies what to do. The combination is usually referred to as an *escape sequence*. The following are commonly used escape sequences:

\n says to issue a newline character;
\t says to issue a tab character;
\b says to backspace;
\" says to print ";
\\ says to print \.

The complete list of escape sequences is given in Section 4.1.

1.2 Comments

Comments may be included in a C program to describe what a function is supposed to do, or, perhaps, to clarify some portion of code. A comment begins with the two-character symbol /* and is terminated by another two-character symbol */. It can occur anywhere that a space, tab or newline (so-called 'whitespace' characters) can. For example

```
#include <stdio.h>

main()    /*  This is our greeting program */
{
  printf("Welcome to the World of C\n");
        /* \n – newline character */
}
```

For the purposes of the compiler, a comment is treated as a single whitespace character.

One restriction on comments is that they cannot be nested. Consider

```
/* start of outer comment

  /* inner comment */

end of outer comment   */
```

Here the first `*/` ends the comment started with the first /*. The second /* is simply part of the first comment. Since, as far as the compiler is concerned, the comment ends with the first `*/`, the word `end` will cause an error. A way to 'comment out' a portion of a program (which itself may contain comments) is discussed in Section 11.1.4.

1.3 Data types

C supports the following data types (among others). Each data type defines constants of that type.

- `char` – a single character; a character constant is a single character enclosed by single quotes (apostrophes). Examples: `'a'`, `'7'`, `'&'`. Characters which are not printable or which have a special use are represented by a backslash (\) followed by another character. For example, `'\n'` represents the newline character and `'\''` (single quote, backslash, single quote, single quote) represents a single quote.
- `int` – integer (whole number); examples: `25`, `0`, `-1`, `-247`, `32767`. C also allows the use of octal and hexadecimal constants. An octal constant begins with a leading zero (`0`) and a hexadecimal constant begins with a leading `0x` or `0X`. For example, the decimal number `59` can be written in octal as `073` and in hexadecimal as `0x3b` or `0x3B` or `0X3b` or `0X3B`.
- `double` – double-precision floating point number; a `double` constant can be written with a decimal point (`73.6`, `-739.31`, `1.0`, `3.1416`), with an exponent using `e` or `E` (`736 E-1` or `736e-1`, whose value is 736×10^{-1}, that is, 73.6) or both a decimal point and an exponent (`0.736 e2` or `0.736 E+2`, whose value is 0.736×10^{2}, that is, 73.6).
- `float` – a single-precision floating point number; other languages call this data type `real`. A `float` constant is written by adding the suffix `f` or `F` to a `double` constant, for example, `73.6f` or `0.736e2F`.

The size of these types depends on the architecture of the particular machine. On many machines, an `int` variable occupies 2 or 4 bytes and a variable of type `float` gives about 6 or 7 significant digits using 4 bytes. Normally, a variable of type `double` gives about twice the number

of significant digits as a variable of type `float`, and is used when greater accuracy is required.

(The types `char`, `int` and `double` may be preceded by 'type qualifiers' to alter their meaning. These qualifiers are discussed in Section 1.4).

A data type commonly used in C is the **string**. A string constant consists of a set of zero or more characters enclosed by double quotation marks (") . Examples:

```
"Once upon a time"
"Pass"
"times 2 ="
"?##$%#)*(_+"
```

C does not allow a string constant to be continued on to another line. In other words, both opening and closing quotes must appear on the same line. However, adjacent strings can be concatenated at compile time. This allows a long string to be broken up and placed on more than one line, as in the following example:

```
printf("Part of a long string can be placed on "
       "one line and the other part could be "
       "placed on the next line. Note that the "
       "pieces are separated by white space, not "
       "commas\n");
```

When this statement is compiled, the five strings are concatenated, forming one string.

In C, a string is stored in an array of characters (see Section 4.4).

For those familiar with the data type `boolean` or `logical` (with constants `true` and `false`) from other languages, it should be noted that C does not have a similar type, at least not explicitly. Rather, C uses the concept of the value of an expression to denote `true/false`. An expression can be used in any context where a `true` or `false` value is required. The expression is considered `true` if its value is non-zero and `false` if its value is zero. For example, if `a` and `b` are integer variables, then it is permissible to write

```
if (a + b) statement1 else statement2
```

`statement1` is executed if `(a + b)` is non-zero (true); `statement2` is executed if `(a + b)` is zero (false).