CHAPTER 1

Introduction to C/C++ Programming

Welcome to a treatise on object-oriented programming. This book intends to be an exhaustive resource material for C/C++ programming. This book would be useful for both experienced and inexperienced programmers, in the fact that all concepts (including Structured Programming) are explained from the scratch, and the book also provides an in-depth treatment of the various Structured and OO features. The primary focus of the book is in achieving program clarity through structured and object-oriented programming techniques. Programming languages are required to instruct computers to do user specified task, i.e. developing softwares or instructions to perform user specified tasks by controlling the computer hardware.

The growth of computers in this information age has been phenomenal and one has already witnessed the shift from large/huge-sized computer systems to desktop compatible PCs. Computers are now proving to be a necessity and no longer a luxury. The book devotes sufficient explanations on structured programming with the understanding that objects in any OO language are constructed using structured programming techniques. The internal structure of objects and the logic for manipulating these objects is implemented using structured techniques.

Another reason for an exhaustive treatment of C is to provide a smooth transition from C code to OO (C++) code. This flexibility is to support a gracious or non-restrictive shift from structured to OO programming. And also the marketing trend has been so that C and C++ are viewed as integrated entities allowing the user to decide on the programming paradigm he/she wants.

1.1 A Computer's Make-up (Organization)

A computer is a device that is capable of performing computations and making logical decisions at speeds alarming when compared with human beings. It is more often the repetitive nature of the tasks and the data size to be handled that one favours computers. Computers process data under the control of instruction sets called *computer program.* The program guides the computer through orderly sets of user specified actions. These users are referred to as *programmers.* To briefly review the various units that a computer comprises are as follows:

1. ***Input Unit:*** This unit obtains information (data or program) from various input devices such as keyboard or mouse and makes it available for other units to process

and decide on further course of action. This is also referred to as the receiving section of a computer.

2. ***Output Unit:*** This takes in the processed information and places it on the output devices to make it available for external use. Examples of output devices are monitor, printers, etc.
3. ***Memory Unit:*** This unit retains the input information for the portion of time it is required for processing and the output information for the time span till it can be redirected to the output unit. This unit is also referred to as Primary Memory or the Random Access Memory (RAM) normally.
4. ***Arithmetic and Logic Unit (ALU):*** This is in fact the manufacturing section of the computer. It aids in performing arithmetic and/logical decision-making operations to transform the input to the desired output. In other words, this section manufactures the to be projected or made use of information.
5. ***Central Processing Unit (CPU):*** This unit acts as a coordinator and supervises the operation of the remaining sections. It decides the timing of information transfer from input unit to memory unit, memory unit to ALU for performing calculations and then finally from memory unit to output unit.
6. ***Secondary Storage:*** This is the long-term and high capacity storage section of the computer. Programs and data not required by the various units are stored here on a permanent basis. Information access time from secondary storage is large in comparison with primary memory. The advantageous fact is that they are less costly in comparison with their primary counterparts.

The form of computing where only one job/task is performed at any point of time (also commonly referred to as single user processing) proved to be inefficient in terms of system throughput. Software systems called Operating Systems evolved to help in a smooth transition/switching between tasks/jobs and thereby improve upon throughput. An OS is typically an optimal resource manager that aids in software–hardware interaction in the best possible manner. Later, evolved multiprogramming to allow simultaneous execution of several jobs. This simultaneous aspect is more of a simulation and is based on time-shared or time-sliced execution of user's tasks. Though it provides that multiprogramming effect, at any point of time it is only one task that is executed. The prime advantage of time sharing is the fact that the user receives responses almost immediately without having to wait for long periods as was in the earlier single user processing case.

1.2 Naming Convention of C++

C++ evolved from C, a structured programming language. There is in fact reasoning behind naming the language as C++ and not as ++C. The ++ is the postincrement operator available with C. The postincrement operator first performs the assignment operation and then increments. Here it conveys the fact that the additional or new features of the language (OO features) will be made available after the assignment of the features of its predecessor or C language. In other words, C++ is existing C language features support first and then the incremental OO features. C was developed by Dennis Ritchie at Bell Laboratories while C++

developed by Bjarne Straustroup supports object oriented programming (OOP). The current trend in software industry is to develop software quickly, correctly and economically. C++ proves to be a good tool for realizing the above goals.

1.3 Need for C++

Objects are essentially reusable software components and model real world entities. Object-oriented programming is more productive than structured programming. Object languages are a lot easier to maintain or modify. C++ is a hybrid OO language while small talk is one pure OO language where all entities are treated as objects. This in no way is a limitation of C++. As has been already stated, this hybridness is to provide a smooth shift from structured programming practices to OO paradigms. C++ absorbed structured programming capabilities of C and object manipulation capabilities of Simula.

The key advantage of objects is reusability and easier maintenance. An OO language emphasizes more on objects (Nouns in real world) rather than functions (Verbs). It is based on the principle of giving primary importance to data rather than functions, which manipulate these data. This is closer to the real world phenomenon of identification by objects (data/attributes) rather than by functions (actions/behaviours). However, functions are equally important, but not as important as the data, which they operate on. One of the key problems with structured or procedural programming is that programmers often tend to develop software from the scratch, or in other words, start afresh on every new project. This is almost reinventing the wheel where resources and time could be used effectively by employing already developed components directly or with little modifications.

Reusability feature of OO languages helps in developing software quickly (less coding effort involved) and without errors (bug-free) with the understanding that the reused components are foolproof and error-free. Reusability is well justified by the software engineering observation that mostly software costs are due to continuous evolution and maintenance. However, originality is as well a requirement. The advantage of writing one's own code is that users will be sure of the working mechanism of the code. When procedural languages have in them the concept of library support, why not extend this reusable feature to codes, which users create. It is in fact a combination of both these features that can aid in effective and efficient software development. Having dwelt on the evolution and need of OO languages, we shall from the subsequent section get to the programming environment involved with C/C++.

1.4 C/C++ Programming Environment

A C/C++ program goes through several phases before the end user (client) requirements are met. For a better understanding of these phases, we will consider that classical and naive example of displaying the string `Hello World` on the screen, an exercise which any novice is exposed to as he/she learns a language. To make things clearer we shall code the above task in C. The five line piece of code which performs the redirection of the string `Hello World` onto the screen is shown in Figure 1.1. (Numbers in the figure indicate the line numbers.)

This being a very simple example we have avoided the key algorithm and flow chart development step, which is the first step of any program development. An algorithm is a well-defined sequence of operations that transform the input to output. In this section, we proceed with

```
1. #include<stdio.h>
2. void main()
3. {
4. printf("Hello World \ n");
5. }
```

Figure 1.1: C code to print `Hello World`.

the assumption that the algorithm is readily available. A code is just a translation of operations specified in the algorithm into a format understandable by the language. In other words, it is exploiting the syntactic features of the language to achieve the desired input–output relationship.

The above code in Figure 1.1 though simple, has a lot of concepts involved in it and needs true interpretation for one to appreciate the programming environment. The various phases of program development are Editing, Preprocessing, Compilation, Linking, Loading and Execution. We shall correlate each of these phases with the above code.

1.4.1 Editor Phase

The editor phase takes care of accepting user keyed in characters which makes up the code and storing it in a file for further use. As the name implies the editor phase allows new entry or modification or editing the contents of the file. Thus, the input to the editor phase is user keyed in characters and output from it is a file, which is normally referred to as the source code file. Programming languages have restrictions on the file name extensions. A C source file should end with `a .c`; a C++ source file should be saved with `.cpp or .cxx or .C` (upper case) extension failing which the code never gets parsed or read. Two editors that are commonly used with UNIX systems are `vi` and `emacs`. Assuming that the above code is saved in a file called `hello.c`, we shall now proceed to the next phase.

1.4.2 Compilation Phase

Having fed in the user developed code the next phase ideally would be to check if the user has made use of the language features in the manner expected or what is referred to as syntactic checking should be carried out. Syntax checking or compilation is typically that phase of program development which checks if the user or programmer has confirmed to the grammar of the language while developing the source code. But even before we get into further details of compilation, we will have to resolve an intermediate and predecessor phase to compilation called preprocessor phase.

1.4.3 Preprocessing Phase

Having already established the fact that there is a certain extent of reusability (in terms of system code) made use of instructured programming, there should be some means of specifying how the already developed code will be reused as a part of user developed code. The preprocessor does this. The preprocessor program or block executes automatically before the

compilation and obeys/accepts commands (referred to as preprocessor directive) that direct what manipulations (including other files content, text replacements) need to be performed on the code even before it gets compiled. The # token in C/C++ is used to indicate that it is a preprocessor directive and needs processing before parsing. Now let us briefly correlate these concepts with code shown in Figure 1.1.

1.4.4 Correlation of the Various Phases

Line 1, `#include<stdio.h>` is a statement for the preprocessor to include the contents of the standard input–output header file (`stdio`). The `.h` extension indicates that it is more of a header file (system/predefined) rather than a user developed/frequent modification susceptible file. Header files need not always be system defined ones. Programmer can as well define header files with the diplomatic restriction that they contain only not susceptible or prone to modification code. The diplomatic rather than an exhaustive restriction is indicative of the fact that programmers can store frequently changing information in header files as well. But then this defeats the very purpose of header files as will be clear in a short while.

Header files are normally provided with only read permissions, at least system-defined header files. Programmers who distribute their header files would expect that this header content remains non-modifiable. Whether it is a system defined or user-defined header file, the client has only read permissions to avoid inadvertent and accidental changes to the file. The first line instructs the preprocessor to include the contents of the standard I/O header file that would contain information related to the standard input and output operations. The fact that it is a system-defined header file is denoted by the angled brackets (<>). User or programmer defined header files are included within double quotes (" ").

1.4.5 Header File Composition

The question that needs to be resolved at this stage is (i) what content will be stored in a header file, (ii) what effect will an header file inclusion have over the user-defined source code. Header files contain data and function declarations. This could be quite contrary to what most C programmers think that header files contain function definitions. Even before we get any further, let us differentiate function declarations from function definition. Function definition is specifying what the function has to do or activities/actions that a function needs to perform wherever it is being invoked. First time or at least non-C programmers may find this terminology of functions confusing.

Functions are basically syntactic provisions in a language to support the concept of modularized programming. Rather than developing the code in a flat-structured fashion, software engineering observation has been that it is wise off to develop code in a modularized fashion. Modularization in layman terms is the principle of dividing the initial problem into several smaller subproblems. Each subproblem is addressed separately and independently.

1.4.6 Code Modularization

As is the case with balance of nature that a divide strategy needs to be conquered at some stage or the other, the solutions of the subproblems are combined/integrated to solve the initial problem. In terms of a programming language problem definition, the simple task of

finding the sum of two numbers fed in by the user and displaying the sum on the screen can be modularized as shown in Figure 1.2. The modularization/division may look trivial here, but with a real project one will be able to appreciate the true need for modularization in favour of flat-structured coding.

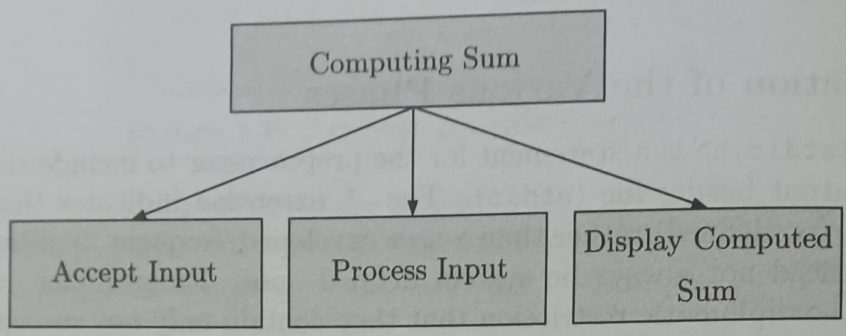

Figure 1.2: Example for modularization of coding.

Modularization apart from the fact that it helps in addressing the problem on hand at lower complexity levels, offers several other key advantages. The primary of them is error localization. It is lot more easier to trace errors as one can identify the subproblem or block that causes errors rather than addressing it as a single large and flat block. The other advantage is the fact that repetitive tasks can be coded easily. For example, if the same subtask of computing the square root of a number is required or made use of more than once, then the user needs to define the solution to the subproblem only once and further on make use of that solution how many ever times he/she wishes. This concept of defining a solution to the subproblem is referred to as function definition assuming or equating functions with subproblems. We started out with the need to differentiate function declarations from function definitions.

As a part of code Figure 1.1, we have made use of one predefined (system defined) function called `printf`, line 4 (expanding to print function) to aid us in the task of displaying the string hello world on the screen. `printf()` is a predefined function that redirects the string or value to be displayed on to the standard output device that is normally the monitor or the screen. There is a certain extent of modularization even in this code. `printf()` is a solution to a subproblem that is integrated or conquered in the initial or the main problem. Thus, the syntax of a mandatory `main()` function in C/C++.

All C/C++ programs need to mandatorily have a function called `main()` defined in them. All user or programmer specified instructions which could be simple instructions or functions (subproblems) by themselves should be contained in the main definition. In this case, the main function makes use of another called `printf()` to aid in the process of displaying. This syntax is often referred to as `main()` calling or invoking `printf()` function. When a function is called transfer of program control takes place from the calling function to the called function. In this case, the calling function is `main()` and the called function is `printf()`.

Once the function call is completed, program control is returned to the next instruction after the function call in the calling function. As there is transfer of program control, data or information within the calling function may not be accessible or visible for the called function. Hence, the justification for functions to accept and return arguments. Information, which the called function needs to get from the calling function, is passed to it and information, which

the calling function needs from the called function, is returned to it by the called function. We shall at a later stage deal with argument passing and functions in greater detail in Chapter 3.

Getting back to where we started from, the `(<stdio.h> header file)` will contain function declarations only. To be precise it contains the declarations of functions related to input-output tasks. A function declaration is additional information for the compiler. It is a mechanism of informing the compiler that further down the line in the source, a specific function with such a name may get defined. For example, the summing task, assuming the function sum will accept two arguments and return the computed sum, the following header specification `int sum (int a, int b);` is referred to as the declaration of the function sum. sum is the name of the function or what is referred to as function identifier. The two values `a` and `b` passed to it are referred to as arguments and in this case they are of data type integer. We have already established the need for arguments passing to overcome that visibility or scope effect between calling and called functions. Function declarations are also referred to as prototypes or signatures (excluding the return type). The template set of `return_type fn_idifier` (argument list) is referred to as function declaration. The names a and b are optional as they serve only as placeholders. It is the data type and the number of arguments that is more crucial to function declaration. This is based on the fact that one cannot always enforce the restriction that a function gets called with a specific integer variable (e.g. `a` only or `a` always). The concept of associating functions with data is more a feature of C++ (OOP). The definition of function would be as follows:

```
int sum ( int a , int b)
{
return a+b;
}
```

1.4.7 Compiler's Treatment of Functions

The way the compiler differentiates between function declaration and definition is that in function definitions; the function header is not semicolon terminated whereas function declaration is always semicolon terminated. Thus, function declarations contain information regarding the number of arguments and the individual data types of each argument and not on how the function performs the task [which would be available in function definition]. Having differentiated function declaration and function definition, what purpose would function declarations serve? It is in fact added headache or burden for the compiler to remember additional detail.

Function declarations are a mandation in C++ but optional in C. In such case, the most recent question needs to be definitely answered. Functions need not always be called with the correct number and data type of arguments. Such function calls are referred to as mismatched function calls. Functions getting called with incorrect number of arguments or incorrect data type of argument will cause serious run time errors. Such function calls are allowed to proceed if function declarations are not provided. The function definition gets executed with manipulation (assumptions) or truncations (round off) to the extent possible. But most often they lead to undesired outputs or outputs not expected by the end user. This undesiredness is a result of the mismatched function calls.

The problem lies in the fact that such mismatched function calls rather being resolved at run time (it is difficult to locate them in the first place), it is better off to catch such errors at the compilation stage itself. This is typically the purpose of function declarations. They catch or handle errors on the fly as and when they encounter rather than resolving it at run time which would lead to resource wastage (time and space) in the fact that all previous operations prior to mismatched function call needs to be undone and executed all over again. Function declarations allow such errors to be caught at the compilation stage itself. Thus, function declarations or signatures catch mismatched function call errors at an early stage and thereby avoid resource draining.

Having justified the need for function declarations let us get back to the concept of header file inclusion. The # include preprocessor directive instructs the preprocessor to paste the contents of the respective header file before the definition of function `main()`. It is more a logical pasting or splicing rather than physical paste or inclusion of source code. Hence revisiting the source [visiting the source code after one compilation] will not reflect this inclusion of source code. But then effectively the source code length or the lines of code is logically increased after the preprocessing is over. The header file `<stdio.h>` was included because the programmer required `printf()` function to display the string `hello world`. `<stdio.h>` contains declarations of standard input and output functions such as `print()` and `scanf()`.

As we have already stated compilation phase checks for syntax errors to check if the programmer has confirmed to the language's grammar. Assuming that the code that is being compiled is syntactically error-free, the output of the compiler phase is an object `(.obj)` file. The `obj` file is the translated machine understandable format of the programmer-specified instructions. To be more precise, it is a sequence of 1's and 0's that correlate with the programmer-specified instructions that the machine can understand. The code shown in Figure 1.1 was stored in the file `hello.c`. This when compiled generates its corresponding machine interpretable version which is the file `hello.obj`. The `obj` file name is self-explanatory of the fact that it is the equivalent machine understandable instruction sequence of the instructions explicitly specified by the programmer.

We have already established the fact that `printf()` definition is not specified by the programmer. We included the associated header file to get to know about the nature of functions (system-defined) that may get invoked as a part of this source code. Header files do not contain function definition. The reasoning shall be explained in a short while. Another issue to be resolved. How does the function definition then get resolved or in other words how does the function `printf()` get called/executed properly, if its definition is not stored in the header file that has been included? This is precisely taken care of by the Linker phase. The compiler during the syntactic error-checking phase searches for the function definition in the same source code.

For functions that have declarations in the user-specified source code but not definitions, the compiler defers the function resolution onto the next stage (Linker). The compiler passes a token/information of all those function's for which it could not locate the function definition. The linker now takes charge and searches the standard system directory [place where the C/C++ software is installed] for the respective definitions. This standard system directory could be `C:\TC\SYS` on an MS-DOS platform or `\usr\bin` on a UNIX/LINUX platform. The dictionary meaning of the word link is to merge or combine. In this case the task of the linker is to merge or combine intermediate output or object versions and generate a final executable or output file. This final executable would carry an .exe extension in a DOS platform or a.out

file in a UNIX platform. Hence there is a sequence of transformations required to transform the object version of the user-specified instructions to the final executable file.

1.4.8 Linking Phase

The standard system directories do not contain the source codes of the various predefined or system-defined functions. Instead they contain the respective object versions of the predefined functions. Again this has some reasoning behind it. Developers of C have taken care of completing the definitions of predefined functions such as `printf()`, `scanf()` in corresponding files namely `printf.c`, `scanf.c`. As has been the syntax to compile source codes and generate binary versions, developers of C have compiled the predefined functions source codes and generated the respective `obj's` or `printf.obj, scanf.obj` are made readily available to the end users.

Thus, the job of the linker is to look out for the object files for functions (predefined) that have been made use of in the programmer-specified code. The linker after having located the object versions integrates the various object files and the object file of user-specified code (`hello.obj`) in this case, to generate the final executable versions (`hello.exe or a.out`). This linking is required because irrespective of the fact whether they are system or user-defined codes all instructions should operate in an integrated fashion towards achieving the initial objective or problem definition that one started with. Linking operation for better understanding can be expressed as a mathematical formula of the form:
`Hello.obj+printf.obj = hello.exe [DOS] and a.out [UNIX/LINUX].`

1.4.9 Loader Phase

We have almost reached the end of our discussion on header file inclusion. The final phase of program development is the Loader. Students or novice programmers are very much affined to this terminology of program runs. The program runs from which location would be the next naive question to be answered. The program runs or executes from the primary memory or the RAM. The loader performs the operation of transferring machine instructions sequentially from the secondary storage to the primary memory. In other words, the loader loads the executable file on to the RAM for execution.

We are still left with one more intriguing and interesting question with respect to header file inclusion. Why should header files contain function declarations and not function definitions? In fact, in our earlier discussion we had highlighted that this is only a diplomatic and not an exhaustive or syntactic restriction. Header files can contain function definitions as well, but generally not advisable from software engineering point of view. We shall justify this principle now. If at all header files were to contain function definitions, then the preprocessor directive as a part of its inclusion process paste the contents of the header file along with the function definition, after which the included and user-specified code gets compiled.

Thus, violating the earlier diplomatic restriction in no way hinders the generation of the executable version. It was due to this that we referred it to be a diplomatic (graceful/decent) restriction. But it throws up a serious issue. The included function definition also getting compiled, means that compiling a user-specified code takes sufficient time (additional compilation time as a result of compiling predefined functions code). This is at a major cost! Why should one compile code that has been already compiled and readily made available as `obj's`. Thus,

placing function definitions in header files increases the compilation time or causes compilation time overhead, hence the justification. A pictorial representation of the various phases of program development is shown in Figure 1.3 for quick and easy understanding.

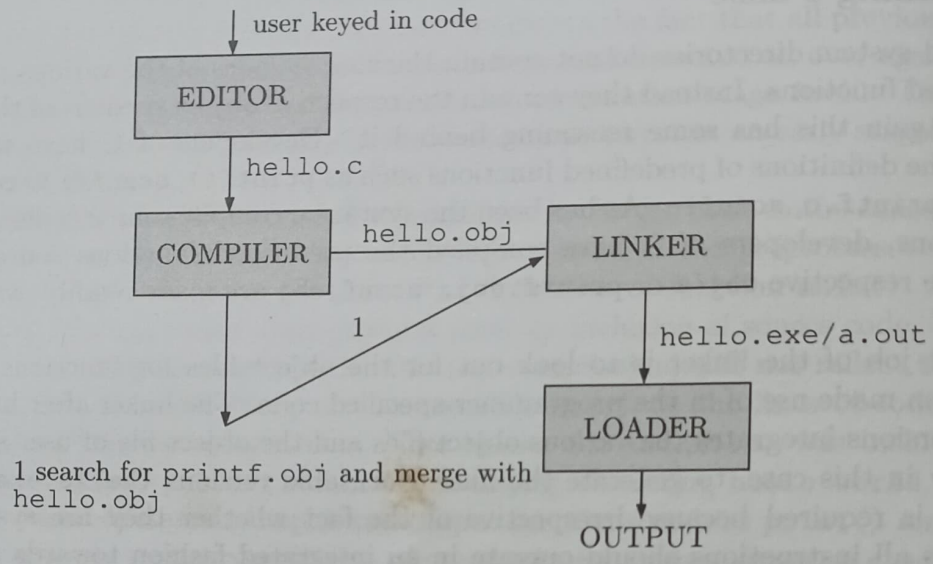

Figure 1.3: Phases of program development.

The above discussion may seem overexhaustive. But to be frank with the readers, the concept of header file inclusion has been abstract for quiet long time. The author's efforts have been to resolve this abstractness or ambiguity. From the next section onwards we shall explore the main features of structured programming.

1.5 Summing-up Two Integers

Let us now consider an example to find the sum of two integer numbers entered by the user via the keyboard. The C code to add two integers is shown in Figure 1.4.

```
1. #include<stdio.h>
2. int main()
3. {
4. int number1,number2,sum=0;
5. printf("Enter the first number \n");
6. scanf("%d",&number1);
7. printf("Enter the second number \n");
8. scanf("%d",&number2);
9. sum=number1+number2;
10. printf("The sum of %d & %d is %d",number1,number2,sum);
11. return 0;
12. }
```

Figure 1.4: C code to add two integers.

1.5.1 Code Interpretation

Line 1 includes declarations of `printf()` and `scanf()`. Line 2 shows the definition of function `main()`. `main()` is a compulsory or mandatory function within which the solution to the problem being solved is expressed according to C/C++ syntax. Any statement (excluding declarations) which the user wishes to be executed should belong to `main()`. This is based on the reasoning that `main ()` function would solve the problem on hand viewing it as a single large or several subproblems. The `int` tag before `main()` is the return type of function `main()`.

Experienced C programmers might be aware of the syntax of functions returning values. User-defined functions that return values had been mentioned in the earlier section. The returned value actually gets caught or stored or returned by or in or to a local variable of the calling function or a global variable. This is usual. But what significance would a `main()` function returning values have? Where would the value get returned? We have already established that `main()` is one function which gets called automatically (once) by the compiler. Had it been user-defined calls one can assume that the value returned by `main()` is stored in a variable. But what happens with the default call? The value returned by the default call of main is used to interpret whether the program execution was successful or not.

A positive integer returned by the default call of main signifies successful or normal program termination. (The compiler was successful in going through the entire block of statements). A negative integer returned indicates abnormal termination. In such cases, further course of action and clean-up [memory/resource release] should be performed. We get back to the phases of program development; the last phase of loading loads the executable file onto the RAM. This is performed as a command [name of exe file and then return/enter pressed by the user]. This is referred to as the command prompt [$—UNIX command prompt, C:—DOS command prompt].

Hence it is from command prompt that phases of compilation, load and execute are performed. The program in execution is nothing but the `main()` function being called. Thus, it is to this command prompt that `main()` returns either the positive or the negative integer for successful or abnormal program termination to aid in further course of action. C programmers would have been exposed to the syntax of `main()` returning void data type normally. It is a good programming practice to make function `main()` return an integer, [in fact, the default return of type of functions is integer].

Line 3 is the beginning of function definition and Line 4 comprises the declaration and assignment statement. Data that is to be manipulated in the code should get distinguished. For the above task, the programmer needs to differentiate two members. Further values entered by the user should be stored for a considerable duration [temporary or permanent] for future reference, hence the need for declaration statement. A declaration statement in C/C++ solves this purpose. It assigns storage or memory locations and names for values that shall be entered by the user or created in the program for further manipulations. Data values encountered in a program or code could be whole numbers (integral entities) or characters or fractions.

Thus any programming language has provisions to accommodate the most commonly used data types. The above code manipulates only on integer data types and the keyword int is used to declare variables of integer type. We shall towards the end of this chapter deal with declarations in detail. For the moment readers need to be clear of the fact that variables being declared in a program is assigning name and memory location to store the value associated

with the variable. It is referred to as a variable to denote the fact that such entities keep changing or varying and do not remain constant. (Another data type available in C/C++).

A variable that is declared can be assigned values by an input statement (user assigns values at run-time) or by an assignment (=) statement of the form `a=5`. This process of declaring and assigning values to variables in a single statement is referred to as a declarative assignment statement. A variable that is declared and assigned values in one statement is also referred to as an initialization statement. `[sum=0]`; The variable sum initialized with a value `0`.

Lines 5 and 7 are referred to as prompt statements. The overall task can be achieved even without these prompt statements. But then for interactive input–output such `printf()` statements guides the user in providing the correct input. The string that would act as a guiding statement is passed as an argument to the `printf()` function. In this case, the two prompt statements made use of enter the first number and enter the second number. This is purely for guided input–output. The `\n` used here also referred to as a new line character is grouped under the set of Escape Sequences. These are special characters that perform specific input–output related functions. The new line escape sequence positions the cursor on a new line for further displays. The `\n` is positioned in the `printf()` whenever the user wishes to start redirecting or printing from a new line. There are a host of escape sequences that shall be explained briefly in this section.

Lines 6 and 8 make use of input functions [`scanf` expands to `scan` function that scans or reads input]. `Scanf` is used to scan for or read values or accept input from the standard input device, normally the keyboard. The `%d` is the format specifier that identifies to the `scanf` function the data type of the value that is to be read in. The `d` in this case expands to decimal (number system) or the integer data type. The other format specifier commonly used is `%f`—floating point, `%c`—character.

Format specifiers will be dealt with in detail later. The `&number1` delimited by a comma operator within a `scanf` function represents the address where the accepted value is stored. Earlier, we mentioned that a declaration statement assigns memory locations for variables and `&number1` precisely returns the address of the variable `number1` and it is in this location that the value read in is stored. The `&` (ampersand) operator is referred to as the address-of-operator.

Line 9 is pretty self-explanatory and adds up the values of `number1` and `number2` and stores (assigns) it in the variable sum for future reference. Line 10 displays the computed sum. The 3 format specifiers get replaced with the values of `number1, number2` and sum when the redirection to the screen is performed in the same order as they occur in the `printf()` call. Line 11 returns an integer to denote successful execution as explained earlier. Line 12 is the end of function definition (`main()`).

1.6 Escape Sequences

The other escape sequences available with C and C++ are as shown in Table 1.1[called so because such sequences escape user control or perform predefined operations].

In the above table, `\t` provides one tab space on the screen in the horizontal direction [8 spaces normally]. `\v` provides one tab space in the vertical direction (in other words combination of `\n` and then `\t`). `\r` positions the cursor on the first position [home] of the current line, not advancing to the next line as is done with `\n`. (Refer code shown in Figure 1.5). `\a` sounds a beep or alert using the inbuilt miniature speaker.

Table 1.1: Escape sequences

Escape sequence	*Operation*
\t	Horizontal tab
\v	Vertical tab
\a	Audible alert
\r	Carriage return
\\	to print backslash
\" "	To print double quotes

```
1. #include<stdio.h>
2. int main()
3. {
4. printf("Hello World \r");
5. printf("World");
6. }
```

Output:

World World [DOS SYSTEM]

World [UNIX]

Figure 1.5: C code using escape sequences.

The string `World` redirected in the second `printf()` call at line 5 overwrites the string `Hello` that would have been redirected as a result of the `printf()` at line 4. However, at the end of this `printf()` there is an `\r` application and hence the cursor after redirecting the string `Hello World` onto the screen would be positioned on the home of the same line. The second `printf()` call thus overwrites the string `Hello` (redirected in the first `printf()` call) The final output as shown above is OS dependent.

The intended output of `World World` occurs only on DOS platforms (or) when used with C/C++ compilers developed for DOS platforms such as TC++, TC [Turbo C++/C compiler]. In LINUX/UNIX based compilers, as the cursor backtracks, the characters in the video buffer or memory get erased leading to the undesired output of only `World` on the screen. The second-half actually would have got erased as the cursor backtracks due to the application of `\r`. This is infact a bug with the CC compiler made available with LINUX systems.

1.7 Arithmetic in C/C++

Most programs perform arithmetic calculations. The operators listed in Table 1.2 are called as binary operators [operators that accept or operate on the operands or inputs]. Asterisk (`*`) performs multiplication, Slash (`/`) indicates division, `%` denotes modulus [remainder on division], `+` (addition), – (subtraction). The division operator when used with two integers discards the fractional part of the quotient. The modulus operator yields the remainder on integer division. [7% 4 results in 3].

Table 1.2: C/C++ arithmetic operators

C/C++ operation	*Arithmetic operator*	*Algebraic expression*	*C/C++ expression*
Addition	+	a+b	a+b
Subtraction	-	a-b	a-b
Multiplication	*	ab	a*b
Division	/	a/b	a/b
Modulus	%	a mod b	a%b

Arithmetic expressions in C/C++ must be entered in straightline form. Algebraic expressions such as a are not acceptable to compilers. Parentheses in C/C++ are used in a manner similar to algebraic expressions. The operators in arithmetic expressions are processed by a predetermined order or what is referred to as operator precedence. The rules of operator precedence are as follows:

- Operators in expressions contained within parentheses are evaluated first. Parentheses can be used to enforce the programmer-desired evaluation given an arithmetic expression to be processed. They are at the highest level of precedence. Nested or embedded parentheses scenarios are handled by processing the innermost parentheses pair first and so on.
- Multiplication, division and modular operations are performed next. Expressions involving several * or % or / operators are evaluated left to right. All these operators occur at the same level of precedence. Addition and subtraction are performed next. Other features explained above are applicable here as well.

For better understanding of the above rules let us now look at the way the following expressions are processed as shown in Figure 1.6.

(1) Algebraic expression is

```
a = bc mod d + e/f - g
```

Equivalent C/C++ expression is

```
a = b * c % d + e/f - g
    6   1   2   4   3   5
```

(2) Algebraic expression is

$$y = ax^2 + bx + c$$

Equivalent C/C++ expression is

```
y = a * x * x + b * x + c
    6   1   2   4   3   5
```

Figure 1.6: Examples for expression evaluation in C.

1.8 Decision-making

C/C++ programs in their course of execution choose to differ from the normal sequential flow, which is often governed by certain conditions becoming either true or false. Based on

whether the condition is satisfied or not a specific or associated set of actions is performed. The construct in C/C++ which helps in decision-making is referred to as control structure and is considered in greater detail in the next chapter. A key requirement as a part of the decision-making process is condition specification and C/C++ supports a host of relational and equality operators to frame conditions involving variables.

1.8.1 Equality or Relational Operators

This section introduces the `if` control structure available in C/C++. The `if` control structure allows a program to make a decision based on the truth value (true or false) associated with some condition. If the condition is satisfied the block of statements or statement within the if part is executed. If the condition is not met (false), then the else part if defined is executed. Otherwise, it simply skips the `if` block. The `if` structure looks as shown in Figure 1.7. A detailed discussion on the `if` and other selection structures in C/C++ is considered in the following chapter. The conditions of the `if` structure can be formed by making use of the

```
if (some condition)
{
⋮
}
else
{
⋮
}
```

Figure 1.7: `if` control structure.

equality and relational operators. Relational operators are at a higher level of precedence in comparison with equality operators. All relational or equality operators have same precedence level and associate left to right. The following Table 1.3 depicts the various relational and algebraic operators of the C/C++ operator set.

Table 1.3: Relational and equality operators

Algebraic operator	*C++ operator*	*C/C++ condition*	*Meaning*
Relational			
>	>	a>b	a is greater than b
<	<	a<b	a is less than b
≥	>=	a>=b	a is greater than or equal to b
≤	<=	a<=b	a is less than or equal to b
Equality			
=	==	a==b	a equal to b
≠	!=	a!=b	a is not equal to b

Common programming errors that can occur with relational or equality operators that never get caught at compilation stage are: `a!=b` if used as `a =!b` will not be a syntax error while it will definitely be a logical error (the not of `b` is assigned to `a`), whereas the intention

of the programmer would have been to check if values of a and b are not equal to one another. That is, the comparison statement (if `(a=!b)`) proceeds as an assignment statement and might cause serious logical errors.

Confusing the equality `(==)` operator with the assignment operator will again cause logical errors as discussed above.

1.9 Operator Precedence Table

Table 1.4 depicts the precedence and associativity of the various operators of C/C++ that the reader has been exposed to. The stream insertion and extraction operator used for input output in C++ will be dealt with in detail later.

Table 1.4: Operator precedence table

Operator	*Associativity*	*Type*
`()`	Left–Right	Parentheses
`*,/,%`	-do-	Multiplicative
`+,-`	-do-	Additive
`<<,>>`	-do-	Stream insertion/extraction (will be explained later)
`<,<=,>,>=`	-do-	Relational
`==,!=`	-do-	Equality
`=`	Right–Left	Assignment

Review Questions

1. Identify the various units that make up a computer.
2. Set of instructions that process data and result in transformation is called ———.
3. What are the logical units of C and C++ programs?
4. Appreciate the need for object-oriented programming in comparison to structured programming.
5. Explain the effect of `\r` and `\a` escape sequence.
6. Name a few C and C++ compilers available for Windows and Unix/Linux platforms.
7. What is the program that combines the output of the compiler with other library functions to generate the final executable?
8. Differentiate function declarations from function definitions.
9. Appreciate the inclusion of header files and their composition.
10. What is the advantage of having function declarations?
11. Develop a simple C++ program to accept two integers from the keyboard and display the sum, product and difference of the two.
12. Develop a simple C++ program to convert a user input Celsius temperature to its equivalent Fahrenheit value.

13. Develop a simple C++ program to swap values in two integer variables (i) using a temporary, and (ii) without using a temporary variable.
14. Develop a C++ program to compute the area and circumference of a circle, given as input the radius. Assume value of $\pi = 3.14$.
15. Identify the order of evaluation in C/C++ of the following expressions:
    (i) $z = x * x + y/7\text{-}3$, and
    (ii) $x = y * y/2 * 3 + z\%6$
16. Develop a C++ program to accept an integer as input and display whether it is an odd number or not.
17. What do you understand by the term compile time error?
18. Differentiate syntax from semantic errors.
19. Appreciate the naming convention behind C++.
20. Name a few editor programs available in Windows and Unix/Linux platforms.