CHAPTER 1

Welcome to C

C is a procedural programming language. The early history of C is closely parallel to the history of UNIX. This is because C was specifically developed to write the operating system UNIX, which was introduced by Bell Laboratories in 1969 as an alternative to the Multics operating system for the PDP-7 computer. The original version of UNIX was written in assembly language, but programs written in assembly language are less portable than programs written in high-level languages; hence, the people at AT&T decided to rewrite the operating system in a high-level language. This decision was followed by the hunt of a suitable language, but there was no suitable high-level language that would also permit bit-level programming.

During the same period (1970), Kenneth Thompson developed a language for systems programming that was named B after its parent language BCPL (which was developed by Martin Richards in 1967). In 1972, C made its first appearance, as an improved version of B. Developed by Dennis Ritchie, C's name is derived from B (i.e., in the alphabet, the letter *C* follows the letter *B*, and in the name BCPL, the letter *C* follows the letter *B*).

Ritchie, with a group of researchers working at Bell Laboratories, also created a compiler for C. Unlike B, the C language is equipped with an extensive collection of standard types. In 1973, the new version of UNIX was released in which more than 90 percent of the source code of UNIX was rewritten in C, which added to its portability. With the arrival of this new version of UNIX, the computing community realized the power of C. Following the publication of the book *The C Programming Language* in 1978 by Brian Kernighan and Dennis Ritchie, C shot to fame.

In 1983, the American National Standards Institute (ANSI) formed a committee, named X3J11, to create a standard specification of C. In 1989, the standard was ratified as ANSI X3.159-1989, "Programming Language C." This version of C is usually called ANSI C, Standard C, or just C89. In 1990, the ANSI C standard (with a few minor modifications) was adopted by the International Standards Organization (ISO) as ISO/IEC 8999:1990. This version is popularly known as C90. In 1995, C89 was modified, and an international character set was added to it. In 1999, it was further modified and published as ISO 9899:1999. This standard is popularly called C99. In 2000, it was adopted as an ANSI standard.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2967-5_1) contains supplementary material, which is available to authorized users.

S. Chavan, *C Recipes*, DOI 10.1007/978-1-4842-2967-5_1

Programs, Software, and Operating System

Before proceeding, let me explain the meaning of the term *computer program* (hereafter, simply *program*). Well, a program is nothing but a set of instructions to be fed to a computer so the computer can do some desired work. The relationship between a program and software can be expressed as follows:

program + portability + documentation + maintenance = software

Portability means the ability of a program to run on different platforms (e.g., the Windows platform, UNIX platform, etc.). Documentation means a user's manual and comments inserted in a program. Maintenance means debugging and modifying the program as per the requests of users.

Microsoft Windows is an *operating system*. It consists of a graphical user interface (GUI). *Graphical* means pictorial, and *interface* means middleman, so a GUI is a pictorial middleman between the user and the internal machinery of a computer that assists a user (meaning a computer user). In a hotel, the waiter takes your order, approaches the kitchen, collects the dish ordered by you, and serves you. Similarly, the operating system takes your order, approaches the internal machinery of computer, and then serves you.

Machine Language and Assembly Language

A microprocessor can be aptly described as the brain of a personal computer. This microprocessor is nothing but a single chip. Various microprocessors are available. *Microprocessor* and *central processing unit* (CPU) are synonymous. A microprocessor consists of an important component called an arithmetic and logic unit (ALU), which performs all the computations. A salient feature of an ALU is that it understands only machine language, which in turn consists of only two alphabets, namely, 0 and 1 (by contrast English consists of 26 letters). Here is a typical machine language instruction:

```
10111100010110
```

A few decades back, programmers did use machine language to write programs. The then-keyboard consisted of only two keys, captioned 0 and 1. Writing a machine language program and then typing it in a computer was a laborious and tedious job. Then came the assembly languages, which eased the task of programmers. Assembly languages are low-level languages. The following is a typical assembly language statement (which performs a multiplication of two numbers), which is certainly more readable than the machine language instruction given earlier:

```
MUL X,  Y
```

If a machine language program consists of, say, 50 statements, then the corresponding assembly language program would also consist of approximately 50 statements. As ALU understands only machine language, special software (called an *assembler*) was developed to translate assembly language programs into machine language programs.

Procedural Languages

A typical procedural language is closer to English than assembly language. For example, here is a statement in the procedural language Pascal:

```
If (rollNumber = 147) Then Write ('Entry denied.');
```

The meaning of this statement, which is quite obvious, is as follows: if the value of `rollNumber` is 147, then display the message "Entry denied." on the screen. To translate a procedural language program into a machine language program, software called a *compiler* is used. Procedural languages are high-level languages.

Programmers use procedural languages in conjunction with the techniques of structured programming. What is structured programming? In a broad sense, the term *structured programming* refers to the movement that transformed the art of programming into a rational science. It all began with a letter by Edsger Dijkstra, "Go To Statement Considered Harmful," published in the March 1968 issue of *Communications of the ACM*. Structured programming rests on the following cornerstones:

- *Modularity*: Instead of writing a one big program, split your program into a number of subprograms or modules.
- *Information hiding*: The interface of a module should exhibit only the least possible information. For example, consider a module that computes the square root of a number. The interface of this module will accept a number and return the square root of that number. The details of this module will remain hidden from the users of this module.
- *Abstraction*: Abstraction is the process of hiding the details in order to facilitate the understanding of a complex system. In a way, abstraction is related to information hiding.

However, as programs grew larger and larger, it became clear that the techniques of structured programming are necessary but not sufficient. Computer scientists then turned to object-oriented programming in order to manage more complex projects.

Object-Oriented Languages

We use computer programs to solve real-life problems. The trouble with the structured paradigm is that using it, you cannot simulate real-life problems on computers conveniently. In a structured paradigm, you use data structures to simulate real-life objects, but these data structures fall far short in simulating real-life objects. Car, house, dog, and tree are the examples of real-life objects, and it is expected that a programming language should be capable of simulating these objects to solve real-life problems. The object-oriented paradigm tackles this problem at its root simply by providing software objects to simulate real-life objects. An object provided by an object-oriented paradigm is an instance of a class and possesses identity, properties, and behavior like real-life objects do. For example, if `Bird` is a class, then `parrot`, `peacock`, `sparrow`, and `eagle` are objects

or instances of the class `Bird`. Also, if `Mammal` is a class, then `cat`, `dog`, `lion`, and `tiger` are objects or instances of the class `Mammal`. Compared to the structured paradigm, the object-oriented paradigm is more capable of using existing code. *Code* means a program or its part.

The object-oriented paradigm is as old as the structured paradigm. The movement of the structured paradigm began with Dijkstra's famous letter "Go To Statement Considered Harmful" in 1968, whereas the object-oriented paradigm has its origin in the programming language SIMULA 67, which appeared in 1967. However, the object-oriented capabilities of SIMULA 67 were not very powerful. The first truly object-oriented language was Smalltalk. In fact, the term *object-oriented* was coined through Smalltalk literature. C is not object-oriented language; it is only a procedural language. In 1983 Bjarne Stroustrup added object-oriented capabilities to C and christened this new language as C++, which was the first object-oriented language widely used and respected by the computer industry. Today, the most popular object-oriented language is Java. Object-oriented languages are high-level languages.

Terminology in Computers

In almost all sciences, the terminology is derived from languages like Greek or Latin. Why? If you derive terminology from the English language, then there is a risk that confusion may occur between the technical meaning and the current usage of that term. In computers, however, terminology is derived from English, causing confusion to new learners. English words such as *tree*, *memory*, *core*, *root*, *folder*, *file*, *directory*, *virus*, *worm*, *garbage*, etc., are used as technical terms in the field of computers. You might be unaware that particular term has some technical meaning attached to it apart from its current nontechnical meaning. To avoid confusion, always have a good computer dictionary on your desk. Whenever in doubt, refer to the dictionary.

Compiled and Interpreted Languages

When a computer scientist designs a new programming language, the major problem is the implementation of that language on various platforms. There are two basic methods for implementing a language, as follows:

- *Compilation*: Code in a high-level language is translated into a low-level language. A file is created to store the compiled or translated code. You are then required to execute the compiled code by giving an appropriate command.
- *Interpretation*: Instructions in code are interpreted (executed), one by one, by a virtual machine (or interpreter). No file is created.

These methods are now discussed in detail.

Compilation

In compilation, the source code in a high-level language is translated into the machine language of an actual machine. FORTRAN, Pascal, Ada, PL/1, COBOL, C, and C++ are compiled languages. For example, consider a C program that displays the text "Hello" on the screen. Say `hello.c` is the file that contains the source code of this program (source code files in C have the extension `.c`). The C compiler *compiles* (or translates) the source code and produces the executable file `hello.exe`. The file `hello.exe` contains instructions in the machine language of the actual machine. You are now required to execute the file `hello.exe` by giving an appropriate command, and execution of the file `hello.exe` is not part of the compilation process. The executable file `hello.exe` that is prepared on the Windows platform can be executed only on the Windows platform. You simply cannot execute this file on the UNIX platform or the Linux platform. However, C compilers for all platforms are available. Hence, you can load the appropriate C compiler on a UNIX or Linux platform, compile the file `hello.c` to produce the executable file `hello.exe`, and then execute it on that platform.

The major benefit of compiled languages is that the execution of compiled programs is fast. The major drawback of compiled languages is that executable versions of programs are platform dependent.

Interpretation

In interpretation, a virtual machine is created by adding a desired number of software layers such that the source code in the high-level language is the "machine language code" for this virtual machine. For example, the language BASIC is an interpreted language. Consider a BASIC program that displays the text "Hello" on the screen. Say the source code of this program is stored in the file `hello.bas`. The source code in `hello.bas` is fed to the BASIC virtual machine, and instructions in `hello.bas` are *interpreted* (executed) by the BASIC virtual machine one by one. Also note that programming statements in `hello.bas` are machine language instructions for the BASIC virtual machine. No new file is created in the interpretation process.

The major benefit of interpreted languages is that programs are platform independent. The major drawback of interpreted languages is that the interpretation (execution) of programs is slow. BASIC, LISP, SNOBOL4, APL, and Java are interpreted languages.

In practice, a pure interpretation, as in the case of BASIC, is seldom used. In almost all interpreted languages (e.g., Java), a combination of compilation and interpretation is used. First, using a compiler, the source code in a high-level language is translated into intermediate-level code. Second, a virtual machine is created such that the intermediate-level code is machine language code for that virtual machine. Intermediate-level code is then fed to a virtual machine for interpretation (execution).

Finally, notice that all scripting languages (e.g., Perl, JavaScript, VBScript, AppleScript, etc.) are pure interpreted languages.

Your First C Program

As a tradition, the first program in a typical C programming book is generally a "Hello, world" program. Let's follow this tradition and create and run (execute) your first program. This program will display the text "Hello, world" on the screen. Type the following text (program) in a C file and save it in the folder `C:\Code` with the file name `hello.c`:

```
#include <stdio.h>
main()
{
  printf("Hello, world\n") ;
  return(0) ;
}
```

Compile and execute this program, and the following line of text appears on the screen:

```
Hello, world
```

A language is called *case-sensitive* if the compiler or interpreter of the language distinguishes between uppercase and lowercase letters. Pascal and BASIC are not case-sensitive languages. C and C++ are case-sensitive languages.

- C is a case-sensitive language, and therefore you should not confuse uppercase and lowercase letters. For example, if you type `Main` instead of `main`, it will result in an error.
- Do not confuse the file name and program name. Here, `hello.c` is the name of the file that contains the source code of the program, whereas `hello` is the program name.

To explain how this program works (or any other program, for that matter), I need to refer to individual lines of code (LOCs) in this program, and hence, I need to number these lines. Therefore, I have rewritten the program `hello` with line numbers added to it as comments (these are multiline comments), as shown here. This program produces the same output as the program `hello`.

```
/* This program will produce the same output as program hello. Only
difference is that this program contains the comments. Comments are for the
convenience of programmers only. Compiler simply ignores these comments.*/
                                                                  /* BL */
#include <stdio.h>                                                /* LOC 1 */
                                                                  /* BL    */
main()                                                            /* LOC 2 */
{                                                                 /* LOC 3 */
  printf("Hello, world\n");                                       /* LOC 4 */
  return(0);                                                      /* LOC 5 */
}                                                                 /* LOC 6 */
```

There are two types of comments in C: multiline comments (also called *block comments*) and single-line comments (also called *line comments*). Single-line comments came from C++ and have been officially incorporated into C since C99.

Now notice the program `hello` rewritten with single-line comments inserted in it, as shown here. This program produces the same output as the program `hello`.

```
// This program will produce the same output as program hello. Only
difference is that this
// program contains the comments. Comments are for the convenience of
programmers only.
// Compiler simply ignores these comments.
                                                                    // BL
#include <stdio.h>                                                  // LOC 1
                                                                    // BL
main()                                                              // LOC 2
{                                                                   // LOC 3
  printf("Hello, world\n");                                         // LOC 4
  return(0);                                                        // LOC 5
}                                                                   // LOC 6
```

Traditionally, C textbooks use only multiline comments and avoid single-line comments. I will follow this convention in this book. In the remaining part of this chapter, I will cover implicit type conversions, explicit type conversions, and the salient features of C.

Salient Features of C

C is a popular language. The following features are responsible for its huge popularity:

- C is a small language. It has only 32 keywords. Hence, it can be learned quickly.
- It has a powerful library of built-in functions. C derives its strength from this library.
- It is a portable language. A C program written for one platform (say, Windows) can be ported to another platform with minor changes (say, Solaris).
- C programs execute fast. Thus, C programs are used where efficiency matters.
- All the constructs required for structured programming are available in C.
- Good number of constructs required for low-level programming are available in C, hence C can be used for systems programming.
- Pointers are available in C, which add to its power.

- The facility of recursion is available in C for solving tricky problems.
- C has the ability to extend itself. Programmers can add the functions coded by them to a library of functions.
- C is almost a strongly typed language.

Implicit Type Conversion

In an assignment statement, the quantity that appears on the right side is called the *r-value*, and the quantity that appears on the left side is called the *l-value*. In every assignment statement, you ensure that the data type of the l-value is the same as that of the r-value. For an example, see the assignment statement given here (assume `intN` to be the `int` variable):

```
intN = 350;                    /* L1, now value of intN is 350 */
```

Here, `L1` means LOC 1. To save the space, I may use the letter `L` to denote LOC in code. In LOC 1, the l-value is `intN`, and the r-value is 350; their data type is the same: `int`. When the compiler compiles such a statement, it checks the types of both sides of the assignment statement without forgetting. This duty of the compiler is termed *type checking*. What happens if the types of both sides are not the same? Type conversion occurs! In type conversion, the type of the value on the right side is changed to that of the left side before assignment. Type conversions can be classified into two categories.

- Implicit or automatic type conversion (discussed in this section)
- Explicit type conversion (discussed in the next section)

Notice the LOC given here (assume `dblN` to be the `double` variable):

```
dblN = 35;                     /* L2, OK, now value of dblN is 35.000000 */
```

In this LOC, the type of `dblN` is `double`, and the type of numeric constant 35 is `int`. Here, the compiler promotes the data type of 35 from `int` (source type) to `double` (destination type), and then it assigns the `double` type constant 35.000000 to `dblN`. This is known as *implicit type conversion* or *automatic type conversion*. In implicit (or automatic) type conversion, type conversion occurs automatically.

In type conversion, the type of the r-value is called the *source type*, and the type of the l-value is called the *destination type*. If the range of the destination type is wider than the range of the source type, then this type of type conversion is called *widening type conversion*. If the range of the destination type is narrower than the range of source type, then this type of type conversion is called *narrowing type conversion*. The type conversion in LOC 2 is a widening type conversion because a range of `double` (destination type) is wider than a range of `int` (source type).

Here is one more example of implicit type conversion (assume intN to be the int variable):

```
intN = 14.85;                    /* L3, OK, now value of intN is 14 */
```

In this LOC, the type of numeric constant 14.85 is double, and the type of inN is int. Here, the compiler demotes the data type of 14.85 from double to int, it truncates and discards its fractional part, and then it assigns the whole-number part, 14, to intN. The type conversion in LOC 3 is a narrowing type conversion.

Here is one more example of implicit type conversion:

```
dblN = 2/4.0;                    /* L4, OK, now value of dblN is 0.500000 */
```

In this LOC, the r-value is an expression that in turn consists of the division of numeric constant 2 by numeric constant 4.0. But the type of numeric constant 2 is int, and the type of numeric constant 4.0 is double. Here, the compiler promotes the type of numeric constant 2 from int to double, and then the division of floating-point numbers 2.0 / 4.0 is performed. The result 0.5 is assigned to dblN.

■ **Note** When different types are mixed in an expression or in an assignment statement, then the compiler performs automatic type conversion while evaluating the expression or performing the assignment. While performing type conversions, the compiler tries its best to prevent the loss of information. But sometimes loss of information is unavoidable.

For example, in LOC 3, there is a loss of information (double type numeric constant 14.85 converted to an int type numeric constant 14). There is no loss of information in widening type conversion, but there is some loss of information in narrowing type conversion. Widening type conversions are always permitted by the compiler happily. Narrowing type conversions are also permitted by the compiler but with reluctance, and sometimes warnings are displayed by the compiler. Type conversions that do not make sense are simply not permitted. Some type conversions are permitted during compile time, but the error is reported during runtime. For example, notice the piece of code given here:

```
double dblN1 = 1.7e+300;                    /* LOC K */
float fltN1;                                /* LOC L */
fltN1 = dblN1;                              /* LOC M */
printf("Value of fltN1  %e\n", fltN1);      /* LOC N */
```

The compiler compiles this piece of code successfully without any warning. However, when you execute this piece of code, then instead of the expected output, the following lines of text are displayed on the screen:

```
Floating point error: Overflow.
Abnormal program termination
```

The program "crashes" during the execution of LOC M in which narrowing type conversion is attempted. When a program is terminated abruptly during runtime, in programmers' language we say that the program *crashed*.

Different languages allow the mixing of types to different extents. Language that freely allows the mixing of different types without any restriction is called a *weakly typed* language or a language with *weak typing*. A language that does not allow the mixing of different types at all is called a *strongly typed* language or a language with *strong typing*.

■ **Note** C is almost a strongly typed language.

C's strong type checking is evident in a function call. If a function expects an `int` type argument and you pass a string of characters to that function as an argument (instead of the `int` type argument), then the compiler reports an error and halts the compilation of the program, confirming that C is a strongly typed language.

Notice that I used the term *almost* in the previous Note because, to a certain extent, implicit type conversion is allowed in C, which makes C an "almost" strongly typed language, rather than a perfectly strongly typed language.

Explicit Type Conversion

Instead of leaving the type conversion at the mercy of the compiler, you can perform the type conversion explicitly. This operation is called *explicit type conversion*, *casting*, or *coercion*. The operator used in casting is called *cast*. Notice the LOC given here (assume `intN` to be an `int` variable):

```
intN = (int)14.85;                /* L1, OK, casting operation performed */
```

In this LOC, the casting operation is performed on the numeric constant 14.85. An operator cast is nothing but `(int)`. In this operation, the type of 14.85 is changed from `double` to `int`, its fractional part is truncated and discarded, and the whole-number part, 14, is returned as a numeric constant of type `int`, which in turn is assigned to `intN`. Here is the generic syntax of a casting operation or explicit type conversion:

```
(desiredType)expression
```

Here, `desiredType` is any valid type such as `char`, `short int`, `int`, `long int`, `float`, `double`, etc. In this syntax, the cast operator is nothing but `(desiredType)`. Notice that parentheses are required and are part of a cast operator. The effect of this casting operation is that the type of expression is changed to `desiredType`.

In LOC 1, a casting operation is performed on the numeric constant, but it can well be performed on variables. Notice the piece of code given here:

```
int intN;                                                    /* L2 */
double dblN = 3.7;                                           /* L3 */
intN = (int)dblN;                                            /* L4 */
printf("Value of intN is: %d\n", intN);                      /* L5 */
printf("Value of dblN is: %lf\n", dblN);                     /* L6 */
printf("Value of dbln with cast (int) is: %d\n", (int)dblN);    /* L7 */
```

This piece of code, after execution, displays the following lines of text on the screen:

```
Value of intN is: 3
Value of dblN is: 3.700000
Value of dbln with cast (int) is: 3
```

In this piece of code, a casting operation is performed on the variable `dblN` twice, first in LOC 4 and second in LOC 7. Notice that after performing the casting operation on `dblN`, the value of `dblN` remains unaffected. Actually, the casting operation is not performed on `dblN`; the value stored in `dblN` is retrieved, and then the casting operation is performed on that retrieved value (i.e., on the numeric constant 3.7). No wonder, after performing the casting operation on `dblN` with operator `(int)` in LOC 4, the variable `dblN` has remained unaffected as is evident after execution of LOC 6. The execution of LOC 6 displays the value of `dblN` to be 3.7. In LOC 7, the argument to the `printf()` function is not a variable but an expression, as shown here:

```
(int)dblN
```

In this first chapter of this book, I discussed various issues related to the C language. In the remaining chapters of the book, you will see all the C recipes. The purpose of a cookbook is to provide you readymade solutions (i.e., recipes) to your problems and in this book also you will find readymade solutions catering to needs of readers at all levels.