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.

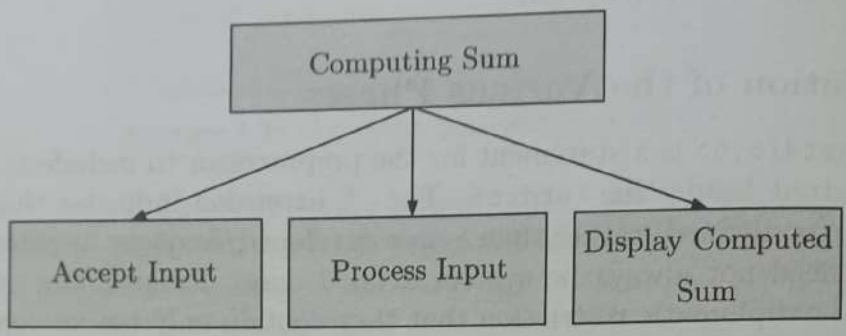

Figure 1.2: Example for modularization of coding.

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.

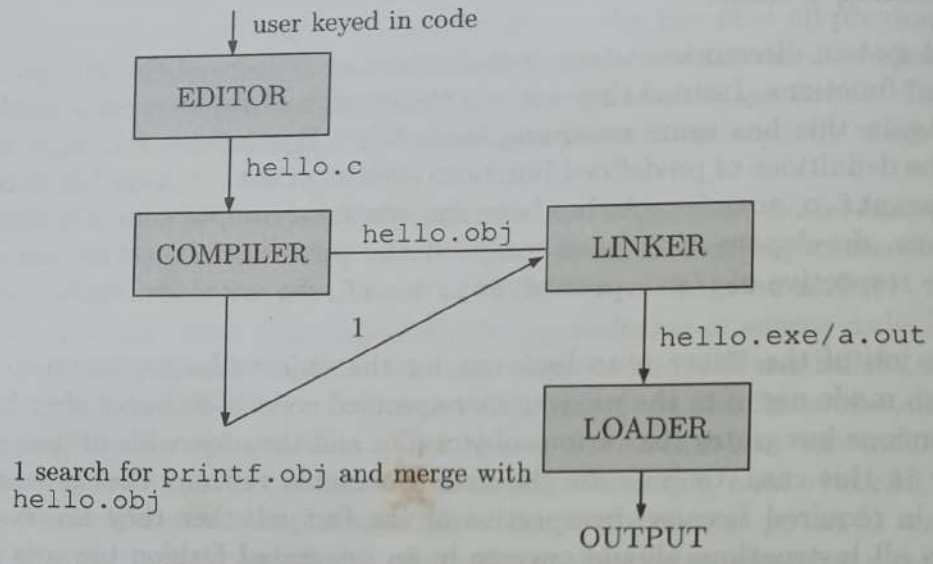

Figure 1.3: Phases of program development.

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`
$\geq$	`>=`	`a>=b`	`a` is greater than or equal to `b`
$\leq$	`<=`	`a<=b`	`a` is less than or equal to `b`
Equality			
$=$	`==`	`a==b`	`a` equal to `b`
$\neq$	`!=`	`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.

CHAPTER 2

Control Structures

Prior to program development for a particular problem, the developer should have a clear understanding of the problem definition and a planned approach towards solving the problem on hand. In the following section, we shall describe algorithms and pseudocodes that are prime requirements for program development.

2.1 Algorithms and Pseudocodes

Computing problems can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of the actions to be executed and the order in which these actions are to be executed is referred to as an algorithm. To formally define algorithm, it is a sequence of well-defined computational steps that needs to be executed in some order to transform the user input to the desired output. It can be viewed as a transformation function. Specifying the order of executions of the various statements that make up a program is called as program control. A pseudocode is an artificial and an informal language that helps programmers develop algorithms.

Pseudocode is similar to everyday English. It is convenient and user-friendly and is not an actual programming language. Pseudocode is not executed on computers. They help the programmers to logically think out a program before writing or developing the program confirming to the language. Pseudocode should include only executable statements [with the program in mind]. Variable declarations that instruct the compiler to reserve memory space, those that does not cause any input or output action during program execution should not be included in pseudocode.

2.2 Control Structures

Generally, program statements are executed sequentially [one after another], also referred to as sequential program execution. Execution need not be always in sequence, the next statement to be executed can be a statement that is not the next in sequence. This is referred to as transfer of control. Programs are normally developed with three control structures, namely the sequence, selection and repetition structures. The default mode of execution is sequential. However, this sequential flow can be altered with the remaining two control structures available.

Algorithms [text-description] and flow charts [pictorial representation] are normally used in an interchanging fashion before program development. Both of them address the issue of chalking out a solution strategy for the problem on hand in an an easy-to-understand format.

not restricting oneself to the programming language features. It is one among the two that is chosen as the first step of program development. For the convenience of beginners we shall towards the end of this section describe the algorithm, flow chart and C/C++ code for a particular problem. There are three types of selection structures available in C/C++ as follows:

- The `if` selection structure either performs [selects] an action if a condition is true, or skips the action if the condition is false. This structure is also referred to as single selection structure.
- The `if-else` selection structure performs an action if a condition is true [if part], and performs a different condition action if the condition is false [else part]. This structure is also referred to as double selection structure.
- The `switch` selection structure performs one of many different actions possible based on the value of an integer or integer expression. This structure is also referred to as multiple selection structure.

C/C++ provides three types of repetition structures, namely `for`, `while` and `do-while`. Each of these words is called a keyword [interested readers refer to Appendix for a list of C/C++ keywords]. Keywords should be made use [typed-in] the same manner as they appear in the keyword list, failing which would cause syntax errors. Thus, there are in all seven control structures, namely `sequence`, `if`, `if-else`, `switch`, `for`, `while`, `do-while`. These control structures can be manipulated [made use of in different or varying combinations of one another] to implement the algorithm.

Programs developed using control structures usually are single entry and single exit based. Control structures may be connected with each other [the exit point of one control structure is connected to the entry point of the next control structure]. This is nothing but control structure stacking [referring to the accumulation of control structures one after the other, top be literal, it is stacking of control structures one above the other]. The only other form of manipulation control structures is nesting them or placing one control structure inside or within another structure. This is referred to as control structure nesting. This chapter will be dedicated to discuss these control structures and possible ways of manipulation in an exhaustive manner with easy-to-understand examples.

2.2.1 Selection Structures in C/C++

The `if` Selection Structure

A selection structure is required to choose one among many alternative courses of action. For example, suppose that the passing marks in an exam is 45, then the pseudocode statement would be: *if student's marks is greater than or equal to 45*, then *display the student has passed in the examination.*

The above pseudocode statement determines if the condition [mark greater than or equal to 45] is TRUE or FALSE. If the condition is true, then the string, the student has passed in the examination, is displayed on the screen and then the next pseudocode statement is performed. If the condition is FALSE, then the display statement is ignored and the next pseudocode statement in order is performed. Readers kindly note the space left before the

display statement. It is referred to as indentations and should be done with source codes. It is a good coding practice to indent control structures. They promote program readability and clarity. The compiler ignores such blank spaces. Even though it is not a syntactic mandation to indent control structure, the author shall take the readers with the confidence to stick on to the diplomatic syntax of indenting control structures. The above pseudocode statements when translated in C looks like

```
if (stud_mark >= 45)
printf("The student has passed in the examination \n");
```

As we have already explained, the condition within the `if()` is checked. If the condition evaluates to TRUE (non-zero value), then the action specified within the `if` is performed [the corresponding `printf` statement gets executed]. Note the indentation that has been provided for the `printf` statement, this clearly identifies that it belongs to the `if` part. Indentations do not cause any performance enhancement. It is just to improve program clarity and for better understanding. The equivalent flow chart representation for the `if` selection structure is as shown in Figure 2.1.

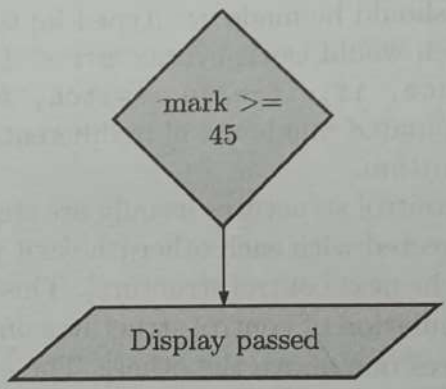

Figure 2.1: Flow chart for the `if` selection structure.

The flow chart consists of an important flow-charting symbol [diamond-shaped] called the decision symbol, used to pictorially represent the decision-making process. To review the other most commonly used flow chart symbols. Oval symbol with start or end labels within it represent the beginning or start and termination or end of an algorithm. [.] Circle symbol used as page connectors, used when a flow chart exceeds within a page limit. [A—made use of in the previous page as last symbol and in the subsequent page as first symbol.]

A parallelogram symbol is used for representing input and output statements [Read Mark]—flow indicator or sequence indicator. Down arrows represent the control flow from the entry till the exit point. Rectangle symbol to represent processing statements or actions `[sum = a + b]`.

We have already stated that the condition of a control structure is more of an expression constructed using relational or equality operators. The decision is made based on the value of the expression; an expression that evaluates to a non-zero value is treated as true value. A zero value is treated as false.

C++ provides the data type Bool to represent values of true or false. This is not available with C.

if-else Selection Structure

The if selection structure discussed earlier performs the required action only if the condition is true, failing which it is skipped. There is no way of specifying an alternate course of action if the condition is false. It is provided in the `if-else` construct. The same pseudocode example taken earlier when extended and coded with the `if-else` construct looks in C like

```
if (std_mark >= 45)
printf("Passed \n");
else
printf("Failed \n");
```

An additional conditional operator available in C/C++ is `?:` [ternary operator]. The operator takes three operands. The operands together with the conditional operator form a conditional expression. The first operand is a condition, the second operand is the value for the entire conditional expression if the condition evaluates to true, and the third operand is the value when the condition evaluates to false. For example, the statement `result = (mark >= 45 ?Passed:Failed);` assuming that the result variable is declared as a string, the condition (expression 1) `mark >= 45` is evaluated. If it evaluates to true, then expression 2`[Passed]` is evaluated and the result of this evaluation is assigned to the LHS variable result. Else expression 3`[Failed]` is evaluated and its value is assigned to the result. In this case, expressions 2 and 3 are more of values rather than conditional expressions. Thus, depending on the mark either or the string `passed` or `failed` is stored in the result variable. The value in a conditional expression can also be statements or actions to be executed. For example, the above statement can be remodelled as follows:

```
mark > = 45 ?  printf("Passed"):printf("Failed");
```

As explained above, the condition `mark >=45` is evaluated. If the condition evaluates to true, then the condition, in this case, the action of redirecting string `Passed` to the screen is executed. If the condition evaluates to false, then the string `Failed` is redirected to the screen.

Nested `if-else` structures test for multiple cases by placing a `if-else` selection structure inside another `if-else` selection structure. The following piece of code carries out 3 conditional checks on mark and displays the corresponding statements:

```
if (mark > = 90)
printf("Excellent \n");
else
if (mark >= 70)
printf("Average \n");
else
if (mark >= 45)
printf("Passed \n");
else
printf("Failed \n");
```

The above style of indenting may cause the programmer to run out of space on a line and single line source codes to be split across multiple lines on the editor window. To avoid this, an equivalent representation shown below can be used.

```
if(mark >= 90)
printf("Excellent \n"); ⇒ 1
else if (mark >= 70)
printf("Average \n") ⇒ 2
else if (mark >= 45)
printf("Passed \n"); ⇒ 3
else
printf("Failed \n"); ⇒ 4
```

If mark is greater than or equal to 90, then 1 is executed and the remaining statements are skipped. If `mark >= 90` evaluates to false, then the second condition `mark >= 70` is checked. If this evaluates to true, then 2 is executed. The pattern follows similarly. The last `printf(4)` is executed when all the preceding conditions evaluate to false.

Note: A nested `if-else` structure can be much faster than a series of single selection, if structures because of the possibility of early exit after any one of the conditions are satisfied. Hence, it is a good programming practice to test for conditions that are likely to be true at the beginning of the nested `if-else` structure enabling faster execution and early exit than will testing infrequently occurring cases first.

The `if` selection structure expects only one statement in its body. To include several statements in the `if` body or `else` body, the statements should be enclosed between open and close curly braces [{ }]. A set of statements enclosed with a pair of braces is called as compound statement. This can be interpreted as forming a block. An example for `if-else` selection structure with compound statements is as follows:

```
if (percentage >= 45)
{
printf("Passed \n");
printf("Can be promoted to higher grade \n");
}
else
{
printf("Failed \n");
printf("Should repeat the course \n");
}
```

Depending on the value of the condition, the two statements that form the `if` or the `else` block are executed. Statements that are not enclosed within braces but succeed the `if` or `else` block will get executed always irrespective of the condition value [based on the sequential flow of program control].

Common programming errors that are possible with selection structures when coded in C/C++ are as follows:

Missing out on one or both the braces of a compound statement may cause syntax or logical errors in the program. Placing a semicolon after the condition in a single selection structure causes logical error and syntax error in double selection structures.

To avoid such errors it is a good programming practice to type the braces of a compound statement first even before the individual statements comprising the block are keyed in. This

prevents accidental omissions of braces. A compound statement can contain declaration as well, in which case only the compound statement is referred to actually as a block. The syntactic compulsion in C is that all variable declarations should appear at the beginning of the block, before any executable statement of the block [main() or any other block]. However, C++ does not impose this syntactic restriction, it allows what is called on the place variable declarations or declaration of variables whenever they are needed is allowed. That is to say variable declarations and executable statements can alternate in C++.

2.3 The while Repetition Structure

A repetition structure allows the programmer to specify that an action is to be repeated on some condition being true. Thus, it is conditional repetition that is performed more often with repetitive control structures available in C/C++. The syntax for the while structure is as follows:

```
while (condition)
{
.
.statements
.
}
```

The list of statements mentioned within the while block should at some stage or the other make the condition of the while block to evaluate to false failing which it causes infinite looping or looping for ever.

Example:

```
counter = 2;
while (counter <= 10)
{
printf("Example \n");
printf("while - loops \n");
counter =counter + 2;
}
```

In this case, the repetition is carried out five times. The condition to be satisfied for repetitive execution is counter <= 10. This being true the statement or block of statements that forms the while loop construct is executed repetitively. As we have stated earlier, the statement counter = counter + 2 takes care of modifying the counter values [each time incremented by two]. At some stage or the other counter value would exceed 10, and this serves as the terminating condition for the repetitive structure. The count of 5 repetitions is arrived at as shown in Table 2.1.

Program execution resumes with the first statement after the close brace of the while block [sequential execution]. The flow line emerging from the rectangular block wraps back to the decision [condition check] that is tested each time through the loop until the condition becomes false.

Table 2.1: Example for counters

Counter value	*Count of repetitions*
2	1
4	2
6	3
8	4
10	5
12	Loop terminated

2.3.1 An Example Program

Problem: A class of 20 students took an exam on C programming. Compute the class average for the above course.
Solution: The class average is the sum of the marks divided by the number of students. The algorithm for solving this problem must accept the marks [input], perform averaging [processing] and then finally display the result [output].

The pseudocode for the above problem is as follows:

1. Set total to zero
2. Set markcounter to 1
3. `while` markcounter is `<= 20`
4. Input next mark
5. Add this mark to total
6. Add one to markcounter
7. Set class average to total divided by 20
8. Print the class average

This form of repetition is called counter-controlled repetition or definite repetition since the number of repetitions is known well in advance. The C source code for the above algorithm is shown in Figure 2.2.

Note that all the variable declarations precede the first executable statement of the main block. The average variable can also be declared as float to reflect floating-point averages. Variables for whom values will not be entered by the user but those that will be made use of in mathematical calculations or those that will be referenced later should be initialized to valid starting point values. In this case, the variables markcounter and total are initialized to 1 and 0. If such variables are not initialized they may contain values that were assigned their memory locations as a part of a previous program in execution. Such variables called automatic (auto) variables, get initialized with JUNK or GARBAGE [junk values if not user initialized]. Storage classes will be dealt with in detail in Chapter 3.

Note: In a counter-controlled repetition, because loop counter is incremented, using the counter value after looping, is referred to as an off by one, or off by increment error, assuming the increment is in steps of one.

```
1. #include<stdio.h>
2. int main()
3. {
4. int total=0,markcounter=1,mark,average;
5. while ( markcounter<= 20)
6. {
7. printf("\n Enter Mark");
8. scanf("%d",&mark);
9. total=total + mark;
10. markcounter=markcounter + 1;
11. }
12. average=total/20;
13. printf("class average is %d",average);
14. return 0;
15. }
```

Note: In the above code:

Variable	Represents
total	sum of marks
markcounter	number of marks
mark	mark obtained
average	class average

Figure 2.2: C code to compute average.

2.4 Sentinel-controlled Repetition

Counter-controlled repetition can be the choice only if the number of repetitions is known in advance. What if in case the above program is to be used to compute the average of a class consisting of `n` students, `n` being an arbitrary number. A sentinel also called a signal or dummy flag value which indicates end of data entry. In this case, mark can be used as a sentinel value and the sentinel check on it could be (`mark ! =-1`). As long as value for mark variable entered by the user is not –1 (end of data entry) marks are accepted and the cumulative total is computed. One extension would be to count the number of marks entered by the user [would be required to compute the average] as the class strength is not known in advance and it is arbitrary.

The modified code looks like

```
printf("Enter value for mark \n");
scanf("%d",&mark);
while (mark !=-1)
{
total=total + mark;
markcounter =markcounter+1;
printf("Enter the next mark \n");
scanf ("%d",&mark);
}
```

Note: Programs that make use or sentinel-controlled repetition should at each stage of data entry clearly remind the end user about the sentinel value. And also there should be prompt statements to clearly indicate to the end user the input type and guide him/her in the input process.

The above code to reflect floating average would declare the variable average of float data type. However, still the result of `average = total / n;` will not reflect the fractional part since the result of an integer division [2 integers being divided] is always an integer and the fractional part is truncated. It is this division that is performed first and then assigned to the average variable. Thus, redeclaring average alone does not solve the purpose. To come out of this C, C++ supports type casting, an operation that creates temporary variables of the data type required.

The following statement `float average = (float) total/n` with average declared as float converts the integer variable total temporarily to type float. The result of a float divided by an integer quantity is a float and the value is assigned to the float variable average. Type casting or type conversion is only a temporary (for the operation) conversion and further references (excluding the type casting) the variable is accessed with the initial declaration in place or in effect.

Note: Choosing a valid data value as sentinel-value can cause serious logic errors. Most programs can be viewed as collection of three phases, namely initialization, processing and termination phases. Initializing variables at the point of declaration avoids the problem of uninitialized data. A good programming practice is to go through an exhaustive pseudocode or algorithmic phase after which development of code is straightforward.

2.5 Assignment Operators

These operators are made use of in assignment statements to assign values to variables. In addition to the naive (=) assignment operator, let us look at the other assignment operators available in C/C++. The `+=` is an abbreviated assignment operator. For example, the statement `a + = 3` expands as `a = a + 3` and performs the assignment of variable `a` incremented by three to the variable `a` itself. In fact, any statement of the form variable = variable operator expression can be written as variable operator = expression.

The advantage of using abbreviated operators is that certain compilers generate codes that run faster when abbreviated assignment operators are used.

2.5.1 Increment–Decrement Operators

C/C++ also provides the unary increment (`++`) and decrement (`--`) operators. These operators can be used to increment or decrement variables by 1. An increment/decrement operator placed before a variable is called preincrement/predecrement operator. An increment /decrement operator placed after a variable is called postincrement/postdecrement operator. The difference between the pre- and post-versions lies in when the assignment and increment/decrement operations are performed.

Unary increment (`++`) and decrement (`--`) operators are an abbreviated form for increment/decrement by 1 and then assignment. For example, `a++` expands as `a=a + 1` and `a--` expands as `a = a - 1`. Now, `++a` and `--a` also expand as `a = a + 1` and `a = a - 1`.

Then where lies the difference. The postincrement/postdecrement operator first executes the assignment and then performs the increment/decrement operations. The preversion first increments/decrements and then assigns the incremented/decremented variable to the LHS variable. In other words, Preincrement/Predecrement→ First, Increment/Decrement, and then Assign Postincrement/Postdecrement→First Assign, and then Increment/Decrement.

For better understanding let us consider the following numeric example shown in Figure 2.3.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  int a=5,b=6,c=0,d=0,e=0,f=0;
5.  c = a++;
6.  d = ++a;
7.  e = b--;
8.  f = --b;
9.  printf("Values after manipulations are
    %d %d %d %d %d %d",a,b,c,d,e,f);
10. return 0;
11. }
```

Figure 2.3: C code to illustrate increment and decrement operators.

- **Line 5:** `c = a++;` postincrement value of `a`, that is assigned the current value of `a` to `c` and then increment `a` by 1. Therefore, line 5 performs `c = 5` and `a = 6`.
- **Line 6:** `d = ++ a;` preincrement value of `a`, that is incremented the current value of `a` by 1 and then assigned this incremented value to `d`. Therefore, line 6 performs `a = 7` [6+1] and `d = 7`.
- **Line 7:** `e = b--;` performs `e = 6` and `b = 5`.
- **Line 8:** `f = --b;` performs `b = 4` and `f = 4`. Thus, the final output looks like: values after manipulations are 7 4 5 7 6 4.

Note: Unary operators should be placed next to their variables without any intervening spaces. It is just to improve code readability. Incrementing/decrementing a variable in a statement by itself, the pre and postversions have the same effect. It is only when a variable appears in the context of an expression involving other variables that pre and postincrement/decrement operators have different effects. In other words, `a++` and `++a` when used as a statement have the net effect of incrementing `a` by 1, with no differentiation between pre- and postincrement.

Attempting to apply increment or decrement on an expression rather than on a variable such as `++(a+5)` will cause syntax errors. The `++,--, + , -` (unary operators)after parentheses in the operator precedence table. Postincrement and postdecrement are before their precounterparts and associate left–right, whereas the preoperators associate right–left.

2.6 The for Repetition Structure

All features of counter-controlled repetition are provided with the `for` construct. The general structure of a `for` loop is for (initalizer; terminator; increment/decrement) The initializer takes care of initializing the loop counter variable to a valid value. The terminator checks the condition for further looping and to decide on the terminating or stopping criterion. The increment/decrement takes care of modifying the loop counter variable by the specified number of steps. The `while` loop of the earlier counter-controlled repetition example when replaced with a `for` loop looks as follows:

```
for (markcounter=1; markcounter <= 10;markcounter++)
```

The loop counter initialization and loop counter modification (`markcounter=markcounter+1`) can be excluded from the code when using a `for` loop. The `for` structure specifies each of the items needed for counter-controlled repetition. If the body of the `for` loop has more than one statement, then the statements should be enclosed within braces [{ }]. C has the requirement that loop counter/control variables also be part of the initial declaration statements. But C++ allows on the place declarations which means as a part of the initializer of a `for` header structure counter variables may get declared. But such variables remain accessible only from within the `for` loop. This restricted use of control variable name is called variables scope.

Scope refers to referential capability of a variable in a program which will be explained in Chapter 3 to follow. Sometimes the initialization and increment/decrement expressions can be comma-separated. Comma operator has the lowest precedence. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression in the list. Comma operators are often used in `for` loops. It aids in providing multiple initializers or increment and decrement expressions when more than once control variable controls the `for` loop. A list of expressions delimited by comma operator associates left to right and within an expression the result/value is the rightmost value.

The three expressions in the `for` structure are optional. If the terminator or the loop continuation test is omitted, C/C++ assumes that the loop continuation condition is true and creates an infinite loop. Initializer can be omitted assuming that the loop counter or control variable is initialized somewhere else in the program. The increment or decrement operation can as well be performed within the body of the `for` or if no increment is needed and thus gets excluded from the `for` header structure. Common programming errors that can occur with for loops are: commas used instead of semicolons in the `for` header cause syntax error.

A semicolon placed after the right parentheses of the `for` header causes the body of the `for` structure to be treated as an empty statement, and causes logical error. This is sometimes used to create a delay loop that does nothing but the counting operation. This delay can be used to provide a time gap when the output operations are otherwise too quick for the end user visibility.

The initializer, terminator and increment of a `for` header can also be expressions. It is valid to have a `for` structure such as: `for (j = x; j <= 4 * x * y; j += y / x)`, assuming `x=2, y=10`, the loop structure reduces as for `(j=2; i <= 80 ; j = j+5)`.

In the above `for` loop, the increment is done in steps of 5 each time. In general, the increment operation of a `for` structure can also be replaced with a decrement operation in which case the loop counts in the downward direction. If the loop continuation condition is

false to start with, then execution proceeds from the statements following the `for`.

A few other examples:

1. Vary the control variable from 5 to 55 in steps of 5 for `(i = 5; i <= 55; i += 5)`.
2. Vary the control variable from 100 to 1 in step of 1 for `(i = 100; i >= 1; i --)`.

A simple application of `for` loop to compute the sum of odd integers from 3 to 99 is shown in Figure 2.4.

```
1. #include<stdio.h>
2. int main()
3. {
4. int i=3, sum = 0;
5. for ( i = 3; i <= 99; i + = 2)
6. sum + = i;
7. printf("Sum of odd integers between 3 & 99 is %d",sum);
8. return 0;
9. }
```

Output:
Sum of odd integers between 3 & 99 is 2499

Figure 2.4: C code to compute sum of odd integers.

2.7 Good Programming Practices with `for` Loop

Avoid merging the `for` body with the `for` header even though it is possible syntactically. This reduces program clarity and makes programs difficult to understand. Limit the size of control structures to a single line of code (LoC). Avoid placing expressions whose values do not change inside loops, but then today's sophisticated compilers can remove such statements and place them outside the loop in the machine language code. This phase is referred to as Loop Optimization.

2.8 The `switch` (Multiple) Selection Structure

The `switch` selection structure is used when more than two alternative actions are possible, and one from the list of actions needs to be performed based on some integral values. The `switch` structure consists of a series of case labels and an optional default case. An example program to understand the syntax and semantics of the `switch`-case construct is shown in Figure 2.5. Develop a program to perform different arithmetic operations of `+,-,/,*` on two numbers input by the user based on a user choice.

Depending on the choice entered by the user the case labels [which can be integer or character] are compared and the case is chosen for execution. Cases actually correspond to the multiple courses of action possible with a programming logic. A program that has five different actions based on the value of a control variable will have five case labels in its

```
1. #include<stdio.h>
2. int main()
3. {
4. int number1, number2, choice, result;
5. printf("Enter number 1");
6. scanf("%d", &number1);
7. printf("\n Enter number 2");
8. scanf("%d", &number2);
9. printf("\n Enter your choice \n");
10. printf("1.Addition \n 2.Subtraction \n 3.Multiplication
    \n 4.Division");
11. scanf ("%d", &choice);
12. switch(choice)
13. {
14. case 1: result = number 1 + number 2;
15.    printf("The sum of %d & %d is %d",number1,number2,result);
16.    break;
17. case 2: result = number 1 - number 2;
18.    printf("The diff. of %d & %d is %d",number1,number2,result);
19.    break;
20. case 3: result = number 1 * number 2;
21.    printf("The prod. of%d&%d is %d",number1,number2,result);
22.    break;
23. case 4: result = number 1 / number 2;
24.    printf("The quot.of %d / %d is %d",number1,number2,result);
25.    break;
26. default: printf("Invalid Choice , Exiting \n");
27.    break;
28. }/*End of switch*/
29. }/*End of main*/
```

Figure 2.5: C code to illustrate use of `switch` construct.

`switch-case` construct. The default case is chosen for execution when the choice entered by the user matches none of the expected values. A block of statements to be treated as a single case need not be enclosed within braces ( ).

We resume back to our definition of `switch` selection structure; it is choosing one action from a list of multiple actions possible. So at any point of time it is only one action that can be performed. After having selected or performed one action there should be some way of quitting the remaining list of actions. The `break` statement precisely performs this. The `break` statement causes program control to proceed with the first statement after the `switch` structure. Missing out on `break` statement(s) will cause program control to proceed to the following cases until the compiler encounters a break in anyone of the following cases. The break statement literally breaks program control from the `switch` structure. This feature is exploited to perform the same action for several cases.

Possible programming errors with switch structures are: omitting the break statement can cause logical errors.Omitting the space between the keyword case and its label can cause a logic error by creating an unused case label. For example, instead of case 10, when coded as case 10, it will not perform the appropriate actions when the control variable has the value of 10. Providing identical case labels within the switch structure can cause syntax error.

A break statement may not be required with the default case if it is listed as the last case. However there is no syntactic compulsion regarding the ordering of the cases within a switch structure. Note that if the switch control variable is of type char, at run time to have the program read characters they are sent to the computer by the pressing of enter key on the keyboard. This places a new line character after the character is processed. This has to be handled separately to make the program work correctly by including a skipping logic with these special characters (\n,\t) [that is empty case statement]. Doing so one can prevent the default case getting intermixed with such special cases.

2.9 The do-while Repetition Structure

The do-while is similar to the while control structure. In while structure, the loop continuation condition is tested at the beginning of the loop or before the body of the loop is executed, the do-while constructs checks for the loop continuation condition after executing the loop body. Thus, the loop is executed at least once. A do-while on termination, execution continues with the statements after the while clause. Other aspects of the do-while and the while repetition structures are the same. do-while can be the choice in cases a sequence of actions will always be performed once and then based on some condition (sentinel value) the same sequence either gets repeated or execution proceeds with the next set of actions.

Syntax for do-while is as follows:

```
do
{
.
.
}
while (condition );
```

A do-while statement with no braces around (syntactically allowed) in a single statement body of the loop appears as follows:

```
do
statement
while (condition);
```

This can cause the reader to misinterpret the statement while (condition), as an empty while construct which clearly is not the case. Hence to avoid this, it is a good practice to always have braces for do-while construct of the form mentioned earlier. Thus, all the set of actions of a do-while construct gets executed at least once.

2.10 break and continue Statements

These statements alter the flow of program control. The `break` statement when made use of within repetitive or control structures such as `for/do-while/while/switch` causes immediate exit from that structure. Program execution continues with the first statement after that structure. The `continue` statement used with repetitive control structures skips the remaining statement(s) after the `continue` statement and proceeds with the next iteration of the loop. In `while` and `do-while` structures the loop continuation condition is evaluated immediately after `continue` statement, while with `for` structure the increment is first executed and then the loop continuation condition is executed. The next exercise will explore on the application of `break` and `continue` statements within a program and its effect on program execution. `break` and `continue` statements do alter the flow of program control to some extent, as will be clear from the following example shown in Figures 2.6 and 2.7.

```
 1.  #include<stdio.h>
 2.  int main()
 3.  {
 4.  int x;
 5.  for(x = 1 ; x <= 20; x ++)
 6.  {
 7.  if(x == 5)
 8.  break;
 9.  printf("%d",x);
10.  }
11.  printf("Exited loop @ %d",x);
12.  return 0;
13.  }
```

Output of the above code:

1 2 3 4 Exited Loop at 5.

Figure 2.6: C code to explain the use of **break**.

2.11 Logical Operators

The C/C++ logical operator can be exploited to form complex conditions that can be used in control structures. The logical operators of `&&(AND)`, `||(OR)`, `!(NOT)` are used to frame complex conditions. The logical `AND (&&)` operator can be used to test if all the conditions (more than 1) are TRUE or not [if `(a==5 && b==8)` C/C++ code$\cdots$].

Logical `AND` first evaluates the leftmost condition. Only if this evaluates to true, does it proceed evaluating the remaining conditions? This is based on the principle that all conditions should evaluate to true for the output of logical `AND` to be true. This optimized logical `AND` is also commonly referred to as short-circuiting. For a better understanding the truth table of a two input `AND` operation is shown in Table 2.2.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  int x;
5.  for(x = 1 ; x <= 10; x ++)
6.  {
7.  if(x == 5)
8.  continue;
9.  printf("%d",x);
10. }
11. printf("Control here") ;
12. return 0;
13. }
```

Output:

1 2 3 4 6 7 8 9 10 Control here.

Figure 2.7: C code to explain the use of `continue`.

Table 2.2: Logical `AND` truth table

Expression 1/Condition 1	*Expression 2/Condition 2*	*Result*
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE

Logical `OR` (||—double pipe symbol) is used to check if at least one among the several conditions is TRUE or not. The truth table is shown in Table 2.3.

Table 2.3: Logical `OR` truth table

Expression 1/Condition 1	*Expression 2/Condition 2*	*Result*
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

Both logical `AND` and `OR` are binary operators. Thus it is only any two conditions that are checked for trueness at any time. Multiple condition checks using `&&`,| textbar proceed left-right combining the first-two conditions and then the result of this operation is logically `AND`ed or `OR`ed with further conditions. The remaining logical operator (`!` - `NOT`) is used to test the reverse meaning of a condition. It is a unary operator and is called logical negation operator. It is placed before a condition when the programmer wants to choose a path on the original condition being false. For example:

```
if ( !  ( x == y ) )
printf("Example \n");
```

The parentheses are important because the ! operator has higher precedence than the quality operator. Otherwise, the expression will be parsed by the compiler as (! X == Y) which is not what we want. Common logic errors that can occur with (=) and (==) operators are as follows:

Often programmers accidentally swap the equality (==) and the assignment (=) operator. Unfortunately, this does not cause syntax error, but can lead to serious logical errors. An example to understand this concept:

```
if (a == 0)
printf("Success \n");
else
printf("Failure \n");
```

The output on the screen is failure. How? Line 2, an intended comparison as a result of the missing = is performed as an assignment operation. It proceeds as an assignment operation. It proceeds as `if` (0). The result of the condition, 0 [false in C/C++] causes the else part to be executed even though the value of a is equal to 0. Any non-zero value returned by a condition is treated as true and zero value is treated as false. And more worse the equality operator (used for comparison) is more of a `read` operation. When this is accidentally swapped with the assignment operator (=) it amounts to the LHS variable having its value changed, an operation that would not have been intended by the programmer.

One diplomatic solution for overcoming this problem would be to code comparison/equality statements as constant value == variable name, e.g. `10 == a` in an `if` condition would solve our problem because an accidental miss of an = in the above format would be treated as `10 = a;` assignment operators expected value to be a variable, which is clearly violated here and hence the error gets caught at the compilation stage itself. Well, this solution is valid only as long as variable is compared with a constant value. But once when two variables need to be compared or checked for equality or compared the above solution solves no purpose. But then more often it is with constant values that variables are checked for equality. The case when two variables are compared, well, there is no way out for tracing this logical error. Programmer will have to live with it but then this arises only due to programmer's slack coding practices.

Now, the other scenario when == is used in the place of =, such as `X == 5;` again this does not lead to syntax errors. It proceeds as a comparison operation and the value returned is true. Irrespective of true or false is returned the value 5 is lost once for all! Unfortunately, there is no way out for this problem as we had for the earlier one.

Review Questions

1. Differentiate an algorithm from a source code.
2. Interpret the term control structure and name the three control structures available in C++.
3. Name a few object-oriented languages other than C++ available in the market today.
4. Develop a C++ program to compute the maximum and minimum of three integers accepted from the user.

5. Develop a C++ program that accepts marks scored by a student in six subjects and display whether he/she has passed or not. Assume a passing minimum. Also display the average in case the student has passed in all subjects.
6. Develop a C++ program that checks if a number input by the user is an (i) armstrong, (ii) adams number or not. An armstrong number is one in which the sum of cubes of the individual digits of the number equates to the original number. An adams number is one in which the reverse of the square of the number is the same as that of square of the reverse of the number. Example for armstrong number is 407, where $4^3 + 0^3 + 7^3 = 407$, and adams number is 12, where square of 12 is 144 and reverse is 441, which is the same as square of 21 (reverse of 12).
7. Develop a C++ program that generates the list of all armstrong and adams numbers between a user-specified range. Accept user-specified lower and upper limit.
8. Develop a C++ program to check if an user input integer is a palindrome or not. A palindrome is an integer that reads the same forward and reverse. Example is 1441.
9. Differentiate the break and continue statements with suitable examples (other than the one used in the text).
10. Develop a C++ program that generates the multiples of 2 within a user-specified range. Display the square, cube and square of numbers within a user-specified range in a tabular column format.
11. Develop a C++ program to compute the factorial of a number using both increment and decrement loops.
12. Develop a C++ program to perform decimal to binary, octal and hexadecimal format conversion and vice versa using `switch` case construct.
13. Develop a C++ program that computes the values of e^x.
14. Explain the effect of a misused assignment operator as a part of a conditional statement within an `if` statement.
15. Visit the directory in Unix/Linux where header files used in C and C++ are stored and appreciate the contents of them.

Chapter 3

Functions

In the first-two chapters, we have come across programs that were not much complex both in terms of problem definitions and program/code development. However, realistic requirements would engage the programmer in developing large programs, larger in terms of the lines of code. Software engineering observations have been that the best way to develop and maintain large-sized programs is to construct it from smaller components or pieces each of which is easier to solve than the original problem. In algorithm terminologies this is called the divide and conquer strategy. Modules in C/C++ are called functions.

C/C++ programs are developed by combining both new programmer defined functions and predefined functions from the standard library. The standard library is quite exhaustive including commonly used functions for mathematical computations, standard input and output, string manipulations and so on. Such functions reduce the effort required on the programmer's side and thereby reducing development time and chances of errors. Good programming practice is to make use of available predefined functions to the extent possible and avoid reinventing the wheel. Another hard fact to be accepted is that programmers should spare their efforts in modifying or rewriting the standard library functions to make them efficient.

A function is called or invoked by a function call. The function call specifies the function name and provides (arguments) which the called function needs to carry out its task. The real world analogy is the hierarchical form of management. A boss (caller function) asks a worker (called function) to perform a task and return (report back) the results to the boss when the work is done. The boss/caller function is totally unaware of how the worker or the called function had performed its task. The worker function can again call other helper or worker functions.

However, for the main caller or boss function such details are hidden and looks as if the work was done by the immediately called worker function. Well in the lighter vein of it, this information hiding may lead to one person assuming credit for work of the other. However, this is accepted universally as a rule of nature. In fact, the main advantage with this approach is that the responsibility lies with the caller function and not with the called function in case of mishaps [Very rarely does this mishap occur but its occurrence does have a chance]. Whether or not this information hiding offers advantages in real life, it does offer key benefits in software engineering that will be explained later in Data Abstraction, an OOP feature.

3.1 Math Library Functions

The math library functions allow the programmer to perform certain mathematical computations. In this section, we shall look at a few such functions. Functions are called by writing the

name of the function followed by a left parenthesis, then the argument list is comma-separated and finally followed by a right parenthesis. For example, to compute the square root of a number using the predefined `sqrt()` function, the programmer would call it in this manner of `sqrt(625.0)` which would return the square root of the number. Here, 625 is the argument to the function `sqrt()`; the square root function accepts and returns a double argument.

All functions of the math library header file return data type double. Math library functions are made use of in the code by including the header file <math.h> that contains the declarations for various mathematical functions such as `pow()`, `sin()`, `cos()`, etc. Function arguments can be constants, variables or expressions. For example, the same sqrt function can be called `sqrt(a+d*f);` assuming `a, d, f` have been suitably declared and initialized as a=24, d=5 and f=5. A few other available math functions are shown in Table 3.1. The above list is not exhaustive. Readers should refer to the <math.h> header file for an idea about the entire range of such math functions available with the language.

Table 3.1: Math library functions

Function	*Description*	*Example*
`ceil(x)`	Round x to the smallest integer >= x.	`ceil (5.2) = 6`
`cos(x)`	Trigonometric cosine of x in radians.	`cos ( 0 ) = 1`
`pow(x,y)`	x raised to the power y	`pow (3,5) =243`

Functions allow programmers to modularize the program. Variables declared within a function definition are called local variables, referring to the fact that they are known only within the function in which they are declared. Such variables are referenceable (accessible) only from within the function. Most functions have a parameter list (called arguments), they provide the means for communicating information between functions. A function's parameters are also local variables.

Key advantages of functions/program modularization are as follows:

1. Divide and conquer makes program size more manageable.
2. Modularization promotes software reusability with programs being created from standard functions that perform the required tasks rather than being created by the user.
3. Avoid code repetition. Codes that are repetitive in nature can be packaged as function(s) and executed from several locations by invoking/calling the function.

A good coding practice is to have functions performing limited or well-defined task and the name of the function, also called function identifier should be meaningful and express the task that it is performing. This promotes readability and reusability. For example, a function that is intended to perform the sum of matrix elements passed to it should be named as `matrix_sum()` [variable names/ function names or identifiers can have underscores in them]. Reusability that is promoted by good modularization is one of the key features of object-oriented programming.

Readers in fact have been already exposed to the concept of defining functions when they defined the function `main()`. Tasks which the user wanted to perform (either single line instructions or statements or subtasks or functions were defined in function `main`. `Main()` is one compulsory function that cannot be given any other name. `main()` will have statements [assignment or computational, control and repetitive structures, function calls, user-defined or predefined functions] specified or defined in them. In fact, in the first chapter as a part

of header file inclusion concepts, we had a detailed discussion on function declarations and function definitions. Those concepts are applicable here as well. The following example shown in Figure 3.1 will define a function square to compute the square of an integer argument passed to it.

```
1.  #include<stdio.h>
2.  int square ( int );
3.  int main()
4.  {
5.  int number;
6.  printf("Enter the no for which you want the find square \n");
7.  scanf ("%d", &number);
8.  printf("Square of %d is %d", number, square (number));
9.  return 0;
10. }
11. int square ( int a )
12. {
13. return ( a * a );
14. }
```

Figure 3.1: C code to compute square of a number using functions.

3.2 Code Interpretation

Line 2 is the declaration of the function square or prototype for function square. As has been already explained `int square (int)` identifies that the function square would accept one integer argument and at the end of processing return an integer as the result of the computation. Function declaration's main purpose is to handle mismatched function calls at compilation stage itself rather than leaving it till run-time. Note that in function declaration variable name or identification is unnecessary because it is only the number and data type of each argument that is important than the variable name.

They act as what are called place holders and it is the place (number, data type of arguments) that is important. However, function declaration is not required if the function definition appears before the first use of it in the program, in which case the definition also acts as a prototype. The void return data type indicates that the function returns nothing (does not return any value).

3.3 Possible Coding Errors with Functions

Possible programming errors while defining functions are as follows:

- Returning a value from a function whose return type has been declared as void
- Forgetting to return a value from a function that is supposed to return value
- Missing out on return type of a function while defining it.

The parameter list of a function definition is comma-separated containing the declarations of the parameters the function receives when it is called.

A function that does not accept any arguments can clearly indicate that the argument list is empty with the keyword void or by leaving it simply empty such as `int square (void);` or `int square();` Both function declarations mean the same. All the arguments in the parameter list should have an explicit data type mentioned. Errors that are possible with function declarations are:

Not mentioning explicit data types for all the arguments in the parameter list. For example, a function header of the form float `somefunc(float x, y)` will cause a syntax error. The float data type is applicable to the argument x only and not to y which is the case with normal variable declarations. The normal variable declaration syntax is not applicable for declaring the arguments of a function header and requires individual data types. The correct form of the above function header will be as follows:

```
float somefunc ( float x, float y)
```

Placing a semicolon at the end of the function header while defining a function will cause syntax errors. Declaring/defining a function parameter (argument) as a local variable within the function definition leads to syntax errors).Avoid having the same names for the arguments passed to a function and the variables within the definition; this may lead to ambiguity. Missing out on the braces (parentheses) during function `call` or `invoke` causes syntax errors.

The declarations and statements within the { } of a function forms the function definition. It is also referred to as the body of the function or block. A block is generally a compound statement that includes declarations and other statements. Blocks can be nested and contain variable declarations in them. But then a function cannot be defined inside another function, it leads to syntax errors. Another possible syntax error occurs if the function prototype, function header and function calls do not agree in the number, type and order of arguments and parameters, and in return value type.

Software engineering observations have been that a function requiring many arguments is probably task-extensive and is an ideal candidate for further subdivision into smaller functions. Programs coded as collection of small functions are easier to maintain, modify and debug. We have already seen that smaller functions promote software reusability. The `return` keyword is used to return value from a function. An empty `return` statement (void return type) simply passes the program control to the caller function to resume execution from the next instruction or statement onwards (in sequence). A `return` statement followed by an expression or value passes the value to the caller function and stores it in the corresponding LHS variable. Let us now develop a simple program to compute the maximum of three numbers using functions, code for which is shown in Figure 3.2.

In function `main` three integers are declared and accepted from the user. The three integers are passed to the function `maximum` which computes the `maximum` (logic is pretty simple) and returns the computed `max (local_x)` to the `main` function which is captured in the variable `max` and then finally displayed along with the three integers input by the user.

One important feature of function prototypes is the coercion of arguments forcing the arguments to the appropriate data type required. Most often the arguments passed by users of predefined or standard library functions will not be in exact match with their respective function prototypes available in header files. One cannot enforce the restrictions that the arguments passed by end-users of predefined functions match exactly. Hence the requirement

```
1.  #include<stdio.h>
2.  int maximum ( int, int , int );
3.  int main ()
4.  {
5.  int num1, num2, num3, max;
6.  printf("Enter the three numbers \n");
7.  scanf ( "%d %d %d",&num1,&num2,&num3);
8.  max = maximum ( num1, num2, num3);
9.  printf ("The maximum of %d , %d & %d is %d",num1,num2,num3,max);
10. return 0;
11. }
12. int maximum ( int a , int b , int c)
13. {
14. int loc max= a;
15. if ( b > loc max)
16. loc max=b;
17. if (c > loc max)
18. loc max=c;
19. return loc max;
20. }
```

Output:

Enter the three numbers

20 40 60

The maximum of 20,30 & 40 is 40.

Figure 3.2: C code to compute maximum using functions.

to convert the arguments of end users to the proper type before the respective function is called.

Conversion from lower byte consumption to higher byte consumption is called promotion and vice versa is called demotion. Such promotions/demotions should follow the rules of the language. For example, the `int` argument passed to the `sqrt()` function gets coerced to double type without changing the value. However, the vice versa conversion would truncate the fractional part and will result in changed values. More exhaustive programming examples involving functions shall be explored in the further sections.

3.4 Storage Classes

In the previous sections, we have seen that a variable gets identified by a name and a data type. In fact, it is based on this identification that variable names are also referred to as identifiers. In addition to these properties of name and data type, each identifier in a program has other attributes/features of storage class, scope and linkage. In this section, we shall elaborate on each of these concepts. The rest of this section to follow reviews the variable

declaration concepts. The storage classes available are: `auto` (automatic), `register`, `extern` (external) and `static`. C++ in addition to the above specifiers supports a mutable storage class specifier which will also be explained here, purely from a better understanding point of view.

An identifier's storage class determines the period or duration during which the identifier exists in memory. Certain identifiers exist in memory for a short period. Certain identifiers are repeatedly created and destroyed [allocation followed by deallocation] while certain identifiers exist for the entire duration of program execution. An identifier's scope identifies the places in a program where an identifier can be referenced. Certain identifiers can be referenced throughout the program while others can be referenced only from within a block / limited portions of the program.

An identifier's linkage determines the reference ability of variables or identifiers across source files, a concept that shall be explored with multiple source file development. The storage class specifiers are grouped in two classes, namely the automatic storage class and the static storage class. The keywords `auto` and `register` are used to declare variables of the automatic storage class type. Such variables are created when the block in which they are declared is entered, exist as long as the block is active and they are destroyed when the block is exited. Automatic storage class specifier is applicable only for variables.

A function's local variables and parameters are of the automatic storage class by default, explicitly declares variables of the automatic class. The syntax is as follows: `auto int x,y;` explicitly declares variables `x` and `y` to be of integer data type and automatic storage class. Such variables exist only in the function body where their declaration or definition appears.

A function's local variables that are of automatic storage class type by default are in short referred to as Automatic Variables. The nature of automatic variables conserves memory space. It is based on the principle of least privilege, not to have unreferenced or unnecessary variables in memory. Data in the machine language version of a program are loaded in registers for calculations and other processing. The storage class specifier register can be placed before an automatic variable declaration to denote that the variable is maintained in high speed registers than in memory.

Access from registers is fast in comparison to access from main memory. Frequently accessed variables those that require frequent updation (read and write) can be maintained in hardware registers thereby eliminating the overhead of repeatedly loading variables from memory to register and back to memory after updation. Multiple storage class specifiers applied for identifiers cause syntax errors.

Variables such as loop counter, cumulative total are better off to be stored in registers rather than in main memory to avoid the above-mentioned overhead. In case of insufficient registers, the compiler might ignore the register declarations and store it in memory itself. The `register` keyword is applicable only with local variables and function parameters. The syntax for a register type variable as follows: `register int count = 1.`

In fact, with the modern optimizing compilers frequently referenced variables are automatically placed in registers even without the register declaration. The next storage class type of static is declared using the key words `extern` and `static`. Static storage class variables exist from the point of program execution. Static variables have storage allocated and initialized when the program begins execution. Even though such variables and functions exist from the point of execution scope still has weightage regarding the `reference` ability of the variables or functions.

Two types of external identifiers with static storage class are global variables and function names, and local variables with static storage class specifier. Global variables and function names default to the `extern` storage class specifier. Global variables are created by placing variable declarations outside the function definition. Such global variables retain their values throughout the program execution. Global variables or functions can be referenced by functions that follow their declaration and definition in the source file. Global variables when compared with local variables allow unintended side effects to occur when a function that does not need access accidentally modifies it. They are similar to goto statements and should be avoided to the extent possible and used only in exceptional cases.

Variables that will be accessed only within functions should be declared as local rather than global variables. Local variables (within function) declared with static storage class specifier are known only in the function where they are defined, but unlike automatic variables static variables retain their values when the function is exited so that the next time the same function is called again static variables contain values they had when the function last exited. All static storage class variables are initialized to zero, whereas auto storage class identifiers are initialized with garbage or junk values.

3.5 Scope Rules

The portion of a program where an identifier has meaning is known as its scope. When we declare a local variable in a block it can be referenced only from within the block or blocks nested within it. This is referred to as block scope. The other scopes available for identifiers are function scope, file scope, function prototype scope and block scope. An identifier declared outside any function has file scope. Such an identifier is known in all functions from the point of declaration until the end of the file. Goto labels (identifiers with :) are the only variables that have function scope. Labels cannot be referenced outside the function body. Labels are also made use of in `switch` structures as case labels. Identifiers declared inside a block have block scope.

Block scope begins at the right brace (}) of the block. Local variables declared at the beginning of function definition and function parameters (local variables of functions) have block scope. When blocks are nested and an identifier in an outer block has the same name as that of an identifier in the inner block, the outer block identifier is hidden till the inner block terminates. Within the inner block it is the inner block's local identifier that is applied and the identifier in the enclosing outer block with the same name is ignored (hidden) within the inner block. As we have already stated, the static local variable that exists for the entire duration of program execution still has only block scope.

Storage class does not influence the scope of an identifier. Identifiers used in the parameter list of a function prototype have function prototype. For such identifiers names are not required, as they serve only as place-holders, the compiler ignores such names and only the individual types are required. Identifiers used in function prototype can be reused in the program without any ambiguity. Though it is syntactically allowed, it is a better practice to avoid identical identifier names within nested blocks. Such duplicate identifiers though allowed if not referenced properly can cause serious logic errors. The following example shown in Figure 3.3 helps in better understanding of storage and scope rule concepts.

```
1.  #include<stdio.h>
2.  void function1(void);
3.  void function2(void);
4.  void function3(void);
5.  int glob var a=1;
6.  int main ()
7.  {int a = 5;
8.  printf ("Local a in outer scope is %d",a);
9.  {
10. int a=8;
11. printf ("Local a in inner scope is %d",a);
12. }
13. printf ("Local a in outer scope is %d",a);
14. function1();
15. function2();
16. function3();
17. function1();
18. function2();
19. function3();
20. printf("Local a in main is %d",a);}
21. void function1(void)
22. {int a =30;
23. printf ("Local a on entering function1 is %d",a);
24. ++a;
25. printf ("Local a on leaving function1 is %d",a);}
26. void function2(void)
27. {static int a = 40;
28. printf("Local static a on entering function2 is %d",a);
29. ++a;
30. printf("Local static a on leaving function2 is %d",a);}
31. void function3 (void)
32. {printf("Global a on entering function3 is %d", a);
33. a = a * 20;
34. printf("Global a on leaving function 3 is %d", a);}
```

Figure 3.3: C code to illustrate storage classes and scope.

Before function `main`, a global variable `a` is declared and initialized (`a = 1`). Blocks where the identifier `a` is redeclared, the global value of `a` is hidden hence the global value is alive and active. Within function `main`, an auto local variable `a` is declared (initialized with 5). Within an inner block, of `main`, `a` is again redeclared and initialized with 8. Within each inner block, the enclosing outer block, value of `a` is hidden. The function set [`function1, function2, function3`] are called twice [basically to differentiate the initialization logic associated with global, auto local and static local variables].

Within `function1` a local automatic variable is declared and initialized with 20. This variable is incremented. This operation is to distinguish the starting point values with each storage class type. Within `function2`, a local static copy of `a` is created and initialized with 40. It is again incremented for reasons explained earlier. In the last `function3`, there is no local copy of `a` that is created and the references to `a` actually refer to the global value of `a` defined outside `main()`. Note that if a global variable is not defined and `function3` [that does

not have a local a defined] refers to a, then a syntax error of the form variable undeclared is thrown. Output of the code shown in Figure 3.3 is shown in Table 3.2.

Table 3.2: Output of the code shown in Figure 3.3

Output:
Local a in outer scope 5
Local a in inner scope 8
Local a in outer scope 5
Local a on entering function1 20
Local a on leaving function1 21
Static a on entering function2 40
Static a on leaving function2 41
Global a on entering function3 1
Global a on leaving function3 20
Local a on entering function1 20
Local a on leaving function1 21
Static a on entering function2 41
Static a on leaving function2 42
Global a on entering function3 20
Global a on leaving function3 400
Local a in main 5.

Note that the local automatic variable a in `function1` is created and destroyed each time the function is entered or exited. That is the reason for the incremented value (done in first call of `function1`] not reflected in the second call of `function1`. But the static local variable a in `function2 ()` retains its value when `function2 ()` is exited and reentered the second time. That is the reason for the modified value of a getting reflected in the second call of `function2 ()`. References to a within `function3` refers to the global value of `a =1` which is modified by the two calls to `function3`. The last statement in `main` refers to the local a defined in the scope of `main`. References to a outside `main` and in functions other than `function1` and `function2` refers to the most recently updated global a.

3.6 Recursion

All along the programs were developed by calling other functions to perform the task. However, in certain occasions it may be required to have function calling themselves within their definition. This is referred to as Recursion. Two types of recursion are: direct and indirect recursion. A function calling itself as an explicit statement is direct recursion. A function calling another function, which in turn calls the earlier or boss function, is indirect recursion. Examples of the above types of recursion are shown in Table 3.3.

Recursion can also be viewed as a divide and conquer strategy. A recursive function has solution to the problem only for the simple or what are called base cases [most often it is the smallest problem size]. The recursive function keeps dividing the problem [to smaller subproblems] until the base case is reached for which solution is known. The recursion step normally involves a return statement since its results will be combined with the portion of the problem, which lead to this call. All the solutions of the n recursive calls are combined to

Table 3.3: Types of recursion

Direct recursion	*Indirect recursion*	
`void function1 ()`	`void function2()`	`void function3`
`{`	`{`	`{`
⋮	⋮	⋮
`funtion1();`	`function3();`	`function2();`
⋮	⋮	⋮
`}`	`}`	`}`

form the overall solution to the problem and pass it back to the original caller function, which would normally be `main()`.

To make recursion solvable, the newly created problem [divided subproblems] must resemble the original problem in nature but should be slightly simpler or smaller than the original problem in terms of problem size. Since the new problem resembles the original problem, the function launches [calls] a fresh copy of itself to work on the smaller problem. This is the recursive step or recursive call. The recursive call or recursion step executes while the original call to the function [first call] is still open and not yet finished executing. The recursion step can result in many more such recursive calls as the function keeps dividing each new subproblem with which the function is called into smaller pieces.

As we have already stated that this division continues until the base case is reached which is also referred to as recursion bottoming out. At that bottoming out point, the function recognizes the base case, returns a result to the previous copy of the function and a sequence of such returns happens all the way up the line until the original call of the function eventually returns the final result to the main function. As an example let us compute the factorial of a number using a recursive program. Towards the latter half a non-recursive solution for the same problem is implemented.

Problem: Develop a function called factorial to compute the factorial of an integer passed to it. The mathematical formula `n!= n(n-1)!` is made use of to implement a recursive solution to the above problem. The base case of the recursion is `n=1` for which the factorial is `1`. Let us now see the program to perform `n!`, code for which is shown in Figure 3.4. The list of recursive calls for `5!` is shown in Figure 3.5 for better understanding. The forward arrows indicate the recursive calls till the base case of `n=1` is reached. When the base case is reached, the last recursive call returns the result of the base case to the previous call and such returns proceed up till the original call in main is reached. The reverse arrows indicate this backward return.

A non-recursive solution to the above problem could be achieved with the following `for` structure:

```
(for int ctr=number; ctr >=1; ctr --) factorial = factorial * ctr;
```

The above `for` structure should be inserted in a valid C code. The loop structure is based on `5!= 5 * 4 * 3 * 2 * 1 = 120`.

```
1. #include<stdio.h>
2. int factorial(int);
3. int main()
4. int number,result;
5. printf("Enter the number for which the factorial
   is to be computed\n");
6. scanf("%d",&number);
7. result= factorial(number);
8. printf("Factorial of %d is %d",number,result);
9. return 0;
10. }
11. int factorial (int no)
12. if (no<=1)
13. return 1;
14. else
15. return no * factorial ( n-1 );
16. }
```

Figure 3.4: C code using recursion to compute n!.

5!→	5*4!→	4*3!→	3*2!→	2*1!→	1!
120	←5*24	←4*6	←3*2	←2*1	←1

Figure 3.5: Trace of `n!` using recursion for `n=5`.

3.6.1 Fibonacci Series using Recursion

The fibonacci series 0,1,1,2,3,5,8,13,21 begins with a 0 and 1 and the subsequent numbers are the sum of the previous two-fibonacci numbers. Fibonacci series can be recursively defined as

`fib( n ) = fib ( n-1 ) + fib ( n-2 )`

Base cases:

`fib ( 0 ) = 0, and fib ( 1 ) = 1;`

Now using the above recursive and base cases let us develop the code to generate the *i*th fibonacci number ; `i` being an user input. Code for recursive version of fibonacci series generation is shown in Figure 3.6.

A pictorial representation for the `call fib (5)`, that is to generate the 5th fibonacci number is shown in Figure 3.7.

The recursive calls may be processed left to right or right to left. The programmer should make no assumptions regarding the processing (left–right) or (right–left). In most cases the results would be the same, however, in certain cases the results will not be the same. The first call to a recursive function cannot be counted as a recursive call. It is the subsequent calls (including the base case call) that make up the total set of recursive calls.

```
 1. # include<stdio.h>
 2. int fib(int);
 3. int main ()
 4. {
 5. int number,result;
 6. printf("Enter the number \n");
 7. scanf("%d",&number);
 8. result = fib (number);
 9. printf("The %dth fibonacci number is %d",number,result);
10. return 0;
11. }
12. int fib (int no)
13. {
14. if ( no == 0 ||no ==1)
15. return no;
16. else
17. return fib(no -1)+fib(no-2);
18. }
```

Figure 3.6: C code to generate fibonacci series using recursion.

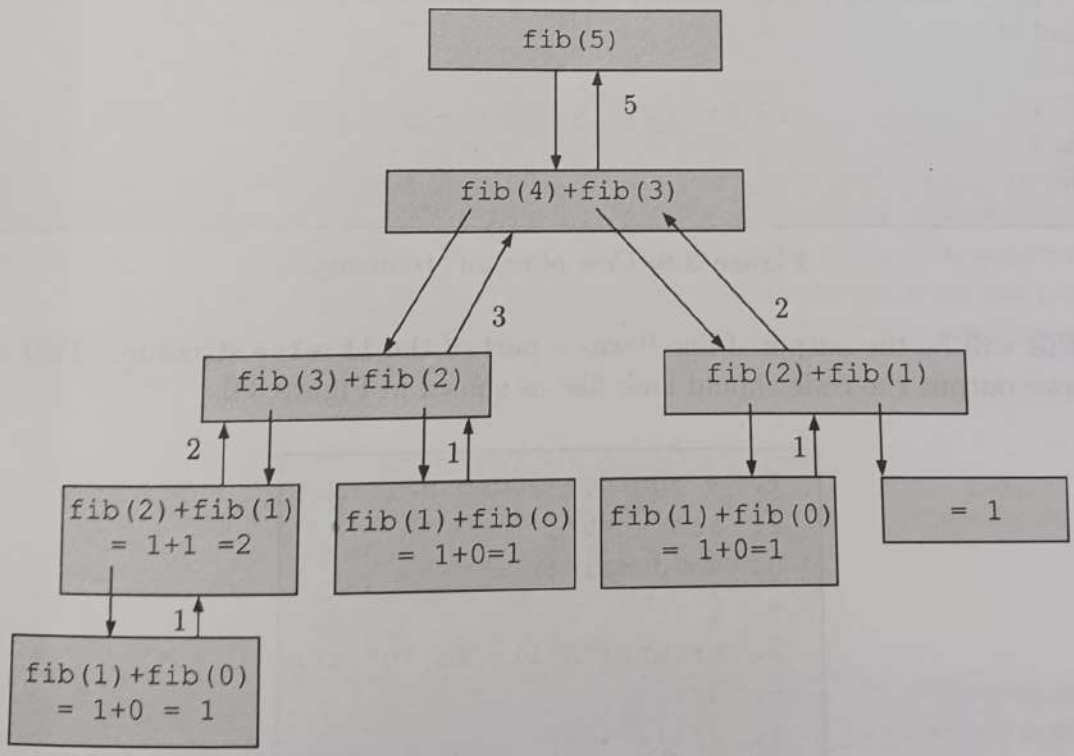

Figure 3.7: Trace of fibonacci code for n=5.

Developing programs that depend on the order of evaluation of operators (subexpressions) can lead to errors since it is a compiler specific issue and varies across compilers. Hence systems built using such programs will not remain stable. In the above pictorial representation, the forward arrows denote the recursive call to function `fib` and the reverse arrows indicate the

value returned after each recursive call terminates. Let us now explore two examples that shall strength our understanding of the concepts of recursion. Readers are advised to walk through the code, trace through each recursive call separately to predict the output.

3.6.2 Another Example for Recursion

What is the output of the following code shown in Figure 3.8?

```
1. # include<stdio.h>
2. int a =0;
3. void main()
4. {
5. printf("Hello %d ₩n",a);
6. a++;
7. if (a < 3)
8. main();
9. printf("World %d \n",a);
10. }
```

Most programmers at the first instance come out with the following output of
Hello 0
Hello 1
Hello 2
World 3.

Figure 3.8: Case of main() recursing.

Well, this will be the output if line 9 was a part of the `if-else` structure. That is to say for the above output the code should look like as shown in Figure 3.9.

```
1. # include<stdio.h>
2. int a =0;
3. void main()
4. {
5. printf("Hello %d \n",a);
6. a++;
7. if (a < 3)
8. main();
9. else
10. printf("World %d \n",a);
11. }
```

Figure 3.9: Case of main() recursing.

But then this was not our original code. The world `printf()` is not a part of the `if-else` construct, but it is a part of the definition of `main()`. So, the number of times the function `main` gets called, this `printf` will also get executed that many number of times. By default `main` is called once by the compiler and twice as a result of the recursive calls [if - part]. Thus, in all `main` gets called three times. Thus, the `printf("World %d",a);` will also get executed three times. In fact, the actual output is

```
Hello 0
Hello 1
Hello 2
World 3
World 3
World 3
```

Having already established that the `world` display occurs three times, let us now look at the program control flow in the above recursion exercise, for which to understand we need to trace through each recursive call separately. The recursion sequence and return of control to the previous function can be understood from the diagrammatic representation given in Figure 3.10.

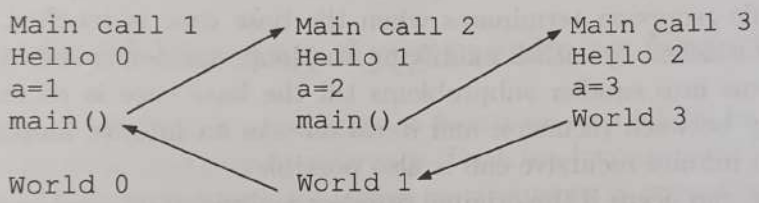

Figure 3.10: Trace of code shown in Figure 3.8.

The above trace should clarify the transfer of program control flow. Truly speaking the `world` string that appears on the screen should have the values of a associated with the respective recursive call. But then since a is a global variable, there will not be separate copies of a created in each call, but there is one global copy over which the updations are performed. Thus, the final stable state of `a (3)` is reflected in each of the display of the `world` string three times. Thus, the output of the above code is as follows:

```
Hello 0
Hello 1
Hello 2
World 3
World 3
World 3
```

This is a classical example to understand the nuances of recursion. The following interesting code shown in Figure 3.11 will be left as an exercise to the reader. Kindly do not test the code on a compiler. The whole charm of it will be lost.

3.6.3 Recursion vs. Iteration

Programmers should be aware of the fact that tasks that are solved by recursion can also be solved in a non-recursive (iterative) manner. Hence there is a need to differentiate recursion

```
1. # include<stdio.h>
2. int a =0;
3. void main()
4. {
5. printf("Hello %d \n",a);
6. a++;
7. if (a < 3)
8. printf("World %d",main());
9. return a;
10. }
```

Figure 3.11: Recursion exercise.

from iteration. Iteration and recursion both are based on control structures. Iteration uses a repetitive control structure, while recursion uses a selection control structure. The repetition in recursion is achieved through repetitive function calls. There is a termination test involved in both.

Iterative structures stop when the loop continuation condition specified in the iterative structure fails, while recursion terminates when the base case is reached. The terminating criterion in each is reached by either modifying the loop counter variable or by subdividing the original problems into smaller subproblems till the base case is reached. In fact, there is a lot of similarity between recursion and iteration. As an infinite looping is possible with iteration so does an infinite recursive call is also possible.

Infinite recursion can occur if the original problem is divided into subproblem(s) that never reach the base case or infinite recursion can as well occur as a result of no condition governing the recursive call. The overhead in terms of the processor time and memory space associated with recursion is more or large when compared with their counterparts, namely iteration. Each recursive call creates a fresh copy of the function (only the variables) that can consume reasonable amount of memory. And also the control transfer or return in recursion could be more.

All such bottlenecks are not associated with iteration. Thus it is more often iterative solutions that is favoured in comparison with recursive solutions. However, to ease out code understanding and debugging recursion may be opted. And also when an apparent iterative solution is not visible and recursion is a natural reflection in the problem definition (subdivisional nature), recursion may be the choice. But never choose recursion when system performance is a key issue.

Note: This being an OO text, in the initial few chapters we have explored structured programming, elaborating on the advanced structured programming concepts available only in C++. The following discussions and in fact, the earlier discussions on function prototypes are with C++ in mind.

3.7 Empty Parameter List Functions

As we have already stated functions that do not accept any arguments is indicated by an empty braces () or explicitly mentioning the keyword `void` within the parentheses [`(void)`].

Both syntaxes are valid and mean the same. However, in C an empty parentheses [not using the keyword `void`] would disable all type checking with respect to this function call and this allows the function to be called with any number or type of argument.

3.8 Inline Functions

Function calls lead to transfer of control and return. This transfer or return of control can cause substantial execution time overhead. To avoid this transfer or return of program control C++ provides inline functions. A function that has been declared to be inline in the function header of a function definition instructs the compiler not to transfer control while paste the function definition at the invocation or call point.

The trade-off is that multiple copies of the same code get inserted at its various invocation points rather than maintaining a simple copy and transfer or return control to or from it. The inline qualifier is chosen normally for smaller functions (small in terms of lines of code of function definition). If used with larger sized function it can improve upon execution time but increase the program size. It is a user- or programmer-dependent issue and it is he who should make a judicious choice between execution time or size trade-off.

3.9 Methods of Passing Arguments to Functions

Two common methods of passing arguments to functions or modules in many programming languages are pass by value and pass by reference. Each has its own merits and demerits, which we shall explore in detail in this section. When an argument is passed by value, the function definition creates a copy of the arguments passed (local copy) to it. References to variables or modifications are done over this duplicate copy of the variables or arguments defined in the caller function. In fact, two types of parameters associated with functions are formal and actual parameters.

Formal parameters are the variable names in the function header of function definition for the purpose of referring to within the function definition because functions do not know the actual arguments or variables until they are called. Thus, the parameters with which a function are gets called in the caller or boss function are called actual parameters, while the parameters that are used in the function header of function definition for manipulation within the definition are called formal parameters.

Thus, in pass by value a duplicate copy of the arguments passed is created and it is this local copy that is referenced within the function definition. For example, a function `swap (a,b)` [intended to exchange the values of `a` and `b`] in which `a` and `b` are actual parameters creates a duplicate copy of `a` and `b` when mapped with the formal parameters of the function definition. The swapping is done over this duplicate copy and as a result the swapped values are reflected within the function definition and not in the caller function. The C code to swap by value is shown in Figure 3.12.

Thus, it can be seen clearly that the swapping effect is reflected only within function definition [the `printf` statement within the function swap] and not reflected in the caller or main function [as shown by the `printf` at line 9]. The advantage with pass by value mechanism is that changes are made only over the copy and do not affect the original values in caller. This prevents the accidental side effects that are so much crucial to good program development.

```
1. #include<stdio.h>
2. void swap (int,int);
3. int main()
4. {
5. int number1,number2;
6. printf("Enter values for number1,number2 \n");
7. scanf ("%d %d",&number1,&number2);
8. swap(number1,number2);
9. printf('The numbers after calling swap in caller function
   is %d %d',number1,number2);
10. return 0;
11. }
12. void swap ( int forno1,int forno2)
13. {
14. int temp;
15. temp = forno1;
16. forno1=forno2;
17. forno=temp;
18. printf("Swapped value in function swap is %d %d
    \n",forno1,forno2);
19. }
```

Output:

Enter values for number1, number 2

20 30

Swapped Values in function swap is 30 20

Figure 3.12: C code to perform swap by value.

The disadvantage is that creation of duplicate copies can cause reasonable wastage of memory space.

Thus, it is the programmer who has to decide which is more important. In cases where memory is important, the second mechanism of pass by reference (address) is used. In this case, the address of the actual parameter is made available for the called function, that is it can access the data directly and hence modifications or updations are reflected in both function definition and the caller function. In fact, there is only one copy that is maintained in the caller function. The `swap` function when coded by pass by reference looks as shown in Figure 3.13.

The outputs clearly reflect the modifications in this case, swapping is reflected not only in the function definition but in the caller function `main` as well, since there is only one copy of `number1` and `number2` that is referenced within the function definition `swap()` and in `main()`. Pass by references are advantageous in that there is no wastage of memory since there is no duplicate copy creation. But the disadvantage is that the original variable is modified and hence accidental changes [side effects] can prove to be a bottleneck.

```
1. #include<stdio.h>
2. void swap (int &, int &);
3. int main()
4. {
5. int number1,number2;
6. printf("Enter values for number1,number2 \n");
7. scanf("%d %d",&number1,&number2);
8. swap(number1,number2);
9. printf("The numbers after calling swap in caller function is
   %d %d",number1,number2);
10. return 0;
11. }
12. void swap ( int &forno1,int *&forno2)
13. {
14. int temp;
15. temp = forno1;
16. forno1=forno2;
17. forno2=temp;
18. printf("Swapped value in function swap is %d %d
   \n",forno1,forno2);
19. }
```

Output:
Enter values for number1, number 2
20 30
Swapped Values in function swap is 30 20
The numbers after calling swap in caller function is 30 20.

Figure 3.13: C code to perform swap by reference.

Note: The advantages of both pass by value and pass by reference can be combined by placing the `const` qualifier before the arguments passed by reference. Such a mechanism would not waste memory since there is no duplication and it being a constant reference, modifications (accidental changes—side effects] will as well not be allowed. This is a syntactic exploitation of the provisions in C and is not a separate passing mechanism.

The following function declaration `Void fn2 (const int & a1,const int &a2)` provides only read access over arguments `a1` and `a2` for `function fn2`. This can be made use of when the called or worker or helper functions need only read permissions over the caller functions variables and will not perform any modifications over it. Hence the following conventions are widely adopted while passing arguments:

1. Small, non-modifiable arguments are passed by value.
2. Large-numbered, non-modifiable arguments are passed as constant references.

3.10 Aliases

References can also be used as aliases for other variables within a function. Reference variables must be initialized in their declaration.

For example:

```
int b =1;
int &a =b;//alias for b
++a; //increment b using alias.
```

Both `a` and `b` refer to the same location. The alias is simply another name for the original variable and all operations supposedly performed on alias are performed actually on the original variable. A reference argument must be an lvalue and not a constant or expressions that return a value.

3.10.1 Possible Coding Errors with References

A few of the possible coding errors involving reference data and manipulation over reference variables are as follows:

1. Not initializing a reference variable at declaration point is syntax error.
2. Declaring multiple references in one statement and assuming that & distributes across a comma-separated list might cause programming errors. For example, `int &a,b`, declares `a` to be a reference variable and `b` to be a normal integer variable. If `b` should also be a reference variable then `&` should be explicitly applied to it like `int &a,&b`. Assuming that `&` distributes across a list of variables declaration might cause logical errors since there would be a mismatch between the way the compiler and the programmer treat the variable. For example, in a statement `int &a,b`, the compiler treats `a` as a reference variable and `b` as a normal variable while the programmer assuming that `&` distributes treats `b` also as a reference variable. Hence, programmers should stick to the syntax of explicitly applying (prepending) the `&` operator for reference variables before their names or identifiers.
3. Functions can also return references but this should be done with extra care. When returning a reference to a variable declared in the called function, the variable should be declared static within that function, otherwise, the reference refers to an automatic variable that is destroyed or discarded when the function terminates. Such a reference variable is said to be undefined and the programs behaviour would be unexpected. In fact, such references are also called dangling or dancing references. One more logical error is attempting to reassign a previously declared reference to be an alias for another variable. This would just copy the other variable's value to the location for which the reference is already an alias. Returning dangling references will not most often be caught at compilation stage, it leads to serious logical errors.

3.11 Default Arguments

We have already seen that caller functions pass arguments to called functions. In certain cases, the programmer can specify that the function is called with default arguments if arguments are not explicitly specified by the caller of the function. Such arguments called default arguments when omitted by the caller function are automatically replaced by the compiler and passed in the call. Default arguments must be the rightmost or trailing arguments in a function's parameter list.

When calling a function with two or more default arguments, if a missed argument is not the rightmost argument in the argument list, all the arguments to the right of that missed arguments must also be omitted. Default arguments should be specified with the first occurrence of the function name mostly the prototype. Default arguments can also be used with inline functions, constants and global variables. For better understanding of defaulting argument, let us look at a programming example for the same as shown in Figure 3.14.

```
1.  #include<stdio.h>
2.  int simpleinterest(int =1,int =1,int=1);
3.  int main()
4.  {
5.  int si;
6.  printf("Simple Interest function called with no user specified
    and all default values");
7.  si=simpleinterest();
8.  printf("Simple Interest function called with 1 user specified
    and 2 default values");
9.  si=simpleinterest(10);
10. printf("Simple Interest function called with 2 user specified
    and 1 default value");
11. si=simpleinterest(10,2);
12. printf("Simple Interest function called with 3 user specified
    and no default values");
13. si=simpleinterest(10,5,3);
14. return 0;
15. }
16. int simpleinterest(int prin,int yrs,int rate)
17. {
18. return prin*yrs*rate;
19. }
```

Figure 3.14: *C* code to demonstrate default arguments to functions.

Note that in lines 6, 8 and 10, the arguments with missing values are replaced with default values as specified in the function header of function prototype. In the first case (line 6), all three arguments are missing and replaced by default values. In the second case (line 8), one argument (`prin`) is specified by the caller and the remaining two arguments are defaulted. In the third case (line 10), two values are caller specified and the third argument is defaulted. In the last case (line 13), all the three arguments are user or caller specified and there is no defaulting that is performed by the compiler.

3.12 Global Variable Access

We have already seen that functions can declare variables local within them (local variables) or refer to variables defined or declared outside, called global variables. We have also seen that

or function header of function definition encodes the function name or identifier and data types of its parameters. This process is normally referred to as name mangling or name decoration and this ensures type safe linkage, to ensure that a function gets called with the correct data type(s).

Type safe linkage ensures that the correct overloaded function confirming to the argument types is invoked. Overloaded functions can have different return types but must mandatorily differ in their parameter lists. Let us now see a programming example for better understanding of function overloading. To compute the cube of an integer and float argument passed to a function called cube(), the C code is as shown in Figure 3.16.

```
1.  #include<stdio.h>
2.  int cube (int=1);
3.  float (float=1.0);
4.  int main()
5.  {
6.  int intno,floatno;
7.  printf("Enter the values for integer and float number
\n");
8.  scanf("%d %f",&intno,&floatno);
9.  printf("Cube of the integer no %d is
%d",intno,cube(intno));
10. printf("Cube of the float no %f is
%f,intno,cube(floatno));
11. return 0;
12. }
13. int cube(int x)
14. {
15. return x*x*x;
16. }
17. float cube(float y)
18. {
19. return y*y*y;
20. }
```

Figure 3.16: C code to demonstrate function overloading.

The compiler uses only the parameter lists to resolve overloaded function calls. Hence functions with identical parameters, but list differing in return types, will not be function overloading. It only leads to syntax errors. There is no compulsion that overloaded functions must match in the number of parameters, but they definitely need to differ in the data types or precisely the signature. However, care should be taken while using function overloading with function defaulting values for its arguments.

A function with default arguments omitted may be identical to some other overloaded function, in which case it will be a syntax error. Of course, it should lead to a syntax error for the compiler would not be able to distinguish the two functions if this is allowed. Also

a program in which there is a function accepting no arguments (declared explicitly as not accepting any arguments and there is another function that contains all default arguments. This again will cause syntax error for the same reason mentioned earlier. Therefore, in a call of the function that has no arguments specified by the caller, the compiler will not be able to distinguish or resolve the two function definitions.

3.14 Function Templates

Well, function overloading was used to deal with functions that perform similar operations but involved different logic on differing data types. Now what if the case when the programming logic is also the same and it is only in the data types that such functions differ. This is an ideal situation for use of function templates even though it can be very well achieved using function overloading as well. But function templates provide a more compact and convenient solution for achieving the same.

The programmer writes a single function template definition, the generic function not relating to any specific data type. This function template is the generic specification of the programming logic involved with the function. Having completed the generic function specification, the compiler automatically generates separate template functions [functions generated out of or from function templates] to handle function calls differing in data types appropriately.

The compiler performs code generation on behalf of the programmer. Hence effectively the lines of code of the function definition are duplicated for the respective data type, or in other words, it is function overloading behind the scenes that is performed by templates with the difference being that the programmer need not specify the definitions for all the overloaded versions. Whereas he needs to complete the function definition for a generic data type and leave it to the compiler to generate data type specific definition of the function.

Thus, defining a single function template defines a whole or entire family of functions. There is no performance enhancement with the use of templates in comparison with user overloaded functions. It is just an efficient code generation strategy and nothing more than that. However, it makes programs more compact and readable. All function templates definitions begin with the keyword templates followed by a list of formal parameters as we had with normal function definitions. The formal parameter list is enclosed within <> angle brackets. The formal parameters could be built in or user defined types [structures or classes which shall be dealt with later], then the function definition follows which is similar to normal function definition. Let us now look at an example to swap two numbers, code for which is as shown in Figure 3.17. These numbers could be integers or float or double.

The above function template declares a single formal type parameter T. Then for each specific invocation of swap, the compiler substitutes the T in the function template specification with actual data type passed. This generated function (template function) is then compiled. As we have already seen it is only code generation performed by the compiler on the programmer's behalf. Once the codes have been generated for each separate invocation of the function template or when all the template functions have been generated from there on it is based on function overloading concepts that further actions take place.

```
 1. #include<stdio.h>
 2. template <class T>
 3. void swap (T &v1 ,T &v2 )
 4. {T temp;
 5. Temp =*v1;
 6. *v1=*v2;
 7. *v2=Temp; }
 8. int main()
 9. {int int1,int2;
10. float no1,no2;
11. char a1,a2;
12. printf("Enter the integer numbers \n");
13. scanf("%d %d",&int1,&int2);
14. printf("Enter the float numbers \n");
15. scanf("%f %f",&no1,&no2);
16. printf("Enter the characters \n");
17. scanf("%c %c",&a1,&a2);
18. swap(int1,int2);
19. printf("Integers after swapping are %d %d",int1,int2);
20. swap(no1,no2);
21. printf("Float numbers after swapping are %f %f",no1,no2);
22. swap(a1,a2);
23. printf("Characters after swapping are %c %c",a1,a2);
24. return 0;}
```

Figure 3.17: C code to illustrate the use of function templates.

Review Questions

1. List the benefits of a function-based code development approach.
2. Differentiate the terms function declaration and function definition.
3. Differentiate an interface from implementation.
4. Develop a function cube which returns the cube value of the integer argument passed to it.
5. Develop a C program that uses functions to test if an input square matrix is a magic square or not. (Note a magic square is a square matrix in which the sum of all columns, sum of all rows, sum of main and trailing diagonals all equate to the same value.)
6. Develop a C program that uses functions to generate a magic square of a user-specified order.
7. Develop a C program that generates the determinant value of a square matrix.
8. Differentiate a static variable from an automatic variable.
9. Identify a few coding situations when the choice of registers is justified.

10. What do you understand by the term scope of a variable?
11. Differentiate scope from storage class.
12. Can you redeclare a variable in C? Justify your answer.
13. Differentiate recursion from iteration.
14. What is the mathematical principle that forms the basis of recursion? Explain with respect to the problem of factorial computation.
15. Develop a C program using recursion to solve the Towers of Hanoi problem. (There are three pegs, A, B and C. A is the source peg and C is the destination. A is initially composed of n disks with the smallest one on top and the largest in terms of size at the bottom. Objective is to transfer the disks from A to C using B as the intermediate peg, always satisfying the condition that a smaller disk is placed on top of a larger one.)
16. Differentiate call by value from call by reference with an example.
17. Explain the notion of default arguments supported with functions in C++.
18. Develop a C(/C++) program to generate the product of three numbers accepted from the user. The user could enter all integers or all floating point numbers. Use the concept of function overloading.
19. Argue the case for function templates over function overloading. Develop a function template to solve the problem defined in the earlier question.
20. Develop a C program using functions to compute the greatest common divisor of two numbers passed to it.

Chapter 4

Arrays

Arrays are data structures consisting of related items or collection of items of the same data type. An array is a consecutive group or continuous group of memory locations that contain elements of the same data type and are referenced with the same name. To retrieve or access a particular memory location or to access a specific array element, the array name and the position, commonly referred to as index is specified. In C++, array indexing or element numbering starts from 0. Thus, `a[2]` would be the syntax to refer or retrieve the third element of the array named assuming indexing is from 0.

In general `a[i]`, the argument `i` within the square brackets is referred to as index or subscript of the array. Index should always be an integer or integer expression. The array name is an `lvalue` and can be used on the LHS of an assignment statement. As we have already seen arrays when declared occupy memory space, to be more precise they occupy contiguous memory blocks or locations. For an array declaration statement in C/C++, the compiler requires an array name, data type of elements that shall be stored in the array and the number of elements [cells or locations] required for the array. For example, the following declaration statement `int a[20]` reserves 20 elements for integer array `(a[0]-a[19])`.

4.1 Array Declaration

The programmer can define an array of characters or floats, or doubles by making use of the appropriate keyword [`char` or `float` or `double`] in the declaration statement. Values for arrays can either be accepted from the user via an input statement or initialized using an assignment statement. The two syntaxes for input of array elements for an array `a` of size `5` is as follows:

```
int a[5];
User Input:
int i;
for(i=0;i<5;i++)
scanf("%d",&a[i]);
Assignment Statement:  int a[5]= {0,5,6,7,8};
```

4.1.1 The First Syntax

Since it is more than one value [in fact, 5 values] that is to be read in from the user, we use a `for loop` with the `scanf()` function to accept 5 values from the user. The `%d` indicates that

the types of elements to be read in are all of integer data types and remember that with array indexing from 0, the `for loop` that runs from 0–4 actually iterates five times, i.e. the values of integer types are read in from the user.

4.1.2 The Second Syntax

Array element if known at compilation itself can be initialized with assignment statement and flower brace syntax as shown above. If there are fewer initializers than the elements of the array, then the compiler automatically initializes the remaining or pending to be assigned elements to zero. For example, in the statement `int n[10] = {0,5,6}`, elements 3–9 are automatically initialized to zero.

As another example `int b[5]= 0` initializes the first element to 09 and the remaining four elements are implicitly assigned 0. However, the programmer must explicitly initialize the first element for the remaining elements to be zeroed. However, additional initializers such as in `int a[5]={10,20,30,40,50,60,70}` where there 7 initializers for 5 array elements would cause a syntax error. Another syntactic provision is that the array size need not be specified if the initializer is available at the compile stage itself. That is to say the following statement `int a[]={5,4,3,2,1}`, creates an array of 5 elements and initializes to the value in the list.

Arrays getting initialized at compile time or with assignments contribute to performance improvement, in terms of execution time, than arrays that need to be accepted from the user. A simple program for better understanding of the above concepts is given below. The following program initializes an array of 5 elements with values of 1–5 and computes the 4th power of each array element and stores in the resultant array result, code for which is shown in Figure 4.1.

```
1. #include<stdio.h>
2. #include<math.h>
3. int main ()
4. {
5. int size=5,i;
6. int input[size]={1,2,3,4,5}, result[size];
7. for(i=0;i<size;i++)
8. {
9. result[i]=pow(input[i],4);
10. printf("%d %d \n",input[i],result[i]);
11. }
12. return 0;
13. }
```

Figure 4.1: C code using arrays.

Observe that the size of the array is stored in a variable, so that for further scaling (array sizes varying but same problem on hand), the modification needs to be done only one line while hard coding the size directly in the loop or other statements would require that many statements or lines of code to be modified. This is purely a scalability issue and creates programs that are easier to maintain.

4.1.3 Syntactic Variations

Another syntactic exploit with arrays is to make the size variable a constant (keyword `const` preceding the `int` keyword) with the assumption that array sizes remain non-modifiable once program execution begins. However, programmers need to take care while manipulating arrays that they do not walk off either end of an array. Walking off either end of an array or commonly referred to as an array out of bounds is the situation where the programmer is referring to indexes that are not valid.

For example, the declaration `int a[3]={0,1,2}`, the programmer referring to `a[-1]` or `a[3]` is walking of either ends[end refers to positive or negative scale movement] exceeding the defined limits of the array. Such errors are not caught at compilation stage and hence the programmer should be careful enough not to indulge in such poor programming practices. If not, he is in for serious logical errors. Readers may wonder how array indices can proceed in the negative direction. Yes, it is a real possibility when looping through an array; the loop counter variable should never fall below 0 and should always be less than the number of array elements.

4.2 Sorting an Array of Numbers

In the following example, we shall develop a program to sort an array of 5 numbers with compile time initialization and then sort them either in ascending order. The sorting algorithm we will employ is the naive bubble sort algorithm. It is naive in terms of a higher time complexity associated with it. But then this is the most natural form of sorting and which humans are more comfortable.

Bubble sort is based on the principle of comparing of adjacent pair of numbers and exchange or swap them if they are out of order or not in the ascending or descending order as required. This comparison is repeated for the next pair until the last number in the sequence is reached at which stage one pas is over. For an array of n elements there are `n-1` comparisons within each pass and `n-1` passes on the whole required to sort the input array. For better understanding let us assume we have the following 5 element input array that is to be sorted in ascending order. The various passes are as shown in Table 4.1.

Table 4.1: Bubble sort passes

Input:
5 4 3 2 1
Expected output: 1 2 3 4 5
Pass 1: 4 3 2 1 5
Pass 2: 3 2 1 4 5
Pass 3: 2 1 3 4 5
Pass 4: 1 2 3 4 5

Thus, in all 4 passes and 4 comparisons within each pass are required to sort the initial input array.

4.2.1 The Code

We shall not modularize this code, it is with purpose. At a later stage with discussions of pointers, we shall explore how the same code can be modularized when passing by reference [using pointers] would be required. The code to perform bubble sort is shown in Figure 4.2.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  const int arrsize=5;
5.  int input[arrsize]={5,4,3,2,1},pass,temp;
6.  printf("Bubble Sort \n");
7.  for (pass=0;pass<arrsize-1;pass++)
8.  for(j=0;j<arrsize-1;j++)
9.  if (input [j]>input [j+1])
10. {
11. temp = input [j];
12. input[j]=input[j+1];
13. input[j+1]=temp;
14. }
15. printf("The Sorted Array is \n");
16. for(i=0;i<arrsize-1;i++)
17. printf("%d \t",input[i]);
18. return 0;
19. }
```

Figure 4.2: C code to perform bubble sort.

We adopt an array initialization purely to avoid additional code. Readers who wish to read input from the user they should have appropriate `scanf` within `for loop`. It is in fact lines 7–14 that actually perform the sorting in ascending order. Here the swapping is done if the adjacent elements of the array are not in the required ascending or descending order.

Note: As an interesting coding logic swapping can also be achieved without using a temporary variable (third variable). The following two code snippets shown in Tables 4.2 and 4.3 perform swapping without using a temporary variable. Let us assume that the numbers to be swapped are in variables a and b and their values are 1 and 2.

Table 4.2: Swapping without temporary variables (code1)

First code:	*Effect*
a = a + b;	a = 1 + 2 = 3;
b = a - b;	b = 3 - 2 = 1;
a = a - b;	a = 3 - 1 = 2;

The ^carat symbol stands for Exclusive OR operation. This is also referred to as the logical bitwise XOR operator. To understand the above logic we need the bitwise representations of

Table 4.3: Swapping without temporary variables (code 2)

Second code:
a = a^b;
b = a^b;
a = a^b;
Trace through:
a = 1 = 0001 ; b = 2 = 0010
a = a^b = 0001^0010 = 0011 ;
b = a^b = 0011^0010 = 0001; (b = 1)
a = a^b = 0011^0001 = 0010; (a = 2)

a and b. Let us assume that it is a 4-bit representation [this suffices for our input] that is followed. The effect of the above statements is as shown in Table 4.3.

4.2.2 XOR

The output is high or 1 if and only if one of its input is high [exclusive highness with respect to the input]. This differs from the inclusive OR operation [+] in that the condition to be satisfied for + is that at least one of its input should be high for the output to be high. For easier understanding, the truth tables of both logical operations, exclusive OR and inclusive OR are shown in Tables 4.4 and 4.5.

Table 4.4: Truth table for exclusive OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.5: Truth table for inclusive OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

4.3 String Manipulations

A string is formally defined as a sequence of characters or an array of characters. C++ provides a string class. But it will be explored when we actually get into OOP. Here, in structured programming we shall manipulate strings treating them as an array of characters. Possible string manipulations are finding the length of a string, string reversal, string concatenation, palindrome checking, etc.

In fact, for most string manipulations predefined functions from the standard library are available. But then, in this section we will be more interested in defining these functions. Definitely not to contradict our earlier statement that one should not reinvent the wheel. It is from a learning point of view rather from performance that we define these functions. In the process one gets exposed to the various issues or niceties in structured programming. In fact, end users here are readers and not software users. The following code in Figure 4.3 performs the various string manipulations of concatenation, string comparison and length of string.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int strlength(char []);
void stringcopy(char [],char[]);
void stringconcat(char [], char []);
void strrev(char []);
int equal(char[],char[]);
char gloarr[200];
int main ()
{
int yes,concatchoice,choice,strlen=0,again=1;
char str1[200],str2[200],str[200];
char dest[50];
while (again)
{
system("clear");
printf("\n What Do U want to Do");
printf("\n 1. Determine the String Length \n 2. Copy a String \n
3. Concatenation \n 4. String Reversal \n 5. String Equality Check \n
printf("Enter your choice (1 -6 any number) \n");
scanf ("%d",&choice);
switch(choice)
{
case 1: system("clear");
printf("String Length Determination ...\n");
printf("Enter the String for which the length is to be determiend \n");
scanf ("%s",str);
strlen = strlength(str);
printf("The length of %s is %d",str,strlen);
printf("\n Exiting String Length Determination");
printf("\n Do U want to Continue with Other Operations(1-Yes,0-No)");
scanf("%d",&again);
break;
case 2: system("clear");
printf("Trying to Copy a String \n);
printf("\n Enter the Source String \n");
scanf("%s",str1);
stringcopy(str1,str2);
printf("The Copied String is %s",str2);
break;
```

Figure 4.3: Continued

```
case 3: system("clear");
printf("\n Enter the First String \n");
scanf("%s",str1);
printf("\n Enter the Second String \n");
scanf("%s",str2);
printf("What do you want to do \n");
printf("Concatenate First to Second(Input 1) Or
Second to First (Input 2 \n");
printf("Enter Your Concatenation Choice \n");
scanf("%d",&concatchoice);
if(concatchoice==1)
{
stringconcat(str1,str2);
}
else if(concatchoice==2)
{
stringconcat(str2,str1);
}
else
{
printf("\n Not A Valid Sub Choice");
printf("Returning to Main Menu \n");
}
break;
case 4: system("clear");
printf("String Reversal \n");
printf("Enter The String to be Reversed \n");
scanf("%s",str);
strrev(str);
break;
case 5: printf("\n Equality Checking");
printf("Enter the string 1");
scanf("%s",str1);
printf("\n Enter the string 2");
scanf("%s",str2);
yes = equal(str1,str2);
if(yes==1)
printf("Equal");
else
printf("Not Equal");
break;
default:printf("\n Invalid Option");
printf("\n Quitting.... Pgm Execution");
}
}
return 0;
}
int strlength(char forstr[])
{
int i=0;
while (forstr[i]!= '\0')
```

Figure 4.3: Continued

```
i++;
return i;
}
void stringcopy(char source[],char dest[])
{
int i=0;
while (source[i]!='\0')
{
dest[i] = source[i];
i++;
}
dest[i]='\0';
return;
}
void stringconcat(char string1[],char string2[])
{
int loclength,destindex;
int i=0,j=0;
loclength = strlength(string2);
destindex = loclength+1;
printf("\n Local Length is %d",loclength);
printf("\n Dest Index is %d",destindex);
while (string1[i]!='\0')
{
string2[destindex]=string1[i];
printf("\n Copied Char is %c \n",string2[destindex]);
i++;
destindex++;
}
string2[destindex]='\0';
printf("\n The Concatenated String is \n");
for(j=0;j<destindex;j++)
printf("%c",string2[j]);
return;
}
void strrev(char input[])
{
int i=0,j=0;
int loclen = strlength(input);
printf("Rev Length is %d",loclen);
while (loclen¿=0)
{
gloarr[i]=input[loclen];
i++;
loclen-;
}
gloarr[i+1]='\0';
printf("The Reversed String is \n");
for(j=0;j<i;j++)
printf("%c",gloarr[j]);
```

Figure 4.3: Continued

```
return;
}
int equal(char s1[],char s2[])
{
int i;
int len1 = strlength(s1);
int len2=strlength(s2);
if(len1!=len2)
return 0;
for(i=0;i<len1;i++)
if(s1[i]!=s2[i])
return 0;
return 1;
}
```

Figure 4.3: C code for string manipulations.

Storage classes are applicable with array declarations as well. Local automatic array variables are created and destroyed each time the block in which they exist are entered and exited. Also, automatic arrays if not initialized by the programmer or user get initialized with junk or garbage values. Static arrays are created when the block in which they are declared is entered first. The next time program control enters the same block; allocations or initializations are not performed from scratch. All these concepts which we explored in storage classes (Chapter 3) remain intact with respect to arrays as well. Arrays that will have write (assignment) operation performed on them only once should be declared as constant. Constant variables can be assigned or initialized with values once. Such array variables or identifiers should be suitably declared as constant arrays.

4.4 Passing Arrays to Functions

Arrays can be passed to functions by specifying the array name without any brackets (`[ ]`). An array with the following declaration `int a [3]` is passed to a function (let us assume the function is `fn1`) with the following syntax of `fn1(a,3);` where `3` represents the size of the array passed to `fn1()`. If the array size is not passed as an argument, then there should be a mechanism of making this information known to the called function, because the called function will need this information to process specific number of elements of the array within its definition. One inefficient solution would be to make the array size a global variable, but then this would have its own bottlenecks of being susceptible to accidental changes.

C++ automatically passes arrays to functions using simulated call by reference. The called function can modify the array element values in the caller functions original array. The name of the array refers to the address of the first element of the array (base address), since the called function is aware of the arrays starting address, manipulating the array element values within the called function, manipulates or modifies the actual elements of the array in the calling function (original memory location).

The reasoning behind allowing automatic calls by reference is that if they were allowed to be passed by value, then the entire array (with all its elements) needs to be duplicated. This can be acceptable when the array size is small. But for realistic large arrays, this would

consume or occupy storage space. However, individual array elements if required can be passed by value as is done with simple variables for which the subscripted name of array element is passed as an argument to the called function. The function header for a function that is to receive arrays should explicitly indicate that an array is being passed. For example the function header for function `fn1()` that is to receive an array `a` of size `arr_size` would be as follows:

```
void fn1 (int a [ ], int arr_size);
```

The size of the array if specified between the square brackets is simply ignored by the compiler. The function header for the above function can also be specified excluding the array name or array identifier and the `arr_size` identifier because these serve only as placeholders and it is the data type that is important. For code integrity and functions that need only read permissions over the array, the array and the array size can be passed as constant values. Such arrays remain non-modifiable within the function definition, attempting to do so causes syntax errors. Let us now implement Linear and Binary Search using arrays. Even before we get on with the code, let us briefly review the Linear and Binary Search Algorithms.

4.5 Linear Search

Linear search this is also referred to as sequential search. The entire array starting from the first position is searched for if the element for which the user wants to know the position is present or not. If the element is found, the search is terminated and the index or position is reported. This procedure is repeated in a sequential fashion till the end of the array is reached. If the comparison yields no match when the array end is reached, it means that the element is not finding the list and the same is reported. As is clear from the strategy the search time could be high if the element that is being for is in the last position of the array. The time complexity of linear search is high. [Beginners need not get disturbed]. We shall towards the end of this section establish the time complexities of linear and binary search. Thus, linear search cannot always be the choice.

4.6 Binary Search

The binary search algorithm is efficient in comparison with linear search in terms of time complexity. The binary search algorithm assumes or expects the input array to be sorted in ascending order. The binary search algorithm maintains three indices low, middle and high throughout the strategy. To perform binary search over the entire array, the starting values of low and high are set to 0[first position] and array size 1 [last position]. The middle position or index is computed as follows:

```
middle = (low + high)/2
```

Binary search is based on the principle of reducing the search space [by a factor of 2] after a comparison is done. If the element being searched for matches with the middle element of the array, then the middle position is returned and the algorithm terminates. If the element being searched for is less than the middle element of the array, then it means that the element, which the user is searching for, is lying on the LHS of the middle position. Now, the further

searching operations are performed only on the LHS of the middle position of the array, which means that the searching space should be resized.

The low and high indices which normally denote the first and last position of the array would be chanted to reflect the reduced subarray. The low index that is pointing to the first position should remain the same and the high should be set to position one less than the middle position [high = mid-1] [since the element being searched for is less than middle element of the array]. The last condition is when the element being searched for is greater than the middle element of the array, then it means that the element, which the user is looking for, is lying on the RS of the middle position of the array.

The high index which is pointing to the last position should remain the same and the low index should be one greater than the middle position or low = mid + 1. The resizing of the array and binary search over the resized array is repeated till the element being searched for is found or until the subarray consists of only one element [length1] and the element in subarray is not the search element. [The case when the element being searched for is not found on the array.] Let us assume that we start with an initial array of length or size 10 and the element being searched for is in the last position. The number of comparisons required with linear search will be 10 to locate the element, whereas with binary search it is only 4 comparisons. [Readers are advised to trace through the algorithm.] When it is the case with a small size of n=10, one can imagine the drastic reduction in the search space and hence the number of comparisons required with really large arrays. For better understanding of let us assume the initial input array is as follows:

1 2 3 4 5 6 7 8 9 10

and the element being searched for is 10. [last position 9]. The various iterations/passes of binary search over the array are shown in Figure 4.4. Hence the element 10 is found at the 9th position in the array. That is to say, the binary search algorithm returns 9 as the position of the element being searched.

1. Pass 1: low =0, high = 9, middle = 4, input[middle] = 5; search element (10) is greater than input [middle]. Thus low = 4 + 1 and high = 9;
2. Pass 2:low =5, high = 9, middle = 7, input[middle] = 8; search element (10) is greater than input [middle]. Thus, low = 7 + 1 and high = 9;
3. Pass 3:low =8, high = 9, middle = 8, input[middle] = 9; search element (10) is greater than input [middle]. Thus low = 8 + 1 and high = 9;
4. Pass 4:low =9, high = 9 , middle = 9, input[middle] = 10; search element (10) is equal to input [middle].

Figure 4.4: Binary search algorithm passes over the input array.

4.7 Time Complexity

Algorithms get evaluated based on time and space complexity. Time complexity represents the total time involved in the execution of an algorithm or to be literal a program. Time complexity is normally dependent on the size of the input being processed by the algorithm. But then there is no golden rule that time complexity will always be dependent on the input size. In certain cases such as number multiplication, it is the number of bits used to represent the numbers that determine the time complexity and not the number (magnitude). For our searching problem, it is the array size that has a direct role to play in the final complexity.

Thus, the time complexity of linear and binary search is directly proportional to the number of comparisons made or required to locate the element being searched. Three cases that are most often considered with time complexity analysis are: the best, average, and worst cases. The best case denotes the input sequence or input for which the algorithm executes in the minimal time possible. The worst case as obvious denotes the input for which the algorithm executes in the maximal time possible. The average case denotes the input combination (sequence) for which the time is somewhere near the average of the best and worst cases.

Symbolic notations associated with time complexities are O (Big O), ω(Omega) and θ(Theta). They are used to represent the worst, best and average time complexities respectively. In terms of values they can assume, they are also referred to as upper, lower and average bounds. Most often it is the worst-case complexities that are computed for this is indicatory of the fact that the algorithm will not take any more than the computed complexity or time (for this is the worst possible input sequence). It is in fact an upper bound on the number of comparison required. But then a true algorithm analysis would also compute average and best case complexities as well. The best case denotes that the algorithm will take at least the computed time when the input is the best possible.

Average case as we have already stated is an intermediate input sequence which falls between the best and the worst cases or one can interpret that the average case almost 50% of the input resembles the best case input sequence while the remaining input resembles the worst case input sequence. In terms of bound, best case represents the at least (`>=`), worst case represents the at most (`<=`) and average case represents the intermediate (`=`) time required to execute the algorithm. Best case for searching is when the element being searched is the first array element. Worst case is when the element being searched is the last element in the array. Average case is intermediate case between best and average or the element being searched is after the first and before the last element.

4.7.1 Linear Search

Best case: 1 comparison required.
Worst Case: n comparisons required.
Average case: In terms of a generic input size, this would also be some integer (`>1` and `< n`).

4.7.2 Binary Search

O $(\log_2 n)$***:*** Logarithmic representations are used to represent time complexities of problems for which the problem size gets reduced by a sizeable factor or to be more precise the reduction follows an exponential distribution. Here, 2 represents the factor by which the problem size

gets reduced with each pass. For an array of size $\wedge$ assume K is the number of times 'repeated division' is performed, binary search could be expressed as $\frac{\wedge}{2k} > 1$. (The array is repeatedly divided until array size is 1). Therefore $\wedge > 2^k \Rightarrow \log_2^n > k \Rightarrow K \leq \log_2^n$. Readers are advised to refer to *Introduction to Algorithms by Thomas*, H. Cormen for a detailed treatment on *Algorithm Analysis*. Now let us get into the coding of Linear and Binary Search algorithms, details of which are shown in Figure 4.5.

4.8 Two and Multidimensional Arrays

Arrays in C and C++ can have multiple subscripts. To start with the basic case of array subscript = 2 which represents two-dimensional arrays are used to represent tabular information consisting of m rows and n columns. Two-dimensional arrays are also referred to as double subscripted arrays. The upper bound on the number of arrays subscripts in modern C++ compilers is around 12. In general, any two-dimensional array consisting of m rows and n columns is represented as m*n or m by n array. The following declaration statement declares a two-dimensional array consisting of 5 rows and 4 columns:

```
int a[5][4]
```

Again in multiple subscripted arrays as well array indexing is from 0. Every element in array a is referenced by a generic syntax of `a[i][j]; i & j` being the subscripts that uniquely identify an element in `a`; `i` normally denotes the row and `j` represents the column subscript.

All the elements of the first row have a first or row subscript of 0. Similarly, all the elements of the third column have a second or column subscript of 2. A multiple subscripted or double subscripted array can be initialized in its declaration almost similar to single subscript arrays. For example a two-dimensional array `a[3][2]` could be declared and initialized as follows:

```
int a[3][2] = 1,2,0,5,6,7.
```

The values are grouped by rows in braces. Not enough initializers within a brace or a row initializes the remaining elements to 0. Let us look at each of the following three two-dimensional array declarations. Irrespective of the number of array subscripts the compiler stores these array elements in contiguous memory locations.

With two-dimensional arrays, the compiler proceeds in a row major fashion (one row after another). When multiple subscripted arrays are passed to function, the sizes of all array subscripts excluding the first array subscript is required. For with a two-dimensional array `a[3][5]` the array when passed to a function `fn1()`, the corresponding function header of `fn1` would be as follows:

```
void fn1(int a[][5])
```

The compiler uses these subscripts sizes to determine the memory locations of elements in multiple subscripted arrays. As we have already stated, all array elements are stored contiguously in memory.

In double subscripted arrays, the first row is stored in the first set of contiguous blocks of memory available and then followed up by the second row. These subscript values available in function declaration or header enable the compiler to decide on the position or location of an array element with double subscripted or two-dimensional arrays. Each row is a single subscript or one-dimensional array. To locate an element of a particular row, the number of

```
#include<stdio.h>
int linearserach(const int [ ], int, int);
int binarysearch(const int [ ], int , int , int ,int);
int main()
{
const int arrsize=10,i;
int input[arrsize], search element, position, search choice;
for(i=0;i;arrsize;i++)
scanf("%d",&input[i]);
printf("Enter the element for which u want ot find the position \n");
scanf ("%d",&search element);
printf("Enter your choice \n 1.Linear Search \n 2. Binary Serach \n 3.Exit
\n');
scanf("%d",&search choice);
switch (search choice)
{
case 1: printf("Linear Search in Progress \n");
position = linearsearch ( input,search element,arr size);
if(position>= 0)
printf("Element %d found at position %d \n",search element,position);
else
printf("Element %d not found in the list",search element);
break;
case 2: printf("Binary Search in progress");
position = binarysearch (input,search element,arr size);
if(position>= 0)
printf("Element %d found at position %d \n",search element,position);
else
printf("Element %d not found in the list",search element);
break;
case 3: printf("Quitting ..\n");
default: printf("Invalid Choice \n");
}
return 0;
}
int linearsearch (const int proarr[], int search key,int arsize) {
int i=0;
for(i=0;i<arsize;i++)
if(proarr[i]==search key)
return i;
return -1;
}
int binarysearch(const int prarr[],int serachk, int low, int high, int size)
{
int mid;
while (low <=high)
{
mid = (low + high )/2;
if(searchk==prarr[mid])
return mid;
else if (searchk< prarr[mid])
high = mid-1;
else
low = mid + q;
}
return -1;
}
```

Figure 4.5: C code to perform linear and binary search.

elements of each row should be known to skip the appropriate number of elements (memory locations) when accessing the particular element.

For example, accessing `a[1][2]` one should skip the first-three memory locations [skipping the first row's 3 elements] to reach the second row after which the third element or column of second to access element `a[1][2]`. In certain situations, two-dimensional or multidimensional arrays might be required to accept values or subject to further processing. The earlier syntax of looping adopted is applicable here as well. Have as many nested for or any other looping construct as you have array subscripts. For example, a two-dimensional array to have values input from the keyboard, the following syntax can be adopted. Let us assume that the array declaration is

```
int a[3][2];
for(i=0;i<3;i++)
for(j=0;j<2;j++)
scanf(''%d'',&a[i][j])
```

The loops execute in an inside out fashion or for an index i the inner loop runs from 0 to 1 [2 columns]. That is, the first row (0) elements consisting of 2 columns are initialized or assigned values that are input via the keyboard. Then the outer loop proceeds to the next i or a syntax matching our row major processing of the array.

4.9 Pointers in C/C++

Pointers are one of the most important and powerful features of C/C++. Pointers are normally used to create dynamic data structures (structures containing data that grow and shrink in size). Examples of such data structures include linked lists, queues, stacks. Pointers are also used to simulate call by reference. Pointer variables contain memory addresses or locations as their values. A variable (non-pointer) contains a value stored or assigned to it, whereas a pointer variable contains the address of another variable that has a particular value.

A variable name directly refers to or directly references a value, whereas a pointer variable indirectly references a value. On this basis, referencing a value via a pointer is also referred to as indirection. As has been the trend to declare variable before their use, pointer variables also have to be declared before they are used. The following declaration

```
int a=8,*b;
```

declares the variable a of integer data type and b to be a pointer to some integer data type.

Pointer declarations are better off to be read from right to left for better understanding. Every variable that should act as a pointer should have a * preceding its name and the * should be repeated with every variable name that is to act as a pointer variable. The * operator does not distribute itself with the variables following it, it is applicable only for the immediate variable, delimiter being the comma or the semicolon. For a better understanding of pointer concept, let us go through the following pictorial notation.

Let us assume the declaration is as follows:

```
int * apointer, b=8;
```

Let us assume that the memory locations assigned to `apointer` and `b` are 4000 and 5000. Thus, pictorially the above statement looks like as shown in Figure 4.6.

As we have already stated that the above declaration statement (pointer declaration) is interpreted right to left, as apointer is a variable that is pointing to some integer data type.

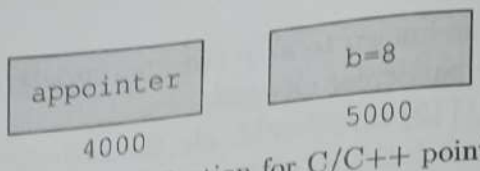

Figure 4.6: Illustration for C/C++ pointers.

But then at some stage or the other, pointer variables will be required to point to specific values. Let us assume in the above case apointer should point to b or apointer should point to a specific integer data type of b, the following assignment statement achieves it:

```
apointer = & b
```

Remember that apointer is a pointer variable and will have as its value the memory address of another variable of the declared data type. In this case apointer, which all along we said will point to some integer data type should point to the specific integer data type b or in other words, apointer should contain the address in memory where b resides. We have already seen that the & operator (ampersand) when applied with variable identifiers or names returns the address of the respective identifier or variable. By now the assignment statement should be clear. To summarize the assignment statement which evaluates right to left assigns or stores the address of b (&b) in the pointer variable apointer. Now that we have seen the syntax to make a pointer point to a specific value, the next requirement would be to refer that indirectly pointed value or what is called as referencing a pointer variable. Getting back to our pictorial representation which would now look like as shown in Figure 4.7.

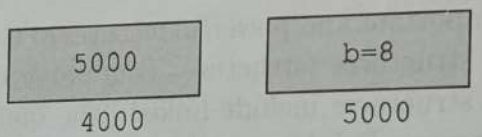

Figure 4.7: Illustration for C/C++ pointers.

To refer to the variables value pointed to by, the * operator is applied before the pointer variable name at places wherever we require the value pointed to. Lets assume we need to sum up the value pointed to by apointer and another non-pointer variable c=7 and store the result in sum; the following statement

```
sum = *apointer + c
```

would achieve the task where *apointer retrieves the value of b=8 pointed to by the apointer. This is referred to as indirection and the * operator is also called the indirection operator or dereferencing operator. The dereferenced or indirected pointer (*apointer) can also be used on the LHS of an assignment statement [the dereferenced pointer is infact an lvalue].

Pointers should be initialized either when they are declared or in an assignment statement. A pointer may be initialized with 0 or keyword NULL [to denote that is a null pointer or it is currently not pointing to a specific value] or with an address. The value 0 is the only integer value that can be assigned directly to apointer variable without casting the integer to the pointer data type. That is, a 0 being assigned to a pointer data type [which can be a char or float or int pointer] is converted to the appropriate pointer type automatically. The keyword NULL is defined in the standard library header files.

Dereferencing a pointer that has not been properly initialized might cause serious run time errors. The dereferencing operator cannot be applied with non-pointer variables and if applied causes syntax error. An interesting syntactic exploitation to be noted with pointer variables is

that the * (dereferencing) and & (address of) operator are inverses of one another, when they are both applied consecutively to a pointer variable in either order, the result is the address stored in or pointed to by the pointer variable. For example, `int *aptr,b=8; aptr = &b;` `*(&aptr)`, and `&(*aptr)` both yield the address of `b` that is stored in the pointer variable `aptr`. How come? The parenthesizing has been done for clarity

`*(&aptr)`: The address of operator returns the address of the `aptr` variable. The dereferencing operator dereferences the value contained at the specific address. We have already explained that a pointer variable contains the address of the variable to which it should point to. The result of `*(&aptr)` returns the address of `b`.

In the second expression of the form `&(*aptr)`, the * operator dereferences the value pointed to by `aptr` or value of `b=8`. The & operator then applied over this is nothing but the address of `b` getting returned. Thus, both expressions are the same.

4.10 Calling Functions by Reference

In the third Chapter on Functions, we have explained two methods of calling functions, one by reference using reference arguments and the other with actual values; or call by value and call by reference (using reference arguments). We shall now look at another mechanism of calling by reference (using pointers). Return statement is used to return a value to the caller function or simply return control back to caller function.

Passing arguments to called functions using reference arguments enabled or provided privilege for the called functions to modify the original values of arguments in the caller functions and thereby avoid the duplicating or copy creation overhead. In this section, we shall use pointer to simulate the reference passing of arguments to functions. Pointers and the indirection operator are used in combination to simulate the call by reference effect. Arrays are not passed using & operator as is done with normal variables since the array name refers to the starting or base address of the array. That is array name is equivalent to `&a[0]`, which implies that the array name is a pointer by itself. The indirection operator can be used in the function called to create an alias and can be used to modify the value (original) in the caller function and thereby simulate the pass by reference to exchange the values of two variables as done by the code shown in Figure 4.8. A function receiving an address as a argument defines a pointer parameter to receive the address [`a,b` in function swap receives the address of `no1` and `no2` declared in `main()`]. This is specified by the function header void `swap (int *, int *);` which denotes that the function swap would receive two arguments that are pointing to some integer data type or receive precisely address as argument. As has been stated previously pointer variable names are not compulsory in function headers as they serve only as place holders. The same syntactic exploitation can be used to pass arrays as well.

Array names, which are pointers to the base, address, such pointers being non-modifiable or constant entities. Thus, the syntax of `int a []` to pass a two-dimensional array automatically is converted by the compiler as `int * const a;` readers are advised to interpret all pointer declarations right to left; that is, `a` is a constant pointer to an integer data type. The address contained in `a` is constant or non-modifiable, i.e. the base address of the array is constant. However, unless explicitly required by the caller function to provide modifications of values define in its scope, within the scope of the called function it is advisable to pass arguments by value, which is the principle of least privilege.

```
1. #include<stdio.h>
2. void swap(int *,int *);
3. int main ()
4. {
5. int no1, no2;
6. printf("Enter the two numbers \n");
7. scanf("%d %d",&no1,&no2);
8. swap(&no1,&no2);
9. printf("Numbers after swapping are %d %d \n",no1,no2)
10. return 0;
11. }
12. void swap(int *a, int *b)
13. {
14. int temp;
15. temp =*a;
16. *a=*b;
17. *b=temp;
18. }
```

Figure 4.8: Swap by reference using pointers.

4.11 Constant Pointers

The constant qualifier (keyword `constant`) when applied with normal or non-pointer variables makes them non-modifiable or constant entities. In fact, no longer can they be called as variables! It is better of to refer them as constant identifiers since a variable by its inherent nature should be modifiable. The constant qualifier can as well be used with pointer variables. In fact, constant qualifiers when applied with normal variables or pointer variables to great extent enforce the concept of least privilege, [the minimum amount of right required is granted] and thereby avoid any unintended or accidental side effects. But then the constant qualifiers can be syntactically exploited to create differing pointer types or types of pointer variables or identifiers in terms of privileges to modify. We shall consider the following four pointer declarations for understanding these pointer types:

```
int * a;
const int * a;
int * const a;
const int * const a;
```

The first declaration `int * a` has been explained already. To briefly review `a` is a pointer to some integer data type. The second declaration `int * const a` declares that `a` is a constant pointer to an integer data type. Again we interpret pointer declaration in right to left fashion for better understanding. A is a constant pointer to some integer data type, means that the content in a (which is the address of another integer variable) is non-modifiable, however, the value pointed to by a which is a constant pointer is modifiable or assigned multiple values at different stages of the program execution. But the pointer (address) remains constant or

non-modifiable. That is with the following declaration:

```
int b =8, c= 5;
int * const a = &b;
*a = 6; // allowed
a = &c ; // not allowed
```

a's content or precisely the address cannot be modified but the value pointed to by a can be changed, or in other words, the address is non-modifiable, but the content is modifiable or can be changed. Constant pointers can be assigned the address of another variable, but only once on the basis that constant identifiers [non-pointer or pointer variables] can be initialized [assigned value] once at the point of declaration or once after the declaration statement. Further, the identifiers cannot be modified. For the third declaration `const int * a`, consider the following example:

```
int b=5,c=6;
a=&b; //allowed
*a=7; //not allowed
```

The statement is interpreted as a being a pointer to an integer constant data type or a points to some integer data type that is a constant. The value which a is pointing to is constant or non-modifiable or the content is constant, while the address is modifiable as shown in the above example. In other words, address of the pointer is modifiable, but the content of the pointer remains constant.

The fourth declaration `const int * const a;` is a constant pointer to a constant integer data type. As is obvious, both the address and content of the pointer variable remains constant. Maximum privilege is provided with the first declaration while minimum privilege is provided with the last declaration. Programmers need to judiciously choose the appropriate declarations for identifiers, arguments [identifiers that are passed to functions].

4.12 Variable Declarations and Their Associated Sizes Occupied

A program can declare variables of integer or character or float or double data type. A variable as per our earlier discussions has a name and a value associated with it, in fact, these are the basic attributes of variables. The memory requirement in terms of storage size is on the data type that is being stored in the variable. An integer data type on modern machines consumes 4 bytes (long) of memory. However, earlier systems had the demarcation between a short and long integers.

Short integers are 2 bytes long and long integers are 4 bytes long. Mostly integer data types are used with a singed interpretation. Thus, an integer array of size 10 would actually consume 40 bytes of memory. The size of operator available with C/C++ can be used to determine the specific system byte consumption for the respective data types. The size of operator when applied with the array names reflects the memory requirement for the entire array, while with pointer variables it reflects the byte consumption of the values pointed to or referred to by the pointer variable. Also, the number of elements in an array can be determined using two size

of operations as follows:

$$\text{Number of elements} = \frac{\text{Size of array}}{\text{Size of array data types}}$$

For example, for the array declaration we had earlier the number of elements = 80/4 = 20 [assuming it is an integer array consuming 80 bytes]. However, the byte consumption for a specific data type may vary across systems. Programs that shall be data type-specific and in turn be system-dependent should explicitly make use of size of operator to check for the system settings. It is just good and efficient coding practice. Size of is an operator and not a function. Its computation is at compile time and there is no overhead resulting from it [in terms of execution time]. In fact, it is included in the keyword list of C/C++. As we have stated an operator `size of` can be applied to any variable name or type name or constant value. The `size of` operator requires parentheses when used with type name [`integer` or `char` or `float`] but does not require parentheses when used with other variable names.

4.13 Pointer Expressions or Arithmetic

Pointer variables as normal non-pointer variables can be used in arithmetic expressions or assignment statements. But then not all operators can be applied with pointers. The number of operators that can be applied with pointers is quite limited. Pointers may be incremented (`++`) or decremented (`-`). Additive arithmetic of pointers with integers as well as with another pointer is allowed. For better understanding let us consider the following example. Pointer arithmetic is entirely machine-dependent as the byte consumption for data types differ across machines, `int a[5]`, (assume location 3000H for `a[0]`), `int *vptr = &a[0]`. In normal arithmetic addition of 3 to 3000 yields 3003, but this will not be the case with the pointer arithmetic.

When an integer is added to or subtracted from a pointer, the pointer gets incremented or decremented by that many times the size of the data type to which the pointer refers. The number of bytes incremented or decremented depends on the data type to which the pointer is actually pointing to. Let us assume that the pointers address us currently 3000. Addition of 1 to it is actually referring to the next valid memory address, (since 0–3 totals to 4 bytes]. But then 3000 which is pointing to an integer would occupy 4 bytes. Thus, 3000–3003 is actually consumed and thus the next available location is 3004 which would be the result of 3000+1, the pointer is incremented to make it point to the next location (not necessarily immediate!).

`vptr= vptr+2` would yield `3000+2*4 = 3008 H`. Thus, `vptr` now points to `v[2]`. In general, the above statement for some generic data type would increment the pointer by twice the number of bytes it takes to store a variable of that generic data type. If a pointer is to be incremented or decremented, then the operators `++,--` can be made use of as follows:

`++vptr, vptr++;--vptr and vptr - -`

Each of the above statements either increments or decrements, the pointer points to the next or previous array location. Pointer variables pointing to the same array may be subtracted from one another. For example, if `vptr` contains the location `3000` and `v2ptr` contains the address `3008`, the statement `x = v2ptr-vptr` assigns the number of array elements from `3000` to `3008` which equals to 2. Pointer arithmetic is meaningful only when performed on

an array. One cannot assume that two variables of the same type are stored contiguously in memory unless they are adjacent array elements.

Pointer arithmetic on pointer variables that does not refer to an array of values is a logical error. Again subtracting or comparing two pointers that does not refer to elements of the same array is also a logical error. As has been stated, running or walking off either ends of an array during pointer arithmetic is also logical error. As far as assignment statements are concerned a pointer can be assigned to another pointer if both pointers are of the same type. If two pointers that are not of the same data type are to be assigned, cast operator can be used to convert the right-hand side pointer (pointer on the right of the = statement) to type of the pointer on the LHS of the = sign so that the two pointers are of the same type and the assignment can be performed.

However, a `void *` (pointer to void) or a generic pointer which can point to any data type can be assigned to all pointer types without any explicit casting. That is `void * = int * / char *` is allowed, whereas `int * /char * = void *` is not allowed. A void pointer (*) cannot be dereferenced since the compiler to dereference a pointer needs to know the number of bytes associated with a data type and with respect to a generic pointer `(void *)` which contains a memory location for an unknown data type and hence the number of bytes to which the pointer refers to is unknown for the compiler. For pointer variables dereferencing to be possible, the compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer.

In the case of a `void *`, the number of bytes cannot be determined from the type. Assigning a pointer of one type to another pointer other than the `void *` is a syntax error and as has been stated dereferencing a `void *` is also a syntax error. We have already stated that pointers can be compared with one another using equality and relational operators, but then such comparison should be made on pointers that point to members of the same array. Such comparisons actually compare the addresses stored in the respective pointers and can be used to determine whether a pointer is 0 or not. Arrays and pointers are closely related with each other and can be used interchangeably. Array names are viewed as constant pointers.

Pointers can be used for array subscripting. Let us assume the following declaration:

```
int a[5],*aptr; aptr
```

can be made to point the first element of the array `a` by `aptr = a;` since the array name without a subscript operator is actually a pointer to the first array element which is equivalent to `aptr =&a[0]`. Array element `a[5]` can be alternatively referenced as `*(aptr+5)`. `5` is actually the displacement or offset to the pointer. A pointer that points to the array beginning, the offset is identical to the array subscript and indicates the element of the array that is to be referenced. This notation is also called pointer or offset notation.

The parentheses are mandatory since * carries higher precedence over +. If the parentheses is not used then the expression is parsed as `*aptr + 5` which would compute the sum of 5 and dereferenced value of `aptr [i.e.a[0]`. Therefore, `*aptr + 5` is actually equal to `a[0] + 5`. The statement `&a[3]` is the same as `bptr+3` [pointer arithmetic] and also `a[3]` is equal to `*(a+3)`. Thus, all array subscriptions can be performed alternatively using a pointer and an offset. Similarly, pointers can also be subscripted as arrays. For example, `aptr[1]` actually refers to the element `a[1]`. This is referred to as pointer or subscript notation. An expression such as `a = a+ 5;` is invalid because array names are not just pointers but constant pointers and thus address modification is not possible. Even though pointer arithmetic-wise it is valid,

thus array names cannot be modified in arithmetic expressions. A programming example to understand array subscripting, pointer/offset with array name as pointer, pointer subscripting and pointer offset with pointer to access the elements of an integer array a is explained using the code in Figure 4.9.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  int a [ ] = {5,10,15,20,25 };
5.  int i, offset;
6.  int * aptr = a; //Array subscript notation
7.  for(i=0;i<5;i++)
8.  printf("%d",a[i]); //Pointer offset notation with array name as pointer
9.  for(offset = 0;offset< 5;offset ++)
10. printf("%d",*(a +offset)); //Pointer subscript notation
11. for(i=0;i<5;i++)
12. printf("%d",aptr[i]); //pointer offset notation with pointer
13. for(offset=0;offset;5;offset++)
14. printf("%d",*(aptr + offset ));
15. return 0;
16. }
```

Figure 4.9: Array subscription using pointers.

Let us now look at another example regarding pointer–array interchangeable use, that performs string copy operation. The code is as shown in Figure 4.10.

The above example aims at developing the code to perform string copying operation. There are two functions `copy1` and `copy2` to perform the same operation of copying strings. `Copy1` manipulates the strings with array notation while `copy2` manipulates strings with the pointer notation. The source strings passed to `copy1` and `copy2` are made as constant strings since it is only read operation that shall be performed over the strings. The direction of the copy operation in the function call would be in the following order of (destination, source). The driver function creates four arrays of characters. `Str2` and `Str4` are initialized with the strings "NIT" and "Trichy". Aim of the above program is to copy the contents of `str2` to `str1` and `str3` to `str4`.

The function `copy1` manipulates the strings with array notation. The for loop within the definition iterates thru the string viewing it as an array of characters. Each character of the source string is assigned to the destination string after checking for if it is a '\0' [null or string terminator.] The loop iterates till the end of source string ['\0'] is encountered. Observe that the copying operation has been achieved as a part of the for header structure itself. Thus the definition of the `for loop` is empty. Similarly the function `copy2` copies the strings manipulating them with pointer notation. Again the definition of the for is an empty statement as the copying operation is accomplished in the for header structured itself by the statement `*s1=*s2`. The loop iterates until the de referenced character from `s2` is not a null character. The test driver function main creates two strings NIT and Trichy and two more empty strings. The functions `copy1` and `copy2` are invoked to test the validity of the copying operation.

```
 1. #include<stdio.h>
 2. void copy1(char *, const char *);
 3. void copy2(char *, const char *);
 4. int main()
 5. {
 6. char str1[10],*str2="NIT";
 7. char str3[10],*str4[]="Trichy";
 8. copy1(str1,str2);
 9. printf("Copied string in str1 is %s",str1);
10. copy2(str3,str4);
11. printf("Copied string is str3 is %s",str3);
12. return 0;
13. }
14. void copy1(char *s1,const char *s2)
15. {
16. int i=0;
17. for(i=0;((s1[i]=s2[i])'='\0');i++)
18. ;
19. }
20. void copy2(char *s1,const char *s2)
21. {
22. for(;((*s1=*s2)'='\0';s1++;s2++)
23. ;
24. }
```

Figure 4.10: String copy using pointers.

4.14 Arrays of Pointers

Arrays can also contain pointers in them. One such example is string array and as we have already seen a string by itself is an array of character. Thus, each string is actually a pointer to the first character. Thus, a string array or an array of string is actually an array of pointers, each pointer referring to the first character of each string. An array of four student names would be declared as follows:

```
char Name_Array[4][50]={"Sudarshan.G", "SriKrishnan", "SaiSiddarth", "Siddarth.J"}
```

`const char * Name_Array[4]` is another equivalent syntax for creating an array of four names. Each name is a pointer to `char` [constant] or each name is by itself an array of characters [array of constant characters]. Although it appears as these names are placed in the array `name_array`, it is only pointers that are actually stored in the `name_array` and each pointer points to the first character of the corresponding string or name. The equivalent syntax of above can as well be used with the first subscript (rows) for the number of strings or names and the second subscript [columns] for the largest string. Both rows and columns being fixed is one disadvantage of the above declaration(1), that is the column size being set to the size of the largest string, considerable memory may be wasted when a large number of names is

stored and each name is shorter than the longest size string declared. Hence, the declaration 2 is a more efficient form of declaring string arrays.

4.15 Function Pointers

In the previous section, we have dealt with pointers to data members. As we declared data pointers, we can also have pointers to functions or what is called as function pointers. Function pointers contain pointers to functions or addresses of functions in memory. As an array name is actually the address in memory of the first array element, a function name is similarly the starting address in memory of the code that performs the function's task. As arrays or pointers were passed to functions, so function pointers can be passed to functions, returned from functions and assigned to other function pointers.

For a better understanding of the function pointers, we shall develop a program to sort an array of numbers either in descending or ascending order. In fact, an earlier code performed the same tasks, but in a redundant fashion. Whether it is ascending or descending order sorting, the sorting algorithm [assuming it is bubble sort] remains the same. Bubble sort is based on the overall principle of comparing adjacent numbers and swapping them if they are not in the right or expected ordering required. Instead of duplicating the bubble sort logic twice [once each for ascending and descending], a solution that implements bubble sort logic only once and takes care of swapping suitably for ascending or descending order would be a better choice than have the bubble sort logic repeated twice just because the swapping condition is different in each case.

In this program, we shall define a function called `bubble_sort` which in addition to the normal arguments of the array to be sorted, array size also accepts a pointer to either of the two functions ascending or descending. Each of these functions return a value 1 (true) if swapping needs to be performed and a 0 (false) if swapping need not be performed. For a better interpretation of the above function declarations, an ascending function returning a value indicates that the adjacent numbers to be compared are not in the required ascending order and further swapping should be done to ensure that they are in the correct ascending or strictly increasing order.

A 1/0 return has been adopted to indicate that 1 represents the TRUE condition of ascending ordering or increasing ordering required for the current pair of numbers being compared and 0 represents a FALSE condition of no ordering required for the current pair of numbers[they are already in the order required]. In other words, the TRUE condition represents the fact that the adjacent numbers or current pair of numbers being compared or checked are not in the required ordering and hence swapping is required. The FALSE condition represents that the numbers are as per the ordering required and hence no swapping is required for the current pair of numbers being processed which s based on bubble sorting logic.

A true or false value indicates whether swapping is required or not for the current ordering desired. Precisely, the task of checking whether the current or adjacent pair of numbers being processed by the bubble sorting logic are already in ascending or descending order or not is taken care of by two functions, namely ascending and descending. If the numbers are not in the required order the functions return a 1 to and then swap the values invoking another helper function called exchange that swaps the values passed to it and this swapping should get reflected in the caller function. Hence it accepts the arguments by reference or pass by

reference is used. To briefly suggest a solution, the various functions and their corresponding declarations are as follows:

```
int ascending(int,int);
int descending(int,int);
void exchange(int *const, int * const);
void bubble (int *,const int, int (*) (int,int));
```

The first-two function headers have been explained previously. The functions accept only two integers because at all stages of the ordering task, the bubble sort logic would process only two, in fact, two immediate or adjacent numbers. The function exchange which does the swapping of values by reference needs to modify only the value or content at an address and not the address where these values reside. That is to say address modifications should be prevented. This is taken care of by declaring the arguments to be of type `int * const` or constant pointers to integers in which case it is only the pointer actual content or value and not the address to which it points to that can be changed. This enforces the principle of least privilege which is so very crucial to good OO software development as we shall see in the later OOP chapters.

The last and as often they say but not the least which is so very true here, the bubble sort function is where the bubble sort sorting logic is implemented. This function accepts the array to be sorted, array size and pointer to either of the two functions ascending or descending. These functions when defined will have separate memory allocations for their definitions and the function name actually refers to the starting address in memory of the above function code. This is now passed as an argument to function `bubble_sort`, which means that the ascending or descending function would get invoked within the definition of function bubble sort to decide on the swapping requirement. In fact, the very purpose of having arguments passed and a formal name for them is to refer them within the function definition.

In this case, it is a pointer to another function that is passed and referring to this pointer within the function definition would actually be invoking the respective function definition whose address is available in the function pointer or to which it is actually pointing to. We have almost gone through the entire program logic with the exception of the syntax of defining or declaring a function pointer. The statement `int (*fname) (int a,int b)` identifies to the compiler that the function fname is actually a pointer to some function which accepts two integer arguments and return s an integer at the point of invocation. Let us know look at the code that performs the above tasks which is shown in Figure 4.11.

In the above code, the declared function pointer gets its address required (address where another function definition resides in memory) only in the statements in lines 14 or 20 when the decision for ascending or descending ordering requirement is made. The function pointer which was only declared as a part of the `bubble_sort` header gets a name identifier for it in the function header of `bubble_sort` function definition [compares the function to which it would refer to, actually compares the two numbers passed]. It is a formal parameter for reference within the function definition and gets related to the function's address to which it should refer or point to based on users ordering choice of ascending or descending. The logic for bubble sort is implemented only once and the comparison task is left to a function that accepts a function pointer which actually does the comparing logic.

The main application of function pointers is in building menu-driven systems. Each option is serviced by a different function. Pointers to each function are stored in array of function

```
#include<stdio.h>
void bubblesort(int [],const int,int (*) (int,int));
int ascending(int,int);
int descending(int,int);
void exchange (int * const, int * const);
int main()
{
const int arraysize=5;
int order required, count, input[arraysize];
printf("\n Ascending(input 1) or Descending Order(input 2) Required \n");
scanf("%d",&order required);
if(order required ==1)
{
bubblesort(input,arraysize,ascending);
printf("The Ascending Ordering of the input array is \n");
}
else
if (ordering required ==2)
{
bubblesort(input,arraysize,descending);
printf("The Descending Ordering of the input array is \n");
}
for(count=0;count<arraysize;count++)
printf("%d \t",input[count]);
return 0;
}
void bubblesort(int pro arr[], const int size, int (*comp) (int,int))
{
for (pass=0;pass<size-1;pass++)
for (int j=0;j<size-1;j++)
if((*comp)(pro arr[j],pro arr[j+1]))
exchange(&pro arr[j],&pro arr[j+1]);
}
int ascending(int a,int b)
{
if (b<a)
return 1;
else
return 0;
}
int descending(int a, int b)
{
if (b>a)
return 1;
else
return 0;
}
void exchange(int *const ptr1, int * const ptr2)
{
int temp;
temp = *ptr1;
ptr1 = * ptr2;
ptr2 = temp;
}
```

Figure 4.11: Bubble sort using function pointers.

pointers. The user's choice is used as a subscript into the array and the pointer in the array is used to call the function. We shall look at an example to understand this application of function pointers, code for which is shown in Figure 4.12. The number of options available for the end user is three.

```
1. #include<stdio.h>
2. void fn1(int);
3. void fn2(int);
4. void fn3(int);
5. int main ()
6. {
7. void (*f[3])(int) = fn1,fn2,fn3;
8. int choice;
9. printf("Enter Choice 1 -fn1, 2- fn2, 3- fn3 \n");
10. scanf("%d",&choice);
11. while (choice >= 0 && choice < 3)
12. {
13. (*f[choice]) (choice);
14. printf("Enter Choice 1 -fn1, 2- fn2 ,3- fn3 \n");
15. scanf("%d",&choice);
16. }
17. return 0;
18. }
19. void fn1(int a)
20. {
21. printf("Function fn1 called with value %d",a);
22. }
23. void fn2(int a)
24. {
25. printf("Function fn2 called with value %d",a);
26. }
27. void fn3(int a)
28. {
29. printf("Function fn3 called with value %d",a);
30. }
```

Figure 4.12: Function pointer's application in menu-driven systems.

The declaration in is read as f is an array of 3 function pointers that each take an integer as an argument and return void. The array is initialized with the three functions that are to associate with the user choice. The user's choice is used to subscript into the array of function pointers. The function call is made as follows:

```
(*f[choice])(choice)
```

in which f[choice] selects the pointer at location choice in the array. The pointer is dereferenced to call the function and choice is passed as the argument to the function. For the sake of program flow understanding, each function pointer prints its argument values and its function name to indicate that the intended function was called correctly.

4.16 Character and String Processing

Characters are the fundamental blocks of a C or C++ program. Every program is composed of a sequence of characters that when grouped together meaningfully forms a string. A string is a series of characters treated as a single unit. A string can contain letters or digits or special characters. String literals or string constants are written in double quotation as

```
char * str = "constant"
```

Strings in C/C++ which are treated as array of characters end with a null character or a string terminator ['\0']. It is the string terminator that identifies that the end of the string is reached with the current character array that is being processed. A string may be assigned in a declaration statement to a character array or a variable of type char *. The following declarations are allowed:

```
char exarray[] = "Sivaselvan";
const char * exarray="Sivaselvan";
```

The first declaration creates a 10 element character array containing the individual characters and the last character stored is '\0'. Array indexes ex-array[0–9] contains the individual characters. The compiler reserves an additional index for the null character or a string terminator ['\0']. This, is an additional index allocated and does not consume the allocations for the character array ex-array. Thus for all string or character arrays the compiler allocates one index in addition to whatever is required by the programmer and it is the index or location that is used to store the strings termination character. When storing a string of characters in a character array, the array should be large enough to hold the largest string that can be stored. A string that is longer than the character array in which it is to be stored, characters beyond the array end overwrite or overflow data in memory locations following the array.

The string handling library makes available functions for string manipulations such as comparison of strings, copying strings, string searching for specific characters, string tokenization [splitting] and length of the string determination. The list of all the functions is made available in the string.h header file that should be included in the program for the string manipulation functions mentioned below to be made use of or called are as shown in Table 4.6. The code for string manipulations is as shown in Figure 4.13.

Table 4.6: String manipulation functions

Function prototype	*Description*
`char * strcpy(char *s1,const char * s2)`	Copies the string `s2` into the character array `s1`. The value of `s1` is returned.
`char * strcat(char * s1,const char *s2)`	Appends string `s2` to `s1`. The first character of `s2` overwrites the '`\0`' of `s1`. Finally, the value of `s1` is returned.
`char * strncpy(char *s1,const *s2,int n)`	Copies at most `n` characters of string `s2` into character array `s1`. Value of `s1` is returned.
`char * strncat(char *s1,const *s2,int n)`	Appends at most `n` characters of string `s2` to `s1`. The first character of `s2` overwrites the '`\0`' of `s1`. Finally, the value of `s1` is returned.
`int strcmp(const char*s1,const char *s2)`	Compares the string `s1` with string `s2`. The function returns a value of zero, if the string are equal, `<0` if string 1 is less than string 2 , and `>0` if `string1` is greater than `string2` respectively.
`int strncmp(const char *s1,const char*s2, int n)`	Similar to the above comparison function, except that it is only the `n` characters that are compared for equality.
`int strlen(const char *s)`	Excluding the '`\0`', it returns the number of characters preceding the `\0`.

Function `strcpy` copies its second argument a string into the first argument, a character array that should be large enough to hold the string being copied. The '`\0`' is also copied. Function `strncpy` is similar to `strcpy` except that the number of characters to be copied from `s2` is also specified in the call. The terminating null character is not copied unless the length of the string or the number of characters to be copied from `s2` is at least 1, greater than the length of the string. If the size is less than that of the string length, then the terminating character is not copied. Only if this size is greater than the number of characters to be copied, that many additional '`\0`'s are appended which means that in the earlier case a `\0` should be physically inserted in the last position, otherwise, the program gets into serious logical errors.

```
#include<stdio.h>
#include<string.h>
int main()
{
char str1[]="National",str2[10],str3[15],
str4[]="Institute",str5[]="Inst";
char input[]="This is to test strtok", *sep;
int choice,i=0;
printf("\n String Manipulations...");
printf("\n 1. strcpy (string copy)");
printf("\n 2. strncpy (string copy))";
printf("\n 3. strcat (concatenation)");
printf("\n 4. strncat (concatenation)");
printf("\n 5. strcmp (comparison)");
printf("\n 6. strncmp (comparison");
printf("\n 7. strtok (separation)");
printf("\n Enter Your Choice :");
scanf("%d",&choice);
switch(choice)
{
case 1: strcpy(str2,str1);
printf("\n Copied String is %s",str2);
break;
case 2: strncpy(str3,str1,6);
str3[6]='\0';
printf("\n Copied String is %s",str3);
break;
case 3: strcat(str1,str4);
printf("\n Concatenated String is %s",str1);
break;
case 4: strncat(str1,str4,5);
printf("\n Concatenated String is %s",str1);
break;
case 5: if(strcmp(str4,str5)==0)
printf("\n Strings are equal");
else
printf("\n Strings are not equal");
break;
case 6: if(strncmp(str4,str5,4)==0)
printf("\n Strings are equal");
else
printf("\n Strings are not equal");
break;
case 7: sep=strtok(input," ");
while(sep[i]!='\0')
{
printf("%c",sep[i]);
i++;
}
break;
default:break;
}
return 0;
}
```

Figure 4.13: String manipulations using built-in functions shown in Table 4.6.

Review Questions

1. Justify the need for arrays construct in any programming situation.
2. Create an integer array and initialize it with values of 0,2,4,6 and 8 using a single assignment statement.
3. Repeat the earlier question, this time accepting the values from the keyboard.
4. Develop a C program to perform linear search over an input array. (Search if a user-specified element is present in the input array or not. If element found returns its position, otherwise, display the message element not found.)
5. Develop a C program to sort an array of numbers using selection sort.
6. Develop a C program to sort an array of numbers using insertion sort.
7. Develop a C program to implement binary search over an input array.
8. Develop a C program to generate the mean, median and mode of elements in an array.
9. Develop a C program using pointers to perform string manipulations of string copy, concatenation and comparison.
10. Justify the need for function pointers.
11. Develop a C program using functions to generate the reverse of a string. Use the function to test if an input string is a palindrome or not. A palindrome is a string that reads the same in the forward and the reverse direction.
12. What do you understand by the term time complexity of an algorithm?
13. Derive the time complexity of binary search algorithm (refer external sources as well).
14. Derive the time complexity of insertion sort algorithm (refer Thomas H Cormen, *Introduction to Algorithms*).
15. Differentiate the declarations `int * a` and `const int * a`.
16. Discuss the unique aspect in C++ with respect to passing arguments to functions which is not supported in traditional C.
17. Why is not mandatory to pass arrays as references?
18. Justify the argument behind the `&` operator not being mandatory when reading in strings.
19. Identify the different programming paradigms and compare them.(refer external sources).
20. Argue the case for object-oriented programming over structured programming.

CHAPTER 5

Object-Oriented Programming C++

Well, in the earlier chapters we have concentrated on the various key features of the structured programming that will be made use of in the object-oriented programming. In fact, a few concepts explained earlier are advanced structured programming features available with C++ and not with C. As stated already, these concepts of structured programming are so very crucial to object-oriented programming. Objects will be constructed by exploiting the various structured programming concepts. From this chapter onwards, we shall explore the important concepts of object-oriented programming, namely Data Encapsulation, Data Abstraction, Operator Overloading, Polymorphism and Inheritance, and Class Templates. In this chapter, we shall details on provide data encapsulation and data abstraction concepts.

5.1 Data Encapsulation and Abstraction

As a part of structured programming, we have seen that it is variable declarations [data] that uniquely identify entities in a program. Structured programming as a result of modularization had several functions performing different subtasks in an integrated fashion under the control of `main()` by manipulating the various variables or identifiers or constants, or data declared in the main in other functions. But then all along in structured programming data and functions which operate on these data were viewed as separate entities and more importance was given to functions.

One key software engineering observation is that data is not given the importance that it is due for. It is in fact these data that distinguish or identify entities. And also in structured programming there is no way of associating functions with specific data. Yes, structured programming does support type checking enabled function calls, but this was only to ensure that the function gets called with the correct number of arguments and the correct data type for each argument. So, this only associates functions with their data types and not with specific data.

5.1.1 Data Encapsulation

Object-oriented programming differs from structured programming in this approach. OOP encapsulates data or what can be called as attributes and functions or behaviours into a single unit or entity called a class. Classes can be viewed as packages. The data and functions of a

class are associated or fixed together. It is this association of data and functions together that is so very crucial to object-oriented programming. This concept of packaging or merging data and functions into a single unit is referred to as Data Encapsulation.

The naming is based on the reasoning that among this set of Data and Functions it is data that is more important and encapsulation refers to the concept of merging data and functions together or encapsulating data and functions within a single entity called a Class. Encapsulation is the process of placing multiple items with a single large entity. The multiple items are data and functions and the single large entity is the class. To correlate with set theory, a class is like a set and data, and functions that form the class are elements or members of the set class. It is based on this reasoning that data and functions of a class are often referred to as data members and member functions. [They are members of the class.]

For a better understanding of these concepts, let us consider a day to day example. A class can be treated as the plan of a building. From a single plan multiple buildings can be created. From a class a programmer can create objects. As a part of this example, we have already said that multiple buildings can be created from a single plan which means that the same plan can be reused as many times the end user wishes to. Similarly, a single class can be reused to create multiple objects. End users when distributed with a class can reuse the class with or without modifications to the class and create multiple objects. This is nothing but the reusability concept of object-oriented programming.

5.1.2 Data Abstraction

Another key property of classes is the concept of Information Hiding, also referred to as data abstraction. An object-oriented program would involved many objects to be manipulated, which means there should be interaction or communication between these objects to achieve a solution for the initial problem definition. Thus, although intercommunication between objects is required [supported via interfaces], a class is not allowed to know the details [data or functions of another class].

In other words, details of implementation of a class are hidden from other classes. It is the hiding of implementation details of a class from other classes that promotes data security to a great extent. Data abstraction is that key feature of OOP wherein end users are provided only with interfaces for manipulation of objects and not the implementation details of the class.

5.2 C++ Characteristics

The programming language we will explore in this and the following chapters is C++, an object-oriented language. With structured programming language the programmers approach to code development is action or function oriented, whereas with C++ programming is object-oriented, with more emphasis or importance to data in relation to functions. Literally, on comparison basis C++ can be called data oriented while C or other structured languages called function oriented. Both types of languages data and functions are important but then it is the relative dominance of one over the other that decides the nature of the programming languages.

The basic unit of programming or code development in structured languages is functions while in OO languages it is objects that constitute the basic unit. Objects are instantiated

or created from classes. Thus, programming in an action or structure-oriented approach, programmers develop functions. Group of actions performing some task constitute functions. A program may involve many such functions and it is these functions that interact with each other to make up the program [overall solution to the initial problem definition]. As we have already seen that functions are given primary importance in structured languages and data exists with secondary importance to help functions perform their task.

One way of identifying functions given in a problem statement or system requirements specification document, assuming that English is the language of choice to express such specification documents, the VERBS (act of doing something or actions(s)) are indicatory of the possible functions of a solution for the problem on hand can have. C++ programmers develop classes. From the above interpretation, classes are also referred to as user- or programmer-defined data types. These classes will be used to create objects or instances of it. In fact, the structured programming syntax of `int a;` to declare `a`, variable of integer data type, `int` could be viewed as a class and `a` as an instance or object of class of `int`. The only difference is that objects will be created from programmer-defined data types rather than from predefined data types. Classes can be identified from a system specification by looking out for NOUNS [identifying entities].

5.2.1 Notion of a Class

In fact, the concept of classes in C++ is an extension of the structure concept in C. C structures which allow multiple predefined data types to be combined to form a new user-defined data type, but again functions never got associated with specific data types. A C++ structure does allow functions and data to be encapsulated. In this aspect, they are quite similar to classes, but they do differ and this shall be explored at a later stage. Classes enable the programmer to model objects or real world entities that have attributes (data members) and behaviours (functions). Such data types are created using the keyword class.

5.2.2 Member Functions

Member functions are also referred to as methods in some OO languages like JAVA. These member functions or methods are called responses to messages sent to an object. Once a class has been defined, it can be used to create or declare objects of it. We will develop a class called `EXAMPLE` that contains an integer data member and functions `initialize` (to initialize the object) and `increment` (to perform increment operation on the data members). The code for the above operation is as follows:

```
class EXAMPLE
{
private :  int data;
public:  void initialize (int val);
void increment ();
};
```

The name of our class is `EXAMPLE`. The class definition begins with the left brace and the definition ends with the close brace and a semicolon. Our example class contains a single data member (`int` data) purely for the purpose of differentiating objects. The class definition also

contains the prototypes for functions `initialize()` and `increment()`. The task of member function `initialize` is to initialize the data member (data) and member function `increment` increments the current value of data by 1.

5.2.3 Access Specifiers

The two keywords of private and public are referred to as access specifiers and are applied with data members and member functions of a class. Access specifiers are syntactic provisions to implement the data abstraction feature of OO languages. In fact, there are three access specifiers, namely private, public and protected. The protected access specifier will be explained in detail when we will discuss on Inheritance.

Access specifiers are nothing but the visibility or accessibility mode for the data members or member functions of the class. As a part of our earlier discussion, we have mentioned that the implementation details of the class must be hidden for the external world. The private access specifier makes the data members or member functions with which it is being applied accessible or visible from only within the class definition or within the {} of the class. These members are not accessible from outside the class definition.

In fact, the member accessibility operator is the dot operator and → for pointer members. That is to access a normal data member (say `int x`) of a class A whose object is `o1`, the syntax of `o1.x` is adopted. Member functions are as well accessed with the same syntax. Again if there is a member function `fn1()` in a class A, with private access specifier, it remains accessible from only within the class. Private members of a class are not accessible from outside the class, even via the objects.

Public members of a class are accessible from outside the class. Access specifiers are required for each and every data member or member function of the class. But when data members or member functions have similar access specifiers, all of them can be grouped together and the access specifier keyword be mentioned only once at the beginning. The access specifier keyword must be followed by a colon symbol.

Access specifiers need to get mentioned with each data member or member function only if its access specifiers differ from the preceding access specifier. That is, let us assume there are three data members and two member functions of a class Test, of which the data members must carry the private access specifier with them and the member functions should be public. This is done by the following declaration. Thus, members of a class with public access specifiers are accessible from outside the class.

The normal convention is to make data members of a class private and member functions of a class public. The reasoning behind this convention is that data members of a class being private, they will not be accidentally modified from outside the class. Such private data members of a class can be accessed either from within the class or from outside indirectly via the public member functions of the class. It is based on this that member functions which will be accessed from outside the class definition via the objects (and then indirectly access private data members) are declared as public member functions which will be accessed from only within the class or from outside the class within other public member functions of the class can be made as private. Such private member functions are also referred to as helper or utility functions in the fact that they help other public member functions of the class to achieve their task.

Member functions of a class can be defined within the class body itself in which case function prototypes within the class will be omitted. Member functions can also be defined outside the class definition, in which case the prototypes or declarations of such member functions will be provided within the class body or definition. Member functions defined in this manner outside the class body or definition still belongs to the scope of the class. The binary scope resolution operator (:) is used to resolve the scope of the member functions defined outside the class with the class.

5.2.4 Scope Resolution Operator

The scope resolution operator preceded by the class name is prefixed before the function header. The return type of the function precedes the entire header specification. As a part of the following example, we shall see the syntax of defining member functions. Member functions being defined outside the class definition with their scopes resolved using the :: operator apart from offering better code readability or clarity differ from the other syntax of defining member functions within the class definition in that such member functions define in that, such member functions defined within the class definition are automatically in lined. Whereas member functions defined outside the class with their scopes resolved using the binary scope resolution operator are not automatically in lined. Such member functions can be explicitly in lined by prepending the keyword inline before the respective function header. Let us now get back to the earlier class `EXAMPLE`. The class definition is as follows:

```
Class EXAMPLE {
private :  int data
public:  void initialize (int val);
void increment();
};
```

Note that we have mentioned the public access specifier only once. All member declarations or definitions follow. As we have already stated, an access specifier holds for the future declarations unless it is overridden by a new type of access specifier. If members of a class are to carry alternating or changing access specifiers, then the access specifier should be mentioned with each member declaration statement. Note that for function definitions within the class body access specifier is required to be mentioned once with the respective function header.

In our class `EXAMPLE`, we have two member functions `initialize()` to initialize the data member (data) of the `EXAMPLE` class, and function `increment()` to increment the data member. Private data members of a class cannot be initialized directly at the point of declaration with an assignments statement. This will cause syntax errors. Data members of the class can be initialized by public member functions of the class or by using a special function called constructor which shall be explained in detail in the next section. For the moment we shall use the function `initialize` to initialize the data members of the class to a valid value. The entire code for the above example is as shown in Figure 5.1.

5.2.5 Input/Output Statements in C++

Note that the statement in line is the display statement in C++. The statement `cout << "Invalid Initializer"<< endl` redirects the string `Invalid Initializer` onto the screen

```
1.  #include<iostream.h>
2.  class EXAMPLE
3.  {
4.  private: int data;
5.  public: void initialize (int val);
6.  void increment();
7.  };
8.  void EXAMPLE :: initialize (int val)
9.  {
10. if (val>=0)
11. data = val;
12. else
13. cout<<"Invalid Initializer value"<<endl;
14. }
15. void EXAMPLE:: increment ()
16. {
17. data = data + 1;
18. cout<<"Incremented Data Member is"<<data<< endl;
19. }
20. int main ()
21. {
22. EXAMPLE O1;
23. O1.initliaze();
24. O1.incrrement ();
25. return 0;
26. }
```

Figure 5.1: C++ programming example.

and is the display statement in C++, equivalent to the printf function in C. The operator << is referred to as the stream insertion operator. When this statement is executed, the value to the right of the operator (in this case the string " ") or the right operand is inserted in the output stream. Normally the right operand is printed exactly as they appear between the double quotes.

However, the characters \n,\a are not printed on the screen. The backslash, escape sequence is a special character that achieves predefined I/O tasks (explained in chapter). The endl string is also called the stream manipulator. It outputs a newline on the screen, in this aspect it is similar to the \n escape sequence. It differs from the \n in that the endl manipulator after redirecting a newline on the screen flushes or clears the output buffer.

This simply means that on some systems where outputs accumulate in the machine in a buffer until there are enough to redirect it to the screen. Thus, the endl flushes the output buffer or redirects the entire string or output sequence that is accumulated. Getting back to the code, it is again within the function main where normally the classes are instantiated, and on this basis main is also referred to as the test deriver function—function that test derives the class. The statement creates an object O1 of type class EXAMPLE.

The next two statements at lines 23 & 24 invoke the functions initialize and increment of class EXAMPLE with respect to the object O1 created in main. Note that the cout statement for display purposes in C++ can redirect a string or the value appearing after the insertion operator << onto the screen. Cout does not require the data types of values to be displayed as was in printf. That is why cout << "Incremented Data Member" << data << endl first displays the string within quotes, then displays the value of data and then prints a newline and flushes the output buffer.

With printf the format specifier of data that is %d for integer should be specified within the function, but there is no such compulsion in C++'s display statement. In fact, cout is not a function as was printf. Cout is an instance or object of the class iostream whose declarations are available in header file ostream.h. In fact, the other complementing input operation is performed in c++ with a cin >> data; statement to read the value of a from the keyboard or the standard input device.

Cin is an instance or object of class istream whose declarations are stored in the header file istream.h. The >> operator is called as the stream extraction operator and extracts input from the standard input devices to the input stream .most C++ programs often require both statements and it is common to include the header file iostream.h[input–output stream] which contains the declarations for the class iostream or IOS. These classes are created by combining the features of class ostream and istream or what is called as by inheriting the features of ostream and istream.

In fact to be precise iostream is created by multiple inheritance from classes ostream and istream. Inheritance is another key feature of oopS that enhances the reusability feature of the software. A class that will have features similar to another but that which also has its own features created by inheriting the other class for similar features and having its own explicit features separately created. The class from which features are inherited is referred to as the parent or base class and the class which inherits features is called the derived class. In the IOS case, the derived class is IOS or iostream and there are two parent or base classes, namely ostream and istream.

Inheritance is an important concept, we have provided a separate chapter for its discussion. For the moment, this understanding of Inheritance would suffice. As we have already said cout or cin is not a function. They are objects of the respective I/O class and << or >> are functions. They are functions that have been overloaded using the operator overloading feature of C++. Operator overloading again a key concept of OOPS will be explained in detail in later chapters. For the moment, one should be clear to interpret cin or cout statements as functions getting invoked with respect to objects cin our cout.

The operator <<, >> can also be cascaded in a single statement. That is to say cout << "Hai" << value; is allowed. In this, two redirections, one a string, and the other a value display are cascaded within a single statement. This proceeds well because at the end of each redirection or indirection, the overloaded << or >> function call would return to a reference to the object of the respective ostream or istream class. The remaining or cascaded operators are performed with respect to this returned object reference. The compiler parses the above statement as ((cout. << ("Hai")). << (value)).

At the end of this a reference to stream is returned and the remaining insertion operations are performed with respect to the returned reference. This is referred to as cascading or chaining of insertion operators. Similarly, even the extraction operators can be cascaded to the cout statement with the only difference that the reference returned is of type cin or a

reference to `cin` is returned after one indirection or extraction operation is performed. It is referred to as indirection, as values are fed or given in to the system and flow is from the external device into the system. In fact, << and >> are overloaded operators in C++.

An operator is said to be overloaded if it has more than one meaning or operation associated with it. <<, >> when used with normal integers imply left shift and right shift bitwise operators. Thus, the << and >> operators have two meanings associated with them, namely insertion or extraction and left shift or right shift operators. Such operators that have more than one meaning with them are called overloaded operators and is implemented using operator overloading. Overloading is implemented using functions [member or non-member] with the restriction that the function name would be the operator that is to be overloaded and preceded by the keyword `operator`. We will not get any further into operator overloading now but shall explore it in detail in chapter.

Note: Member functions are normally shorter than stand alone functions and functions of a non-object-oriented language. This is because the data are already available in the object as data members of the class and hence the member function calls often require no arguments or at least fewer arguments than a function call in a structured or no OO language or what are also called stand alone functions.

5.2.6 Class Declaration Syntax

Attempting to initialize a data member in the class definition is a syntax error. Classes can be instantiated as many times the client wants and there by create multiple objects. Memory allocation is done for multiple objects individually only for data members. The member functions of the class (object version) reside at a common memory location so that multiple objects can share this function code. That is memory allocation across multiple objects of the same class is done only for data members, whereas member functions are not duplicated in locations across multiple objects. Also, objects of the class share one common copy of the member functions of the class.

As we have already stated that member access specifiers need to be mentioned only if the current access specifier differs from the earlier specifier. And also members of a class [data or member functions] default to the private access specifier. That is, if the access specifier for a data member or member function is not explicitly specified in the class definition, then the compiler assumes the access specifier for the respective member to be type private. The best programming practice is to list all private members of the class first and then follow it up with protected members if any and then finally list the public members. With this practice one can avoid frequent repetitions of access specifier's keyword. Since an access specifier specified (or defaulted to private) is valid and applicable for all subsequent data members or member functions until it is overridden by another access specifier. For better understanding let us go through the following example:

```
class SAMPLE {
private:int a;
int b;
int c;
int d;
void fn1();
```

```
public:
.
.
.
};
```

5.2.7 Choice of Access Specifiers

Here, the private access specifier for the data member (`int a`) [though not a compulsion to be explicitly mentioned as all members default to the private mode] is applicable for all subsequent members till the prototype for member function `fn1`. Now we have a new access specifier public and this is applicable for all subsequent members until we have another specifier differing from the current one. As we have already stated readers are advised to group all private members [data or member functions] first, the followed by protected members and then finally the public members of the class. There is no requirement that all grouped private or protected public members of the class be placed between a pair of { } braces. The compiler automatically treats members of a class not overridden by a new access specifier differing from the earlier specifier as belonging to the same group.

5.2.8 Possible Coding Errors

Possible syntax errors with class definition is missing out the semicolon after the class definition. Applying size of operator on objects of a class reflects only the size of the data members. Member functions size is not reflected in the size of the object. This was explained earlier. The reasoning behind this is that it is data members that will remain unique and vary across objects. It is these data members that differ across objects and require explicit memory allocations. Hence it is only these data members memory consumption that is reflected in the size of operator application on an object.

5.3 Interfaces and Implementation

Separating interfaces from implementation is one of the fundamental issues of good software development. The public member functions of a class are also referred to as public services or public interfaces of the class. In fact, to be precise interfaces are the prototypes of member functions of a class with the public access specifier. Classes will be distributed to end users or clients. Thus end users or clients should be given only details they actually require and should not be overloaded with unnecessary distributions.

C++ is one language that promotes information hiding. The implementation details of a class are hidden from the end users or clients. Implementation details are nothing but function definitions, to be precise the member function definitions. These functions are not distributed to end users or clients. As was the case with C (wherein only `printf's` declaration is provided in the to be included header files], it is only the prototypes or declarations of the prototypes or declarations of these member functions that are distributed to clients.

5.3.1 Principle of Least Privilege

Prototypes of member functions are also referred to as interfaces or services. It is only these public interfaces that are distributed to end users. Thus, there is a need to separate interface from implementation. In fact, it is only the compiled object versions of an implementation that is distributed to end users. This is again nothing but the principle of least privilege, wherein end users have access to the interfaces and object versions of the implementation and not the source file. But then this is not with the intention of just hiding the source file. This syntax of separating interface from implementation avoids unnecessary recompilation of already compiled code and makes available to end users details that are required.

Thus, the accepted software engineering practice is to place the class declarations in header file to be included by clients who wish to use the class. This is the public interface and provides the clients with function prototypes for it to call the member functions. The member function's definitions are stored in a separate source file (`.cpp`) which forms the class's definitions. The end user or clients statements are normally defined in function `main()` in another source file (`.cpp`). The class declarations are made use of by including the respective user defined header file. Recoding our earlier EXAMPLE class to follow the principle of separating interface from implementation, the code looks as shown in Figure 5.2.

At the time of compilation the clients code in `main.cpp` is compiled and linked along with the member functions object code. Member function's source file is not compiled again unless it changes in definition. Clients interested in creating interfaces need to just compile the main program with the source code file (implementation) that generates respective object versions. Readers are advised to refer the programming language manual on how to compile multiple source files.

Note: Despite the fact that the public and private member access specifiers may be repeated and intermixed, list all the public members of a class first in one group and the list all the private members in another group. This syntax clearly emphasizes the class's interface. Keeping all data members of a class private and having public interfaces to modify these data members hides implementation details from the end users or clients promotes program modifiability and reduces bugs.

Defining member functions inside the class definitions automatically in lines the member functions. This can lead to performance improvements in terms of execution speed, but violates the principle of separating interface from implementation as a result of which if the implementation changes in definition the clients code must be recompiled. When interfaces are clearly separated from implementation and the implementation changes its function definition, provided the interfaces remain constant, the clients code need not be recompiled. It gets linked with the object version of the changed implementation, thus avoiding the unnecessary compilation time overhead.

5.3.2 Class vs. Structure

As a good coding practice only small member functions [in terms of lines of code] and stable member functions [those that are not susceptible to frequent changes] should be defined within the class definition, because change is a rule than a exception. C++ class construct is a natural evolution from the struct construct. But then struct construct is disadvantageous in that invalid values can be assigned to the members of a structure since the program has direct

```
"Example.h"
class Example
{
private : int data;
Public : void initialize(int val=0);
void increment();
};
"example.cpp"
#include "example.h"
void Example::initialize(int ini)
{
void EXAMPLE :: initialize (int val)
{
if (val>=0)
data = val;
else
cout << "Invalid Initializer value"<<endl;
}
void EXAMPLE:: increment ()
{
data = data + 1;
cout << "Incremented Data Member is"<<data<<endl;
}
"main.cpp"
int main ()
{
EXAMPLE O1;
O1.initliaze();
O1.increment ();
return 0;
}
```

Figure 5.2: C++ programming example (interfaces/implementation based).

access to data. Again, if the structure's implementation changes all programs that use this structure must also be changed since there is no interface available as we have with classes to modify data members and thereby ensure that data remains in consistent state. Other disadvantage with structures is that they cannot be compared in their entirety; they should be compared member by member.

5.4 Constructors and Destructors

In the earlier example(class EXAMPLE), we had two member functions of which one was to initialize the data member of the class. Also, C++ does not allow the data members to be initialized at the point of declaration. It is only via the member functions of the class that we set valid values for data members of the class after having read values for them. The member function initialize initialized the data member (data) of the class EXAMPLE.

But then for the object to be initialized, we require an explicit call by the programmer in the form of `objectname.initialize (5)`. C++ provides an automatic way of initializing objects [initializing the data members that constitute the object] thru constructors. Constructors are also member functions of the class. But they have a syntactic restriction that the member function name should be the same as that of the class name. In fact, the compiler distinguishes between normal member functions and constructor member functions based on the fact that constructors have the same name as that of the class name.

Constructor's very purpose is to set (initialize) data members of the class. Constructors are called automatically when an object of the class is created. Hence they require arguments passing mechanism to be available. Thus constructors similar to normal member functions can accept arguments. Constructors cannot have return types or values associated with them because constructors are expected to perform the object initialization task and not other processing (though syntactically allowed) should be performed within constructors. Thus there is no need to return values after a constructors task is over, for the initialization would have been performed.

The reasoning behind calling these member functions as constructors is that these member functions should construct the object (setting the data members to valid values) whenever a class is instantiated or an object of a class is created. A naive classification of constructors is default and user-or programmer-defined constructors. Default constructors are those that accept no arguments. Again they can be defined by the programmer. If not defined by the programmer, the compiler automatically creates a default constructor. But then with compiler created default constructors there is no surety that the data members of the object would be initialized with valid and programmer desired or expected initializers. Hence it is always advised to go in for programmer-defined default constructors.

Default constructors [either compiler or programmer] are called automatically when the client or end user who creates an object or instance of the class does not supply the initializer arguments at the point of creation like `EXAMPLE 01`. In such cases where object `01` is being created and there are no initializer arguments specified, the compiler calls (automatically) the programmer-defined default constructor, if there is one. Else the constructor automatically creates and class one.

Getting on with programmer-defined constructors, there can be more than one programmer-defined constructor for a class or constructors for a class can be overloaded. This is nothing but the concept of function overloading. For overloaded functions to be called correctly, the functions should differ in their signatures as we have explained already in the earlier chapters. For overloaded constructors to be called correctly, each constructor's signature should differ from the other. Let us now look at an example that makes use of both default and user-defined constructors. Consider the following class definition

```
"Test.h"
Class Test
{
private:  int a;
float b;
public:  Test ();
test (int,float);
display();
```

```
#include "test.h"
Test::Test()
{
a=0;
b=0.0;
}
Test:  Test(int v1,float v2)
{
a=(v1>=0)?v1:0;
b=(v2>=0)?v2:0;
}
};
```

In the above example, we have two constructors defined. The first `Test()` is the default constructor which sets the data members of the objects to valid starting point values. The second constructor is programmer-defined which accepts two arguments and after validation assigns the arguments passed to the data members of the objects. Note that we can exploit the syntax of defaulting arguments when defining constructors.

For example, in the above class `Test;` the user-defined constructor can be specified a prototype or declaration in `Test.h` as follows:

```
Test (int=0,float =0.0);
```

in which case if the values are not provided at the point of object creation, then the default values are assigned. Thus, the user-defined constructor also serves the purpose of a default constructor. There can be only one default constructor for a class. Thus, if a user-defined constructor is defaulting, its arguments then the default constructor should not be defined by the programmer, since there can be only one default constructor for the class.

Now looking at how constructors get invoked at the point of object creation [normally within the test driver function], within main function statement at line (`Test O1`), creates an object `O1` and invokes the default constructor of `Test` class. This explanation is with our earlier syntax of defining a default constructor and a programmer-defined constructor that is not defaulting its arguments. However with the other syntax as well it remains the same. Thus the statement `Test O1` is equivalent to the call `O1.Test()` and the statement `Test O2(3,5.6)` invokes `O2.Test(3,5.6)`. Now with the second method of user-defined constructors defaulting its arguments or the user-defined constructor also serving as a default constructor, the statements:

Test O1: User-defined constructor called with all its arguments defaulted. Equivalent to the call `O1. Test (0, 0.0)`.

Test O2 (5): User-defined constructor called with one argument passed and the remaining defaulted. Equivalent to the call `O2.Test(5,0.0)`.

Test O3(5,5.2): User-defined constructor with none of its arguments defaulted, that is all arguments are passed at the point of object creation itself. Equivalent to the call `O3.Test (5, 5.2)`.

5.4.1 Destructors

Now getting on with destructors, as constructors are used to create or construct or build objects destructors are used to destruct or destroy objects. In terms of resources, the purpose or task of destructors is to reclaim memory that has been allocated for the members of the class when the program terminates and these members will not be used anymore. Destructors are also member functions and carry the same name as that of the class name, but they also have an additional tilde character or symbol (~) preceding the function header.

The reasoning behind this convention is that ~ symbol in computer terminologies associates with the negation or not operator or complement. Thus, it is the complement or negation of constructor and precisely the task of destructor is to destroy or reclaim memory or opposite of constructors. There can be only one destructor defined for a class. The destructor can be either programmer-defined or can be generated by the compiler on the programmers behalf in which case they are referred to as default destructors.

Whether it is programmer-defined or default destructors, they cannot accept arguments or return values. The reasoning behind not allowing arguments to be passed to destructors is that the purpose of destructors is memory reclaiming or what is called termination house keeping. Functions which perform memory reclaiming logic, there will not be any assignment or any other processing logic associated and hence there is no need to pass any arguments.

Destructors are not allowed to return values since they do not perform any processing logic other than memory reclaiming that can be passed on to the external world. Normally, destructors are defined by the programmer only if members are allocated memory dynamically or at run time [using new operator, which shall be explained further]. Members that have been allocated memory dynamically using new operator are reclaimed by applying the delete operator on the dynamically allocated data member within the destructor definition.

Most probably the dynamic allocation of memory using new operator would be performed within the constructor definition. Normally, destructors get called in the reverse order of constructor calls or that is to say the most recently constructed object is destructed first, then the next most recently constructed object and so on. For better understanding, let us consider the following example (the earlier declared `Test` class is being used here):

```
int main()
{
Test O1 (5,6.2);
Test O2;
Test O3(5);
return 0;
}
```

The order in which the constructors and destructors are called is as follows: First the constructor for O1 is called, followed by the constructor for O2 and then finally the constructor for object O3 is called. When the closing brace is encountered (the scope in which the object exists terminates), the destructor for O3 is called first, followed by the destructor for O2, and then finally the object O1 is destructed or destroyed.

Destructors are automatically called when the program terminates and the objects are no longer required to be retained in memory. Normally it is at the end of main (the close {brace}) definition that objects get destroyed or destructors get invoked. But then as we had the concept

of storage classes with variables determining the memory allocation or deallocation objects also have storage classes associated with them and this does change the order of destructors being called. Storage classes and objects shall be explained in the next section in detail.

For the moment, our assumption shall be that objects of a class are created only within the definition of function `main()` in which case the argument that objects get destroyed in the reverse order of construction holds true. As we have stated earlier it is only for dynamically allocated members that the programmer can reclaim memory within the destructor definition using delete operator [will be explained later].

Members that have been allocated memory statically at compile time or by the compiler are automatically reclaimed based on the scope and storage class. The reasoning behind is that programmer can reclaim memory only for those members for which one has allocated memory at run time dynamically and not for those members whose allocations have been done by the compiler statically at compile time. In the above Test example, the destructor declaration or prototype would be `~Test ()`. The definition of this function can contain a display to identify that the destructor of the class has been in fact called. Since there are no members that have been allocated memory dynamically, we do not have any memory reclaiming logic here in this case. However, programs involving dynamic allocations have destructors reclaiming memory [further examples will do so].

Note: Attempting to declare a return type for a constructor and returning a value from a constructor definition will cause syntax errors. Also attempting to pass arguments to destructor or returning value from destructor will cause syntax errors.

If there exists a member function of a class that is performing the operation of initializing the members of the class to valid values, then such a member function can be called within the constructor and thereby avoid repeating code and promote easier code maintenance. In fact, such member functions are normally made private since they are visible to public interfaces or member functions of the class.

It is these private member functions that help in other public member functions or interfaces of the class in achieving their task are referred to as helper or utility functions. Note that constructors and destructors are normally given public access specifiers because their very purpose of existence requires access to such functions from the outside world and hence the justification to provide public access specifier for constructors and destructors.

Declaring default values for arguments of a function in both the prototype [header file] and the member function definition will cause syntax errors. Overloading of destructors is not allowed. Let us now look at another example for better understanding of features discussed so far. The following program should compute the average percentage scored by a student in 8 subjects, code for which is shown in Figure 5.3.

In the above example, the `totalmarks()`, a private member function is the helper or utility that is made use of in computing the average percentage scored by a student in 8 subjects. With the assumption that the logic involved in computation of average is naive, let us now proceed to the next section that elaborates on class member accessibility.

5.5 Controlled Access to Class Members

The member access specifiers public, protected and private in the ascending order of secured access are used to provide controlled access to data members and member functions of a

```
"student.h"
Class Student {
int totalmarks();
int marks[8];
public: Student();
void readmarks();
void setmarks(int,int);
void print();
};
"student.cpp"
#include<iostream.h>
#include"student.h" Student::Student ()
{
for (int i =0;i<8; i++)
marks[i]=0;
}
void Student:: readmarks()
{
int mark;
for (int i =0;i<8; i++)
{
cout <<"Enter mark for subject"<<i;
cin >>mark;
setmarks(i,mark);
}
}
void Student:: setmarks(int subj,int mark)
{
if (mark>=0)
marks[subj]=mark
else
cout <<"Invalid Marks Entry \n";
}
void Student::print()
{
int localtotal = totalmarks();
cout<<"Rounded off average is "<<localtotal/8;
}
int Student::totalmarks()
{
int ltotal =0;
for (int i =0;i < 8; i++)
ltotal + = marks[i];
return ltotal;
}
int main ()
{
Student s1;
s1.readmarks();
s1.print();
return 0;
}
```

Figure 5.3: C++ code for average computation.

class. The default access mode for all class members is private, so that all members after the class definition beginning and before the first access specifier are private. A member's access specifier as we have seen is applicable until the next differing member access specifier or till the end of the class definition [close brace {}].

Intermixing of member access specifiers across statements is allowed but can be quite confusing. Hence it is advised to group all public members first, followed up by protected members and then finally the private members. A class's private members can be accessed only by member functions and friend [will be explained later]. The public members of a class can be accessed from outside or within any function in the program. Public members provide the users of a class an overall view about the class's services or behaviours available. This set of services forms the public interface of the class.

The private members of a class are not accessible to clients. This forms the implementation. A non-member or non-friend function trying to access a private member of a class causes syntax error. The reason behind mentioning the public members of a class at the first is to provide the clients [who will be given this class declaration header file] a clear view of the interfaces that are available for his or her se. Data members made private and member functions public facilitate easier debugging because data manipulation errors are now localized to within the member functions of the class or friends of the class.

Since non-member functions' or non-friend functions' first place do not have any sort of syntactic access to private data members of a class and hence no chance of any data manipulations within non-member or non-friend functions. Access to a class's private data is done carefully by providing appropriate set and get functions that read values for private data from user and set the data members to the read in values after validation on the values read in to ensure the consistency and integrity of the data members.

Private member functions of a class that shall be invoked by other public member or friend functions of the class are referred to as helper or access or utility, or predicate functions. Predicate naming is based on the fact that such helper or utility function that performs simple Boolean conditions checking of the form that results in true or false values. An example would be an `isempty()` helper or predicate function made available within a class Stack to avoid popping when the stack is empty. Thus such private member functions exist purely to help the major public member functions or friends of the class in achieving their overall task.

The earlier example of students uses a utility function called totalmarks that computes the average of marks scored by a student. Such functions are also called accessor methods. Based on the task that they perform, they can be categorized as read or get member functions that just read and return private data values. Set or write member functions that modify the private data values, member functions such as constructors that perform initialization task and other conversion functions as we will see later that convert objects of onto type to another. In fact, such member functions are called conversion constructors and also the helper or utility functions that help the public member functions in achieving their task are referred to as predicate functions.

5.6 Scope of Class Members

The data members [variables declared in the class definitions] and member functions [prototypes declaration] of a class have class scope. Within class scope all the members of the class

are accessible by all the member functions following it and can be directly referenced by name. Member functions can be overloaded by other member functions of the class. Variables within member function definition have function scope [known only within the function].

A member function defining a variable with the same name as that of a data member name, the function scope variables overwrites the data member or class scope variable within the function definition. Such hidden class scope variables can be accessed within function definition by making use of binary scope resolution operator and prefixing the class name before it. Hidden global variables are accessed by making use of unary scope resolution operator. The syntax for the above two operations is as follows:

```
Classname::data member/variable name;
::globalvariablenames
```

Normally, it is the dot operator that is used with object name or a reference to the object to access the object members. A pointer to an object accesses the data members of the object using the arrow operator. These operators are often referred to as the dot and arrow member selection operators, these operators performing the job of selecting members of an object. The same syntax is also used to select members of a structure.

5.7 Order of Constructors or Destructor Calls

Constructors and destructors that perform the task of initializing and destructing the objects are called automatically by the compiler at the point of object creation and program termination when the object is no longer required to be in memory. As we said storage classes does determine the order in which these member functions calls are made, based on the scope in which the class is instantiate or objects of the class are created. Although normally the destructors are invoked in the reverse order of construction, storage classes of objects can alter this ordering of calls.

The following rules hold true:

Objects that exist in global scope have their constructors called first before any other function's (inclusive of main) execution begins. But then no ordering can be guaranteed with respect to constructor calls among global objects [more than one global object existing]. The destructors for such a global object is called when `main()` ends or `exit()` function is called. In case if the program terminates abnormally, i.e. by calling `abort()`, destructors for global objects are not called.

5.7.1 Automatic Local Objects

Constructors are called each time when execution reaches the point where such objects are created. Similar to automatic variables their corresponding destructors are called every time when the scope or block in which subjects are created or exist is exited. Destructors for local objects are not called if either of `exit()` or `abort()` call is made.

5.7.2 Static Local Objects

Constructors are called only once at the beginning when execution or control reaches the point of object creation. Similarly destructors are called when `main()` ends or `exit()` is called.

Abnormal program termination with calls to `abort()`, static local object's destructors are not called. Let us now go through a simple example similar to the one in chapter . We have a class called EXAMPLE, containing a data member i to distinguish objects from one another. The constructors and destructors of the class are defined to have corresponding display statements with the object number, purely for the purpose of identifying the ordering of constructor and destructor calls. The code looks as shown in Figure 5.4.

```
"example.h"
Class EXAMPLE
{
private : int i;
public: EXAMPLE(int = 0);
~EXAMPLE();
};
"example.cpp"
EXAMPLE::EXAMPLE(void)
{
if (val>=0)
i = val;
else
cout<< "Invalid intialiser";
cout<< "constructor called for object" <<i<<endl;
};
EXAMPLE::~EXAMPLE(void)
{
cout << "Destructor called for object" <<i<<endl;
};
"main.cpp"
#include "example.h"
void standalonefunction(void);
EXAMPLE one(1); //global object
int main()
{
EXAMPLE two(2); // Auto local object
Static EXAMPLE three(3); //static local object in main
Standalonefunction(); //standalone function that again creates objects
of class EXAMPLE;
EXAMPLE four(4);
return 0;
};
void standalonefunction()
{
EXAMPLE five(5); //automatic object local to function stanalonefunction
Static EXAMPLE six(6);// Static object local to function
stanalonefunction
EXAMPLE seven(7); //automatic object local to function standalonefunction
}
```

Figure 5.4: C++ code illustrating constructor/destructor calls.

The output of the above code is shown in Figure 5.5. The order in which the constructor and destructor calls are as shown in the above output.

```
Constructor called for object 1
Constructor called for object 2
Constructor called for object 3
Constructor called for object 5
Constructor called for object 6
Constructor called for object 7
Destructor called for object 7
Destructor called for object 5
Constructor called for object 4
Destructor called for object 4
```

Figure 5.5: Output for code in Figure 5.4.

Note: The assignment operator when used with objects performs default member wise copy, where each member of the RHS object is assigned individually to the same member of the LHS object. This is similar to the application of = operator on structure variables.

Objects should be passed to functions as constant references and thereby avoid the duplicating overhead associated with pass by value and the accidental modification or side effect overhead associated with pass by reference. Constant reference is the best way of passing objects to functions.

5.7.3 A Subtle Trap

Returning a reference to a private data member: As we have already seen a reference to an object is an alias for the name of the object and hence can be used LHS of an assignment statement. Thus, the reference makes a perfectly acceptable lvalue that can receive a value. But then this can be exploited by having a public member function of a class returning a non-constant reference to a private data member of that class. Let us consider the following example for better understanding of these concepts, code for which is shown in Figure 5.6. Note that set value that returns a reference to one of its private data members can be modified from outside the class using an assignment statement or as a part of an assignment statement as an `lvalue`.

5.8 Constant Member Functions and Objects

The principle of least privilege is one of the key features of object-oriented programming. The constant qualifier to a great extent is helpful in implementing the principle of least privilege, which is one of the most fundamental principles of good software engineering. Certain objects require being modifiable while certain other objects require only read access. An object that is declared to be constant and further attempts to modify it is a syntax error. Declaring variables and objects as constant can improve performance since constants are interpreted in optimized forms by certain compilers.

Only constant member functions can be invoked on constant objects. Constant member functions cannot modify the objects state or to be precise, they cannot modify the data

```
class Test {
public: Test (int =0, int =0);
void setvalue(int,int);
int getval();
int &set value(int);
private : int a;
int b;
};
Test :: Test (int v1,int v2)
{
a=v1;
b=v2;
}
int & setvalue (int v3)
{
a = v3;
return a;
}
int main()
{
Test O1;
int &aref=O1.setvalue(50);
aref=30;//undesired modification
O1.sevalue(50)=70;//returned reference used as lvalue
return 0;
}
```

Figure 5.6: Member functions returning references.

members of the objects. The **constant** keyword is mentioned in both functions prototype and definition after the function header. Member functions that require only read permission can be declared to be as constant functions. The **constant** keyword should be mentioned at the end of both function declaration and definition.

Note: A member function that has been declared to as constant and modifies the object state or its data members is a syntax error. Also, invoking non-constant member function on a constant object is a syntax error. As normal functions can be overloaded, even non-constant member functions can be overloaded with their non-constant counterparts. The compiler resolves overloaded member function calls based on the objects nature, since only constant member functions can be invoked on constant objects.

The only exception with constant objects with respect to constant member functions being called on them is that constructors and destructors of constant objects cannot be declared as constant since constructor's very purpose is to initialize objects data members and destructors purpose is to perform termination clean up; both of which require write or modification permissions. Thus constant declaration is not allowed for constant objects. Thus even though constructors or destructors are non-constant member functions, they can still be invoked on constant objects. Possible combinations of member functions type and object types are as shown in Table 5.1.

Table 5.1: Member function types and objects

Object type	*Member function type*	*Allowed/disallowed*
Constant	Constant	Allowed
Constant	Non-constant	Disallowed
Non-constant	Constant	Allowed
Non-constant	Non-constant	Allowed

Readers need to differentiate the fact that with constant objects [data members cannot be modified] while with constant functions no modifying logic can be specified within its definition. Only read type operation or code can be specified within the function definition of constant functions. Constant data members of a class cannot be initialized using an assignment statement. Constant members of a class are initialized using member initializer syntax.

Non-constant data members can also initialized using the member initializer syntax. In case there are multiple constant class members, the member initializer list is comma separate. Composition is another feature of OOPS, wherein objects of other classes are by themselves members of a class or objects of classes are contained or composed within a class definition also makes use of the member initializer syntax to initialize the composed object data members, [indirectly or actually] calling the constructor of the contained - composed object. Let us now look at a programming example to understand the member initializer syntax.

```
class Test {
private :  int data;
const int value;
public:  Test (int=0,int=1);
void addval()  data + = value;
void display () const;
};
Test::Test (int d,int v):value(v)
{
data = d;
}
void Test::display() const
cout << "data is" << data << "Value is" << value << endl;
}
```

Note that in the above code the constant value data member is initialized in the constructor header using a member initializer operator (: colon). Trying to initialize a constant data member of a class using an assignment statement is a syntax error.

Not providing member initializer syntax for constant data member is also a syntax error. All member functions that do not modify the object should be declared as constant. This allows such functions to be invoked on constant objects and also prevents any object modifying (accidental side effect) statement within the definition. Attempts to do so will be caught at compilation stage itself.

5.9 Composition—Objects as Members of Classes

Composition is another key feature of OOP wherein a class has objects of other classes as its members. Realistically, the software engineering observation has been that end users or clients require objects of already defined classes as its members. For better understanding, let us consider the following example. An organization is wishing to create employee information details. Details of employees include first name, last name, hire data and birth date. We shall incorporate composition concept of OOP to achieve the task. We shall have two classes employee and date whose data members are as follows: **Date : month, day** and **year**—all of type integers. **Employee: fname, lname, Birth_Date, Join_Date**. The declaration of class **Date** is as shown in Figure 5.7.

```
"date.h"
class Date {
private : int month;
int day;
int year;
int check day(int);
public: Date (int =1,int =1,int=2003);
void display() const;
~Date() {cout<<"Destructor for date object"<<endl; }
};
"date.cpp"
#include"date.h"
Date::Date (it m,int y, int d)
{
cout<<"constructor for date object"<<endl;
if(m>= 0 && m<=12)
month = m;
else
month =1;
year =( (y>1900 && y<2075)?y:0);
day = checkday(d);
}
int Date:: checkdate(int today)
{
static constint daysofmonth[13]=0,31,28,31,30,31,30,31,31,30,31,30,31;
if(today >0 && today <=daysofmonth[moth])
return today;
if(month ==2 && today ==29 && (year % 400 ==0) ||( year % 4==0 && year
% 100 !=0))
return today;
count <<"Invalid Day Value"<<endl;
return 1;
}
"employee.h"
#include "date.h"
class Employee {
private : char fname [25];
```

Figure 5.7: Continued

```
char lname [25];
const Date Birth Date;
const Date Join Date;
public: Employee(char *, char *, int,int,int,int,int,int);
void display() const;
~Employee();
};
"employee.cpp"
#include"employee.h"
#include"date.h"
Employee::Employee (char *fn,char *ln,int bm,int bd,int by,int jd,int
jm,int
jy):Birth Date(bm,bd,by),Join Date(jm,jd,jy)
{
int length = strlen(fn);
strncpy(fname,fn,length);
fname[length]='\0';
Continued on Next Page
5.9. COMPOSITION-OBJECTS AS MEMBERS OF CLASSES 141
int length = strlen(ln);
strncpy(lname,ln,length);
lname[length]='\0';
cout << "constructor for employee object" <<fname<<lname<<endl;
}
void Employee::display () const
{
cout <<lastname<<firstname;
Join Date.print();
Bith Date.print();
cout<<endl;
}
Employee::~Employee()
{
cout<< "destructor called for employee" <<fname<<lname<<endl;
}
"main.cpp"
#include"employee.h"
#include<iostream.h>
int main()
{
Employee e1("Siva","Selvan",23,7,1978,16,7,2003);
e1.display();
}
```

Figure 5.7: C++ programming example for class composition.

When an object is created its constructor is called automatically. In cases of classes con
ing or containing objects of other class care should be taken to pass arguments to me
object constructors and to invoke member object constructors properly. Member objec
constructed in the order in which they are declared within the class definition and not
order in which they are listed in the constructor's member intializer list. In fact, th
structors of the member objects are invoked before the composing object or containing
object constructor.

Observe that the constructor of the enclosing class (employee) accepts arguments inclusive of the member objects constructors. In case of the member initializer list missing, the default constructors of the member objects are invoked. Again if the default constructors are not provided and the member initializer list is missing as well, this leads to syntax error. A class that has as its member another object with public access specifier encapsulation and hiding concepts of that composed objects private members remain intact. The output of the above example illustrates the order in which constructors and destructors calls are made.

5.10 this Pointer

Every object has access to its own address through a pointer called **this** pointer. The **this** pointer is not reflected in the size of operation on the object since the **this** pointer. The **this** pointer is passed onto the object implicitly by the compiler every time a non-static member function call is made to the object.

The **this** pointer can be used to reference both data members and member functions of an object. It can be used both implicitly as well as explicitly. The type of **this** pointer depends on type of object and the member Function where **this** pointer is referenced is a constant function or not. In a non-constant member function of class **Test**, the **this** pointer is of type **Test * const**. In a constant member function of the class **Test**, the type of the **this** pointer is const **Test * const**. Every non-static member function has access to **this** pointer to the object for which the member function is being invoked. Let us now look at a simple example for the understanding of the concepts explored with respect to the **this** pointer, code for which is shown in Figure 5.8.

```
class Test
{
private : int x;
Test (int =0);
void printf() const;
};
Test :: Test (int a) {x = a;}
void Test::print () cosnt
{
Cout << x << this→x<<(*this).x;
}
int main ()
{
Test O1(2);
O1.print();
return 0;
}
```

Figure 5.8: C++ code to illustrate the use of **this** pointer.

The parenthesizing in line is required because of operator precedence rules. If the expression is not parenthesized, since . carries higher precedence over *, the expression would be interpreted as *(this.x), however, the dot operator cannot be used with a pointer and hence it leads to a syntax error.

Two main applications of **this** pointers are to enable cascaded member function class and prevent object self assignment as will be explained in the later sections. In fact, the cascaded member function call will be explained in the following section.

5.11 Cascaded Member Function Calls

Cascading of member function calls is nothing but the process of invoking several member functions of a class with respect to a single object and in a single line statement of the form `o1.input ().output ().exit ()`. Here `input`, `output` and `exit` are three member functions of some class and should get executed or invoked with object `o1` of the respective class. This feature is also called chaining of member function calls. Let us now look at an example that implements the cascading of member functions, code for which is shown in Figure 5.9.

```
class Test {
private: int mem1;
int mem2;
int mem3;
public: Test (int =0,int=0,int=0);
Test & setmember1(int);
Test & setmember2(int);
Test & setmember3(int);
Test & setall(int,int,int);
};
Test::Test (int v1,int v2,int v3)
{
setall (v1,v2,v3);
}
Test & Test::setall(int val1,int val2,int val3)
{
setmember1(val1);
setmember2(val2);
setmember3(val3);
}
return *this;
}
Test & setmember1(int val1)
{
mem1=val1> 0?val1:0;
return * this;
}
Test & setmember2(int val2)
{
mem2=val2> 0?val2:0;
return * this;
}
Test & setmember3(int val3)
{
mem3=val3> 0?val3:0;
return * this;
}
int main ()
{
Test O1,O2;
O1.setmember1(5);
O1.setmember2(10);
O1.setmember3(15);
O2.setmember1(20).setmember2(25).setmember3(30);
return 0;
}
```

Figure 5.9: C++ code supporting cascaded member function calls.

The above example has three integer data members and four public interfaces, three of which set the individual data members and one among them (int,int,int) sets all the three data members at one stretch. All the four interfaces are scheduled to return a reference to class Test, so as to support cascading.

The cascaded member function call in line of main() is parsed left to right (since the dot operator associates left to right). First, the member function setmember1 is executed with respect to object O2. At the end of its call it returns the this pointer or the address of the object with respect to which the member function was invoked. Thus, the next function call is executed with respect to this returned address. Thus, this pointers have a crucial role to play in supporting cascading of member function. We have just illustrated with one instance of cascaded member function calls. Readers are advised to try all possible combinations of calls to setmember1, setmember2, setmember3 and set all in a cascaded fashion. For better understanding of the cascaded member function call, the parsing syntax is shown below

```
(((O2.setmember1(20)).setmember2(25)).setmember3(30)); 1 2 3
```

The parentheses and the numbers below them indicated the order in which the entire cascaded call is processed. At the end of each call, as we have said earlier the address of the object with respect to which the member function was called is returned. Thus, the object with respect to which the remaining function calls should proceed is available. This avoids repetition of object name with each function call(s).

5.12 Dynamic Memory Allocation

C vs. C++

The new and delete Operators available in C++ are used to allocated memory dynamically at run-time. C's mechanism of allocating memory dynamically is using the malloc and free function calls. Static allocation of memory [at compiler time] may cause wastage of memory space. The statements in C to allocate memory are as follows:

```
int * eptr;
eptr= malloc (sizeof(int));
```

The above statement dynamically creates an object of the type int. The corresponding statement to release the memory allocated using malloc is free(eptr); or in general free(ptrname). Memory allocation in C requires an explicit call of function malloc and within it use of the size of operator. Certain early versions of C would also require the pointer returned by malloc to be casted to the type of ptr on the LHS side of the assignment statement. And also the allocated block of memory using malloc call is not initialized for which calloc is to be used. In C++, the equivalent statement to allocate memory dynamically for the above type is eptr=new int; assuming that eptr has been suitably declared to be a pointer to an integer.

The keyword new is an operator that automatically creates an object of the correct size, then automatically calls the constructor for the object and returns a pointer of the type for which memory has been created. In cases of memory allocation being most successful, the new operator automatically throws an exception indicating the failure of memory allocation. Exception handling another feature of OOP can be used to handle exceptions as we will discuss

in the chapter - on Exception Handling. To reclaim memory or destroy the object for which memory has been allocated using `new` operator, the `delete` operator is used as `delete eptr`. C++ allows initializers to be provided for a newly created object as follows:

```
Double * eptr = new double (9.67);
```

Array declaration (dynamic allocation) in C++ is as follows:

```
int * a = new int [20];
```

Reclaiming the space allocated for `a` is by `delete [ ] a`. The main advantage of using `new` and `delete` operators is that the constructors and destructors of the class get called automatically. However, `new` and `delete` operators when mixed with `malloc/free` function calls cases logical error. Space that has been created by `malloc/new` must be released by `delete/free` operator only. Also an array that has allocated space dynamically should be deleted using `delete []`. Application of just delete on an array causes logical error.

Memory created as an array should be deleted with `delete []` operator and memory created for an individual element should be deleted with the `delete` operator. After our discussion on static members of a class, we shall develop a program that involves dynamic memory allocation. The application of size of operator on an object reflects only the size occupied by the data members of the object and not the member functions size. This was because only the data members uniquely distinguish objects and require explicitly memory allocations. Thus, all data members are allocated separate memory allocations for each instance of the class. One exception from this is static class members.

Static members of a class are not allocated individual memory locations for each object but they are shared across objects of the class. There is only one copy of static members that is maintained in memory. In fact, static members are class-wide property or information and not object-specific or object-wide. Static members are created on a per class basis rather than on a per object-basis. The declaration of static class members begins with the keyword `static`. Thus, static members are a property of the class and not a property of the specific objects of the class. Static members have class scope. The access specifiers of public or protected or private are applicable with static members as well.

A class's public static members can be accessed through any objects of the class or when no objects of the class are existing. They can be accessed using the class name and the binary scope resolution operator. A class's private or protected static members can be accessed only through public member function or through public member function or friends of the class. To access such members when no objects of the class are existing, a public static member function must be provided and this function must be called with the class name and the scope resolution operator when objects are existing, a static member can be referenced via any of the objects of the class. All reference refers to the shared copy of the static member. The student example shall be modified to allocated space dynamically and count the number of students that are created at any point of time, code for which is shown in Figure 5.10.

The count data member which is a static class should reflect the total number of student objects (students) that have been created. Each time an object of class student is created (at which point constructor of the student class is called), the count should be incremented. Hence this incrementing logic is placed in the constructor definition so that each time an object is created the static (common across objects) member count is suitably incremented. Since count is a private static data member, a static function called `readcount ()` that returns the value of count is provided.

```
"student.h"
class Student {
private: char * firstname;
char * lastname;
static int count;
public: Student(const char *, const char *);
~Student();
static int readcount();
void print() const;
};
"student.cpp"
#include "student.h"
int Student::count = 0;
int Student :: readcount () {return count ; }
Student :: Student (const char * fn, const char * ln)
{
firstname = new char [ strlen (fn) + 1 ];
strcpy(firstname,fn);
lastname = new char [ strlen (ln) + 1 ];
strcpy(lastname,ln);
++count;
cout << "Constructor called for"<< firstname << lastname;
}
Student :: ~ Student ()
{
cout <<"Destructor being called for"<<firstname<<lastname;
cout<<endl;
delete [] firstname;
delete [] lastname;
- -count;
}
"main.cpp"
int main ()
{
cout<<"Count of students before any student object is created \n";
cout << Student :: readcount;
Student * s1 = new Student ("Siva","Selvan");
Student * s2 = new Student("Ganesh","Ramani");
cout<<"Count of students after student objects have been created \n";
cout<< s1→readcount();
cout<<"Student 1 details \n";
s1→print();
cout<<"Student 2 details \n";
s2→print();
delete s1;
delete s2;
s1=0;
s2=0;
cout<<"Count of students after student objects have been deleted \n";
cout << Student :: readcount();
}
void Student : printf () const
{
cout <<"First name is" << firstname <<endl;
cout <<"Last name is" << lastname <<endl;
}
```

Figure 5.10: C++ code illustrating the use of **new** operator.

Note that when there are no objects of the class student existing, the static member is referred to using the class name and the scope resolution operator. When objects of the class are existing, static data members can be referred to using one of the objects or using the class name with the :: operator. Member functions can be declared to be static if only they do not refer to non-static class members. As we have already stated, static member functions do not have `this` pointer associated with them because static data members and static member functions exist independent of the objects of the class. Note that static members [data or functions] of a class exist even when there are no objects of the class existing and can be used or referred to.

5.13 Friends

The dictionary or real time meaning of the word friend is somebody who can be trusted or confided in (expressing confidence). It is based on this trust that we have the concept of friends supported in C++. The access specifiers private or public or protected provided varying levels of access to data member or member functions to the external or the outside world. As it is only public members of a class can be accessed from outside.

Protected members of a class can be accessed from outside the class but only by derived class objects. Private members which provide the peak secured access can be accessed from only within the class or from outside via public member functions of the class. One exception to this is friends. Friends can access private members of a class directly from the external world or outside the class. A function or an entire class can be made a friend to another class. Friend function's extensive use is in operator overloading which will be explored in the next chapter. The keyword `friend` is used to declare a class or a function as a friend of another class. For example, to declare a function `f1()` or a class B as a friend of class A, the following syntax shown in Table 5.2 should be adopted.

Table 5.2: Declaration syntax for friends

`Class A`
`{`
`friend class B;`
`};`
	`Class A`
	`{`
	`friend void f1();`
	`};`

The class or function which is to act as a friend of a class, the corresponding function or the class which is to act as a friend to a class should be included in the class definition only [header file] with the keyword `friend` preceding the function or class as done in the above statement. The first code declares class B to be a friend of class A and the second code declares function `f1()` to be a friend of class A.

Note friends are neither data members or member functions of the class. It is just expressing at most confidence in providing access to all its members irrespective of the access specifier to the external world. Since access specifiers are not applicable with friends, friend declarations can be placed anywhere in the class definition. Friendship should be explicitly granted by a class and cannot be assumed or taken for granted.

For a class y to be a friend of class x, the class x should explicitly declare that y is a friend of it as was done in the example. An example involving friend functions, where the private

data member of a class is accessed and modified by a friend function is shown in Figure 5.11. Note how the data member data of Example object `exmp` is set and accessed from outside the class by non-member function `fn1`. This is possible only because the class has declared the respective function to be a friend and hence its private members are accessible to it.

```
 1. #include<iostream.h>
 2. class Example
 3. {
 4. friend void fn1(Example &,int);
 5. void display()
 6. {
 7. cout<<"Data Member Value is"<<data;
 8. }
 9. public:Example()
10. {
11. data=0;
12. }
13. private:
14. int data;
15. };
16. void fn1(Example &e,int d)
17. {
18. e.data=d; //this is allowed as a result of friends feature
19. }
20. int main()
21. {
22. Example exmp;
23. exmp.display();
24. fn1(exmp,5);//new value for data member set by a nonmember
    but friend function.
25. exmp.display();
26. return 0;
27. }
```

Figure 5.11: C++ code to illustrate `friend` functions.

Friendship is neither symmetric nor transitive. If a class x is a friend of class y, y cannot be assumed to be a friend of class x for which class x should explicitly declare that y is a friend of it. On similar lines, if a class x is a friend of class y and y is a friend of class z, transitive inference that x is a friend of z is not valid. Such properties of binary relations does not hold true with friendship relationship. Overloaded functions can as well be declared or made friends to other classes.

Review Questions

1. List the various key object oriented programming features.
2. Explain the term data encapsulation and highlight the same with respect to the basic unit of programming in an object oriented language such as C++.

3. Justify the reasoning behind the generally used access specifiers of private for data members and public for member functions of a C++ class.
4. What is the use of the scope resolution operator? Can member functions be defined inside the class definition? Explain.
5. Differentiate the newline escape sequence from endl stream manipulator.
6. Develop a C++ program to generate the factorial of a user-specified number. Use C++ style input and output statements (use of classes is not required).
7. Develop a C++ program to matrix manipulations of addition, subtraction and multiplication. Use classes.
8. Define a constructor and identify the general responsibilities of a constructor for a C++ class.
9. Develop a C++ program using classes that implements the sorting algorithms of insertion, bubble and selection sort.
10. Explain the use of `this` pointer in C++.
11. Explain class composition and differentiate it from data encapsulation.
12. Define dynamic memory allocation and allocate an integer array of size 30 dynamically using `malloc` and new features of C and C++. Appreciate the limitations of `malloc` in comparison to the benefits offered by new operator.
13. Discuss the need for friend functions in C++ with an example.
14. Develop a Class Time package in C++ and support functions to convert time expressed in 24 hours format to 12 hour format (assume data members of hour, minute and second respectively).
15. Develop a C++ Calendar package that displays the calendar for a given month input. Also support a function to display the day given the date of the month.

CHAPTER 6

Operator Overloading

In the previous chapter, we have discussed class or object creation. Manipulations on class objects were accomplished be sending messages in the form of member function calls to objects. But then with mathematical classes, the member function call syntax is complex. The C++'s operator set could be exploited using the concept of operator overloading. Operator overloading is the process of assigning new meaning(s) or operations to operators from the C++ operator set. It is referred to as overloading because more than one meaning is associated or assigned for operator or new meaning(s) or operations in addition to the existing meaning are assigned for the operator.(or) in other words, operators are overloaded to perform different operations as per the requirements.

One simple example for an already overloaded operator in the language is << and >> operator. The first operation is the bitwise left shift and right shift operations. The other overloaded operations associated with the above operator are stream insertion or extraction [output-input statement] in C++. Another classical example of overloading is the binary + operator, since the + operator performs overloaded operations of pointer, integer and floating number arithmetic.

Depending on the context in which it is used overloaded operators perform the intended or correct operations. Almost (except of a few that shall be mentioned) all the operators of C++ operator set can be overloaded. In this chapter, we shall explore the implementation details of operator overloading. But then operator overloading can always be substituted by the member function syntax, but then it is one solution wherein the appropriate member function call is performed automatically by the compiler, not expecting the programmer or end user to explicitly call the appropriate function. It is just a more realistic and convenient way of manipulating class objects.

6.1 Introduction

Operators that are overloaded or assigned meaning by the programmer already have a meaning or operation associated with them. When we stated that overloaded operators perform the appropriate operation depending on the context in which they are used, it is nothing but the operand type. Thus, when an overloaded operator is made use of or applied with built-in data types, the existing (normal) meaning of the operator is in force. When used with objects or user-defined data types, the newly assigned or overloaded meaning is in force. Operator overloading though a powerful programming technique should be used carefully. An example

of operator overloading would be to define the + operator to perform addition of matrices [assuming we have a user-defined data type class Matrix].

Operator overloading is achieved or implemented by defining a function with the keyword operator followed by the operator that is to be overloaded [in fact, this is the name of the function or the function identifier] and within the function definition or body of the function the logic associated with the operator's new function or meaning is specified. To overload the operator +, the function header would be of the form operator +. Operator overloading can be implemented via both member functions as well as non-member functions. The choice between a member function and a non-member function overload depends on the operator's application. We shall provide details on this choice in the next section.

6.2 Rules to be Followed in Operator Overloading

6.2.1 Overloaded Meaning should be Realistic

Operator overloading should be intuitive and realistic. That is to say that though it is always possible to syntactically overload the + operator to perform subtraction on user-defined data types, such an overload that is unrealistic and confusing. C++, for that matter any OO language should support reusability and extensibility. Programmers who are to reuse or extend or maintain such code would proceed with the natural or normal conventions and would require certain effort to first place spot the meaning or operation associated with the respective operator. To put it in simple words, such overloading is unscrupulous and idiotic.

6.2.2 Built-in Operator Overloading Support

Two operators that can be used with both built-in and user-defined data types [class objects] with no overloading required are the = (assignment) and the & (address of) operator. These two operators when applied with built-in data types or used-defined data types perform the operations of assignment and returning the address of the [built-in or user-defined data type] in memory. As we have already mentioned, operator overloading is more often required for classes representing mathematical entities such as complex or rational numbers, wherein it would be extremely realistic or convenient to overload the arithmetic or relational operators on such user-defined data types. And in such cases, the requirement would be to overload quite a few or substantial number of frequently used operators.

6.2.3 User Definition Support

Operator overloading mechanism is to provide a precise or concise notation for manipulation of user-defined data types. Operator overloading similar to friend functions is neither granted nor automatic. It has to be explicitly declared and defined by the programmer. The programmer must explicitly define the operator functions to perform the required operations. A good programming practice is to overload operators on user-defined data types, to perform operations that are quite similar to operations on built-in data types. As a part of our discussion on friends, we have mentioned that friends feature of C++ shall be used in operator overloading.

6.2.4 Methods and Precedence of Overloading

Operator overload functions can either be member functions of user-defined data types or friend functions or non-member, non-friend functions. Most of C++ operator can be overloaded, the exception's being `*, ::, ?:` and size of operator. Attempting to overload such non-overloadable operators cause syntax error. The other dont's of operator overloading are as follows. The precedence of an operator [language-dependent] cannot be changed by operator overloading, since allowing it would cause the existing or fixed precedence vs. programmer-defined precedence determination is complex. An easy way out from this precedence bottleneck is to exploit the parenthesizing to change the order of evaluation of overloaded operators.

6.2.5 Associativity and Arity cannot be Changed

The associativity of operators cannot be changed by overloading. The number of operands which the operator takes or operates on cannot be changed by overloading. This is also referred to as arity of operators, the naming from the existing convention of unary, binary and ternary operators. Thus, the arity of operators cannot be changed via operator overloading. Thus, unary or binary operators can be overloaded only as unary or binary operators. The ternary operator `(?:)` cannot be overloaded. Certain operators that have both unary and binary operations associated with them such as `&` → address of (unary) and bitwise, and (binary) and `*` → dereferencing (unary), and multiplication (binary) can be overloaded separately.

6.2.6 New Operators cannot be Created

One syntactic compulsion with operator overloading is that programmers can only overload existing operators [operators defined in the language set]. The programmer cannot altogether create new operators, only existing operators can be overloaded. Thus, assuming that one can create operators such as `:=` or `**` is invalid. Operators that do not belong to the C++'s operator set cannot be overloaded. Attempts to create new operators via operator overloading will cause syntax errors. The compiler simply does not recognize such new operators. Thus, via operator overloading it is only new meanings for existing operators and not new operators that can be created. And also the context or meaning of how an operator works on objects of built-in types cannot be changed by operator overloading.

The control lies with the programmer only for manipulations on objects of user-defined classes. With built-in data types or objects, it is always the predefined meaning that is applicable. Operator overloading always works only on or with objects of user-defined types or with a mixture of user-defined and built-in-type. But at least one operand of an overloaded operator must be of user-defined type for the overloaded meaning to be in force. When all the operands of an overloaded operator is of built-in or existing (fixed) meaning of operator that is applicable or in force. Thus, the powerful programming feature of operator overloading wherein programmer has control over meaning or operations of operators lies only with user-defined data types.

6.2.7 Operator Overloading should be Explicit

Operator overloading should be always explicitly specified by the programmer. There is no implicit overloading performed by the compiler on the programmers behalf. Assuming that

shorthand notation (such as +=) operations are overloaded when their basic forms [+,=] are overloaded is a syntax error. That is, if + & = operators are overloaded individually += is not overloaded implicitly and if required should be overloaded explicitly by the programmer. Another example for implicit overloading not possible is that related operators do not get overloaded implicitly. That is assuming that on overloading ==, != is also overloaded is a syntax error. All operators need to be explicitly overloaded by the programmer. But then already overloaded operators should be utilized to the extent possible. That is, if + and = have been overloaded by the programmer, they should be used while overloading +=. Code repetition should be avoided always, since they are cumbersome in terms of both code size and program maintenance.

6.3 Methods of Defining Operator Functions

In the earlier section, we have mentioned that operator overloading functions can either be member functions or non-member functions, or friend functions. In case of non-member function overload, all the arguments (mostly two, with binary operators in mind) must be explicitly passed to the function, since it being a non-member function would require information about all the values (operands to be precise). In the case of member function overload, only one argument should be passed. Since it is a member function, the objects with which it gets called or associated is already known and hence for binary operator overload via member function only one operand is required.

6.3.1 Member Function Overload

For better understanding overloading of binary operators like `a+b` [assuming `a` and `b` are user-defined class objects] would be parsed as `a.+(b)`. Thus, the object with which it gets invoked is actually the LHS operand of binary operators. Hence only one operand is to be passed to the function call explicitly; the other operand is implicitly taken by the compiler. In fact, the compiler uses `this` pointer to obtain the operand. `This` pointer is nothing but the object for which a member function was executed most recently, nothing but the object or operand that was not passed as a part of the function call. That is for the expression `a + b` [again `a` and `b` are assumed to be instances of user-defined types], we already stated the parsing would be of the form `a.+(b)`. Thus, within the member function [operator `overload function→ operator + (b)` referring to the this pointer is actually referring to the address of object `a`.

6.3.2 Non-member Function Overload

The operator function mentioned within braces above might not be entire in its signature. It is from the number of operands required point of view that operator overloading is explored in this section. We shall be complete with the signatures of operator overload functions in the programming examples that are to follow. Thus, the LHS operand of a binary operator must be always an object of the class of which it is scheduled to be a member function. If the LHS operand will be an object of a different type [but not an instance of the class to which it can be a member function], the operator overload function should always be a non-member function. And if such non-member functions require access over private data members of another class,

they are better off to be declared as friend functions of such classes whose private members may be referred to within this non-member operator overload function.

A classical example for a friend non-member function overload would be the overloading of `<<, >>` operators with respect to user-defined class objects. Let us assume we have a user-defined class called `Integer_Array`. A statement of the form `Integer_Array o1 (10)` would create an integer array of size 10, initialized to 0's. Assuming that the array has been assigned values [some other member function available], the programmer at some stage or the other would require to display the array contents. It would be convenient if a redirection or output statement of the form `cout << o1` to display all the array contents [in this case `o1`] is available. Rather than the end user of this class having to go through each index of the array, this indexing logic can be specified as a part of the operator overloading logic for the `<<` operator. In other words, the end user statement of the form `cout << o1` would be interpreted by the compiler as the new overloaded meaning of the `<<` operator to redirect the entire objects contents on to the screen.

Readers should observe that in a statement of this form `[cout << o1]`, the LHS operand of the operator `<<` is always an instance of the built-in class ostream [`cout` is an object of class `ostream`]. Thus, such operators should always be overloaded as non-member functions. And also since the redirection operation would often require access over private data members [the array contents], the `<<` operator is normally overloaded as non-member friend function of the user-defined class `Integer_Array`.

In the next section, we shall get into the programming intricacies or niceties of these `<<` and `>>` operators overload. As is obvious the `>>` operator would also be overloaded as non-member friend function since LHS operand of `>>` will always be of type built-in class `istream.[cin]` and again would require access to private data members of the user-defined class `Integer_Array`. The last form of operator overload, non-member, non-friend function is as well possible but then such a function to access the private data members of a class [user-defined] with which the operator is applied, public interface functions should be available in the class which should perform this job. Thus having discussed on member vs. non-member operator overload function issues, for better understanding, let us go though this example.

A non-member function overloaded operator such as `cout << userdefinedobj;` is parsed by the compiler (or leads a call to) operator `<<(cout,userdefinedobj)`. A member function overloaded operator such as `+` when used as follows `a + b` is parsed by the compiler as `a.operator + (b)`. In both the above examples it is assumed that corresponding operator functions have been appropriately defined. Thus, operator member functions of a user-defined class are called only when the left operand of a binary operator is always an object of that class or when the single operand of a unary operator is an object of that class.

One valid reason to go in for a non-member function overload is to make the operator support commutative property. For better understanding of this, let us assume `o1` and `o2` are instances of two user-defined classes A and B. If `o1` and `o2` are to be added [assuming A and B are representing mathematical entities], the statement issued will be of the form `o1 + o2`. Let us assume that `+` has been overloaded in class A as member function in which case the compiler parses the above expression as `o1.operator +(o2)`. No problems as of now.

With normal arithmetic `+` is commutative and based on the principle that operator overloading will be to the extent possible identical to normal or fixed operations, the `+` operation on objects of `o1` and `o2` should be commutative. That is a statement of the form `o2 + o1` should be valid. But if `+` is overloaded as a member function of class A, then the LHS operand

should always be of type class A, whereas in this case or for + to be commutative, the LHS operand can also be an in stance of class B. Thus, at least operators that need to support commutative property should be overloaded as non-member functions. For example, in the above case if + is overloaded as a non-member function (can also be a friend function), the statements(s) `o1+o2 or o2 + o1` get parsed as follows:

`operator + (o1,o2)`→ first case and operator `+ (o2,o1)`→ second case

In all the signatures that we have seen with respect to this chapter on operator overloading, we have not expressed concern as to how passing arguments [objects, which are operands to overloaded operators] by value or by reference. This was with purpose, to understand the issues involved in operator overloading in a comprehensive manner. The above sections were devoted to operator overloading. It so happens that operator overloading is implemented using a function with the keyword `operator`. We did not want to confuse readers with issues of passing by reference of by value or by constant reference, at least while discussing operator overloading. The programming examples to follow, we will mostly pass objects [which are nothing but operands to the overloaded operator] by reference, if modification or update is required and as constant reference if only read permissions are required over the object or operand.

Note: The reasoning behind the LHS operand of a binary operator to be always of class type where the operator is overloaded as a member function is that the compiler, for all member function overloads, associates the operator function call with the LHS object or operand. Thus, if a binary operator that has been overloaded as a member function of one user-defined class and is applied with an object of built-in or some other user-defined class type on the LHS of the operator, the compiler [after having searched for a non-member function version of the overloaded operator] would associate with which it gets associated on the LHS, where it would not be able to locate one such function. Thus this is the reasoning behind the syntactic restriction that LHS operand of a member function overloaded operator is always of the respective class type. This is a syntactic restriction and violations if are resolved at compilation stage itself. We have had enough on issues involved in operator overloading. It's high time we get exposed to programming examples involving operator overloading. The following sections will elaborate on overloading of various operators from C++ operator set.

6.4 Overloading of << and >> Operators

Let us assume a student information [details of Name, Register Number and Age] is to be created. We will create a class called Student whose data members are Name, Register Number and Age. We shall overload the stream insertion (<<) and stream extraction (>>) operators for input or output of student information at one shot. [`eg:cout<<student1; cin>>student1`] should either accept or redirect values for all three members values to the screen] obviously the LHS operand being an instance of a predefined class [istream or ostream] we shall go in for a non-member friend function overload since both operators require access over private data members of the class. The code is as shown in Figure 6.1.

```
1.  class student
2.  {
3.  friend ostream & operator << (ostream &, const student &);
4.  friend istream & operator >>(istream &, student &);
5.  private : char name [50];
6.  char regno[30];
7.  int age;
8.  };
9.  ostream & operator<<(ostream & o, const student & stud)
10. {
11. o <<"Name of Student is \n"<<name<<endl<<"Register
    No is \n"<<regno<<endl<<"Age is"<<age<<endl;
12. return o;
13. }
14. istream & operator >> (istream & i, student & stud)
15. {
16. i >> "overloaded Input Operator";
17. i >> stud.name;
18. i >> stud.regno;
19. i >> stud.age;
20. return i;
21. }
22. int main ()
23. {
24. Student S1;
25. cout <<"Enter student details \n";
26. cin >> S1;
27. cout <<"Displaying student details \n";
28. cout >> S1;
29. return 0;
30. }
```

Figure 6.1: C++ code to overload input and output operators.

6.5 Code Interpretation

The student class has two non-member friend functions, operator << and operator >> each of which accepts the respective stream reference as an argument, because they are overloaded as non-member functions. The output operator function accepts the student object as constant reference, since it is only a read operation and read access is what is required over the student object. The input operator accepts the student object as a non-constant reference, since it being an input operation, one would require write permissions over the object. In both cases, arguments are passed by reference to avoid the duplicating overhead associated with pass by value mechanism. Now let us get on with the definition of the operator overload functions.

The named `ostream` reference is used within the definition of the output operator definition and each of the three data members of the student class are redirected to this named ostream reference (o) separately. The definition of the input operator overload function is quite similar to the output operator definition. In the driver function, we test drive the overloaded operators by accepting and redirecting values for/of the student object at one shot. When the compiler encounters a statement of the form `cout << stud`; in main the compiler generates the function call of the form operator `<< (cout,std)`, wherein the formal parameters o and `stud` become

aliases for cout and student objects. Both the operator functions return a reference to the respective stream to support cascaded input or output operations. Cascading of operators has been explained in the earlier chapter. Cascading of operators avoid duplication of the object names associated with the input or output operation. At end of an operator function call, the returned `ostream` or `istream` reference is used to process the remaining input or an output operation that needs to be completed. The definition of the input operator is similar to the output operator, except for the input `stream` reference replacing the `ostream` reference.

In this example, we overload the `stream` insertion and extraction operators to output or input student details of Name, Regno and Age at once. Name, Regno and age are private data members of the student's class. C++ allows insertion and extraction of character arrays at once. Thus, we will not go though each index of the character arrays to redirect the array contents or accept input for the array. Well, in this class we have purposely excluded the constructor and destructor definitions. It is left as an exercise for the reader and does not have any significant role to play in operator overloading. The two operators to be overloaded << and >> are defined as non-member friend functions for reasons explained already. Now elaborating on their signatures which are as follows:

```
friend ostream & operator << (ostream &, const stud &);
friend istream & operator >> (istream &, stud &);
```

The keyword `operator` denotes that it is an operator overloading function. The function names are << and >> respectively. The keyword `friend` indicates that the two functions (non-member) will be friends to the class student and thus have access to the class's private data members. Since << and >> are both binary operators and are operator-overloaded as non-member functions, we need to pass both the LHS and RHS arguments (operands) of the << and >> binary operands of the << and >> binary operators. In the stream insertion, the LHS operand is always an object of type class `ostream` and the RHS operand is always an object of user-defined type class student.

Thus, the two arguments passed to `friend` function operator << are of type `ostream` and student classes. They are passed by reference to avoid the duplicating or copying overhead associated with call by value and also references to the formal parameters of output and `stud` should actually refer to the original `cout` and user-defined object of class student. Observe that the output operation requires only read permissions over the student object passed by reference. Hence we make it a constant reference to enforce the principle of least privilege. The return type of the function is a reference to `ostream`. This is with purpose. We observe that the existing stream insertion and extraction operators support cascading. That is multiple redirections or indirection can be supported with a single `cout` or `cin` statement by cascading or chaining the << and >> operators as follows:

Let us assume `a` and `b` are of type integer. Thus, to redirect `a` and `b` at one shot using a single `cout` statement and cascading of <<operator, we issue the statement `cout << a << "\n" <<b`.

The cascaded indirection is quiet similar to the above redirection operation. To remember a fact, in our discussion on `this` pointers, we have mentioned that one if its main application is in supporting cascading of operators, the two operator non-member functions of our student class return references to `ostream` and `istream` precisely to support cascading. Within the respective definitions of the non-member functions << and >>, we return the formal reference parameters of output and input which is nothing but aliases for `count` and `cin`. Thus, when

a statement such as `cout << stud1 << stud2` is issued in the test driver function, assuming that `stud1` and `stud2` are two objects of user-defined class student, the compiler parses the above statement as operator `<< (cout,stud);` at the end of this non-member function call returns a reference to `ostream[cout alias o]`. Thus, the remaining statement is treated by the compiler as `cout<< stud2` which is again parsed as operator `<< (cout,stud2)`, at which point the returned `ostream` is captured by output buffer.

Cascading of >> (stream extract input) operator is quiet similar to the << operator except that the predefined class is `istream` and the function accepts references to class `istream` and student and returns references to `istream`. The reasoning behind two references arguments being passed and a reference being returned is exactly the same as that explained for `stream` insertion operator << overload. The definitions of the two non-member functions operator overload are quite simple. The `stream` insertion operator overload definition specifiers the logic for redirecting the three data members of the class student. Note that C++ allows `char` arrays to be redirected at one shot. The function after displaying each data member returns a reference to `ostream` variable (o — which is actually an alias for `cout`).

The stream extraction operator >> function defines the logic for accepting the values for three data members. At the end of its processing, the function returns a reference to `istream` (i) which is an alias for `cin` object. Note that the student object passed by reference to the operator >> function is non-constant. If this is because the function of >> would be to accept values from the keyboard and assign it to the respective data members which means modification or assignment operation will be performed over the object (student object) values will be read in from the keyboard and assigned to the `studentobject.name, studentobject.regno and studentobject.age`. Thus, write permissions are required over the object. Hence the student object reference that is passed to the operator >> non-member function is passed as non-constant reference argument.

Well, again since cascading of >> is supported with the existing meaning of >> operator, the overloaded version of >> should also support cascading. That is as we overloaded the << operator to support cascading or redirection of many student objects at one shot, so should the >> operator also support cascading or in other words details of more than one student object should be possible to read in using a single `cin` and cascaded or chain of >> operators. Thus the operator function operator >> is scheduled to return a reference to `istream` [the formal parameter being input which is in fact an alias for `cin`]. Within the definition of this non-member function, the logic for accepting values for individual data members is specified. Again we exploit the fact that C++ allows character arrays to be read in at one shot.

At the end of its processing, the function returns the `istream` reference [formal parameter input is returned] to enable cascading operation. Thus, the statement `cin >> stud1 >> stud2`; assuming `stud1` and `stud2` are objects of the user-defined class student, the compiler parses the above expression as operator `>> (cin,stud1)`; at the end of which a reference to `istream` (input—alias for `cin`) is returned. The remaining portion of the statement is then parsed as operator `>> (cin, stud2)`. The `main()` function serves as the driver function to test drive out student class. As is obvious the function creates student objects `stud1` and `stud2`. It then makes use of the programmer overloaded >> operator to read in values for data members of the objects `stud1` and `stud2`. Then we check if values were properly read in or not, we make use of the overloaded << operator to redirect the data members of `stud1` and `stud2` or state of `stud1` and `stud2` onto the screen. In the next section, we shall overload more complex operators.

6.6 Overloading an Array Class

In this section we will discuss on a programming example to overload a user-defined class `Array`. As it is the language already has a built-in array declaration supported. Thus, when we start out constructing a new array data type, there should be some additional features made available with this creation in comparison with the already available construct. To design our class in an efficient manner, let us first go though the bottlenecks associated with the existing array construct. The various problems associated with the array construct supported in the language are explained below.

Arrays can be walked off at either ends or what is often referred to as Array out of Bounds is a common problem with the built-in array construct. We will overload the `[]` array access operator to check for the out of bounds [exceeding the lower limit [index being `< 0`) and upper limit [index being > maximum size of the array] condition and then suitably perform the requisite action. Only character arrays can be input or output at once without having to overload the `<<` and `>>` operators. In our `IntArray` class, we shall overload the stream insertion and extraction operators to perform input and output of even integer arrays at once.

Arrays cannot be compared with equality of relational operators with when using the built-in array construct of the language. This comparison is not allowed with the existing construct since array names are nothing but pointers to the starting address of the array in memory or what is also referred to as base address of the array. In the `IntArray` class we create, we shall overload the `==` and `!=` operators to check if two Integer arrays are equal in their contents or not.

Also arrays cannot be assigned to another array identifier using an assignment statement with the built-in array constructs. We will overload the assignment operator to allow assignment of one integer array object to another integer array object. This assignment operation is not allowed with the built-in construct since array names are constant pointers and a constant pointer cannot appear on the LHS of an assignment operator.

Most often when arrays are passed as arguments to functions, their sizes are as well passed. But then with the integer `Array` class that we shall develop, manipulation of integer array objects via functions will not require the sizes of the array to be passed as an argument to functions that manipulate such objects. Instead the size of the array objects is made available as a data member and hence there is no requirement to pass the array size of functions that manipulate such array objects.

6.6.1 IntArray Class

The `IntArray` class has as its data member's size [size of the array], an integer pointer `[aptr]` that points to the starting or base address of the array. Since the programmer would like to keep a count of the total number of `IntArray` objects created at any point of time, we declared a static variable array count of type integer. It is declared as a static member, so that each time an object of class `IntArray` is created, the count variable is incremented by one. If this is declared as a normal data member that would be created on a per object basis, then the array count variable will be created separately for each object and thus the increment on such a member will not be a cumulative one, as each object has a separate copy of the count variable. Now let us concentrate on the various functions that this class is to support. First, let us complete the constructor for the class.

6.6.2 Constructor

The signature of the constructor is as follows:

```
IntAray (int =20);
```

The integer argument passed to the constructor is the size of the array. In case the user fails to specify that the array size at the point of object creation, then the array size is defaulted to 20. Thus, the above signature serves both as the default as well user-defined constructors. Definition of the user-defined constructor is as shown below.

```
IntArray(int sz)
{
size = sz;
aptr = new int [size];
++ arraycount;
for (int i=0;i<size; i++)
aptr[i]=0;
}
```

The passed size value (sz) is assigned or set to the size data member. Readers should note that the memory required to store the array elements of the required size for the array variable or pointer has not yet been allocated. Well, we wanted to create arrays of the required size dynamically at run time and not statically at compile time in which case the size of the array cannot exceed the specified maximum limit. And also as has been already mentioned statically allocated arrays may cause wastage of storage space the array size is always greater than the maximum value or limit specified in the source code. Thus, to avoid the above-mentioned bottlenecks, we decide to allocate space for the array dynamically as per the required size. Now having fixed or set the size of the data member, the next job of the constructor would be to allocate the memory for the required size. The statement `aptr = new int [size];` allocates memory of the required size and returns a pointer to `aptr` which is the base address of the array. Having allocated the memory of the required size, we then initialize the array contents to 0 using a `for loop` as follows:

```
for (int i=0;i<size; i++) aptr[i]=0;
```

6.6.3 Destructor

```
Signature is ~IntArray();
Definition:
~IntArray()
{
delete [ ] aptr;
};
```

The destructor deallocates the space allocated for the array using the statement `delete[ ] aptr`. Remember it is a collection or an array of spaces that is to be deallocated and hence the `[]` within the delete `aptr` statement is applied.

6.6.4 Assignment Operator Overload(=)

Now we will overload the assignment operator to assign array objects to one another. Note that both the LHS and RHS operands of the binary operators (=) are of the same user-defined (`IntArray`) type. Hence we will overload this operator as a member function. Since it will be overloaded as a member function, we will pass only one argument, the RHS operand (also of `IntArray` type) to this function. Since we require only read permissions over the RHS operand [array objects whose contents should be read and assigned to the LHS array objects], we shall pass it as a constant reference. We pass it by reference to avoid the duplication overhead, this reasoning remains valid for the other operators to be overloaded, as well. Thus the signature for the = operator.

Note that the existing = operator supports cascading, that is a statement with normal variables of the form `a = b = c` is executed right to left, that is c's contents are assigned to `b` which is then assigned to `a`. When this is the case with the normal meaning of the operator, based on the principle that operator overloading should be intuitive and similar to the existing meaning of the operator. That is cascading should be supported even in our overloaded version of =. That is multiple array objects assignment in a single statement should be possible.

For this, the function returns a reference of the array type. The reasoning behind this return type is the compiler would parse an overloaded = application in an expression of the form `o1=o2=o3`, where `o1,o2` and `o3` are assumed to be objects of class `IntArray`, right to left as follows:

First, the part `o2=o3` is parsed as `o2.=(o3)`. Then the remaining expression `o1. = (o2=o3)` would be parsed as `o1. operator = (o2)` if and only if the call `o2.=(o3)` returns an operand of the `IntArray` type. The returned reference is also made a constant reference to prevent multiple assignments in a cascaded application of the = operator. This is because parenthesizing the above expression can change the right to left normal evaluation order. By returning a constant reference (which will not be modifiable after only one assignment (initial)), we prevent manipulated multiple assignments such as `((o1=o2)=o3)`. Thus the signature of = operator is as follows:

```
const IntArray & operator = (const IntArray &);
Now let us go though the definition of the = operator function.
const IntArray & operator = (const IntArray & rt)
{
if (&right != this)
{
if(size !=rt.size)
{
delete [] aptr;
size = rt.size;
aptr = new int [size];
}
for (int i=0;i<size;i++)
aptr [i]= rt.aptr[i];
}
else
return * this;
}
```

Before performing assignment operation we need to check if we are trying to perform self-assignment. That is assigning the same object to itself which is meaningless and would drain system resources unnecessarily. This self-assignment test is performed by `(if (&right != this))`; `&right` returns the address of the passed or right-hand side argument and this actually refers to the address of the object or LHS operand since `this` pointer refers to the object with respect to which a member function was invoked most recently. Thus, the above statement checks if the address of the LHS and RHS operands or objects are the same or not. If and only if the objects differ in their addresses do we proceed with the assignment operation else we proceed to the next cascaded operation if any.

Now proceeding with the assignment logic, the first thing we need to check is if the size of the LHS array object is equal to the RHS array object size. For an array to be assigned to another array the first requirement would be that the size of the LHS array be the same as that of RHS array. If not the size of the LHS array should be resized to the size of the RHS array. This, is precisely what is performed by the inner if block. Now if the LHS array should be resized, then the memory allocations for LHS array should be resized. Thus, when the sizes of LHS and RHS array objects are not the same, we reclaim the space allocated for LHS array `[delete [ ] aptr];`, then allocates space for the resized array `[aptr = new int [size]]`. Now having resized (if required), we are left with the task of assigning RHS array values to the LHS array. This is performed with the following statements:

```
for (int i=0;i < size; i++)
aptr [ i ]= rt.aptr[i];
```

Finally, after having assigned (if this is not self-assignment) or skipped self-assignment operation, we return the `this` pointer or the address of the object with respect to which the member function is invoked. This return is to support cascading. Note that reference to size or `aptr` within a member function is actually the LHS array object's size or `aptr`. This is so because the compiler automatically associates the member function call with the operand (array object) on the LHS of the operator being overloaded.

6.6.5 Comparison Operator Overload(==)

Now we shall overload the == operator to compare if two array contents are equal to one another or not. Again this is overloaded as a member function for similar reasons explained in the earlier overload. Since it is only read permission over the RHS operand or object and read operations that will be performed within the comparison function, we make both the RHS operand as well as the function constant. C++ supports an additional data type called Boolean (keyword `bool`) to indicate the values of True or False. Hence the return type of the function is `bool` [Boolean]. The equality operator in its normal or existent meaning does not support cascading. Hence, our overloaded version of == will not support cascading and also the chance of such an application is extremely rare. Thus, the signature for overloading the == operator in our `IntArray` class `is:bool operator == (const IntArray &) const`.

Now getting on with the definition, arrays can be equal if and only if first place their sizes are the same. Thus, we check if the array objects on LHS and RHS of the == operator are same in size or not. If their sizes are different, there is no point in proceeding to check if their contents match or not. Thus we simply return FALSE and stop checking for array equality.

Now, if array sizes are equal, then we proceed to the logic of matching the array contents. Even if at one index the arrays differ in their values, we return FALSE and stop further equality checking. The function returns TRUE if and only if the values at all the array indexes [limit—user-defined at run time] are equal. The definition for the == operator is as follows:

```
bool operator == (const IntArray & r) const
{
if (size !  = r.size)
return FALSE;
for (int i=0;i<size;i++)
if (aptr[i] !=r.aptr[i])
return FALSE;
return TRUE;
}
```

6.6.6 Overloading the != Operator

Having overloaded the == operator, now we shall overload the != operator based on the principle that operator overloading should make use of already overloaded operators to the extent possible, we shall use the already overloaded == operator to overload the != operator. The result of a != operator will also be of Boolean type, with possible results of TRUE/FALSE. For better understanding let us assume a and b to be two integer variables with values of 4 and 4. Thus the test if (a ==b) returns true. Now, to apply the != check all we need to do is if (!(a==b)), in this case both a and b being equal the result of a==b will be TRUE and ! (TRUE) will be FALSE. In fact, the values are equal. Now, if a =4 and b =3 a==b would return FALSE and ! (FALSE) would return TRUE, to indicates that a and b are not equal to one another. Now, let us get back to our IntArray class overloading example. The signature of != operator is as follows: `bool operator != (const IntArray &) const;`

The definition is a simple one line statement as follows:

```
bool operator != (const IntArray &) const
{
return !  ( *this == right);
}
```

Yes, the single line return statement solves the task. The conditional check * this == right actually checks for equality of * this [LHS object] and RHS object. Note that * this actually dereferences the base address of the LHS array object, since this pointer refers to the address of the object for which the member function is executed off late. The remaining logic is quiet similar to as that was discussed for the simple case of a & b variable equality checking.

6.6.7 Subscription Operator Overload ([])

During overloading of array subscription operator [], the operator is overloaded to include the out of bounds checking. The operator is overloaded as a member function because the LHS operand is always an instance of the user-defined class IntArray. The main task of

the operator function is to test if the index used with the array is valid or not. That is not exceeding the maximum size of the array or the index being negative.

Thus, the operator [] function accepts the index or subscript as an argument. There is another issue to be addressed, the return type. Now, the indexing `[a[i]` can be both on the LHS or RHS of an assignment, that is `[I = a[i] or a[i] = c]` or one step further `a[i]=c[i]`. Thus, in one case the result (return type) of the operator should create an `lvalue` `[a[i]=c]` and in the other case should create an `rvalue [c=a[i]]`. `Lvalue` is the memory location or address where the array value `a[i]` is stored or will be stored. `Rvalue` is the content at a memory location or the value of `a[i]`. The first case where an `lvalue` should be created the signature looks as follows:

```
int & operator [ ] (int);.
```

The second case where an `rvalue` should be created the signature of [] looks as follows:

```
Const int & operator [ ] (int) const;
```

The `const int` reference returned ensures that only read [`rvalue` creation] and not write [`lvalue` creation] operation is allowed. The function is also made a constant function to allow constant array object call the function. The definitions of the above two operator overload functions are similar. We check for the index to be within the valid range or not. If yes then either the address or the value is returned. Else the index is forcibly set to the valid minimum or maximum index. Well, this is one solution to overcome out of bounds error. Another solution could be to terminate the program in execution by calling abort. The definition of the operator [] function is as follows:

```
int & operator [] (int sscript)
{
if (sscript >=0 && sscript < size)
return aptr[sscript];
else
abort();
}
const int operator [ ] (int sscript) const
{
if (sscript >=0 && sscript < size)
return aptr[sscript];
else
abort();
}
```

6.6.8 Input–Output Operator Overload (<<, >>)

Overloading the stream insertion or extraction operator to allow integer arrays to be input or output at one shot is quite simple. The signatures of the << and >> operators are as follows:

```
friend ostream & operator << (ostream &, const IntArray &);
friend istream & operator >> (istream &, IntArray &);
```

The earlier example was devoted for << and >> overloading. The reasoning behind both operators being overloaded as friend non-member function is similar to that in the previous example. Let us get on with the operator function definitions. The logic would be to traverse through each index of the array via a `for` loop and make use of the existing << and >> to accept or redirect one array value. The function definition is as follows:

```
ostream & operator << (ostream & o,const IntArray & a)
{
for (int i=0;i<a.size;i++)
o << a.aptr[i];
return o;
}
```

6.6.9 Copy Constructors

Well, in our earlier discussions on constructors we have seen two kinds of constructors, namely the default and user-defined constructors. In fact, there are two more constructors, which are called copy and conversion constructors. In this section, we shall provide details on these two constructor types. Well, in the previous example we overloaded the = assignment operator to allow assignment of objects to one another. With this = operator, the objects are assumed to be declared. But then if we require to copy the state of one object to another object at the point of creation itself, copy constructors serve the purpose. Copy constructors are thus used to copy one object's state into another object at the point of creation or instantiation. For example, the statement

```
IntArray o1 (8);
cin >> a;
IntArray o2 (o1);
```

In the above statements, we create an `IntArray` object of size 8 and then read in values using the overloaded >> operator. The next statement creates another instance o2 of IntArray and copies the state of `o1` onto `o2` using the syntax `o2(o1)`. Thus, when the compiler sees the expression `o2(o1)` it invokes the copy constructor of the class to copy or assign the state of `o1 to o2`. Obviously, in this case, the object that is being copied (destination) is of the same size as that of the source object `(o1)`. Again copy constructors can either be user (programmer-defined) or default (compiler-defined). Well, with default copy constructors there is no guarantee that the destination object is in a consistent state. Copy constructors accept as argument the source object and this has to be only by reference. The reason shall be explained in a short while. The signature for copy constructor of class `IntArray` is as follows:

```
IntArray (const IntArray &)
```

The definition of the above copy constructor is as follows:

```
IntArray (const IntArray & source)
{
size = source.size;
aptr = new int [size];
```

```
for (int i=0;i<size;i++)
aptr [i]= source.aptr[i];
}
```

The copy constructor definition is quiet similar to the = operator overload. First, we copy the size of the source object onto the destination object size data member. Next, the memory is allocated for the required size of array elements. Then, we copy the elements of the source object array onto the destination object array using a `for loop` as shown above. Well, the definition of the copy constructor is quiet similar to the = operator definition, however, there is a difference. Now we shall explain the reasoning behind copy constructors accepting arguments only by reference.

Copy constructors can be passed arguments only by reference and pass by value is not allowed. There is a subtle reasoning behind it. If arguments were allowed to be passed by value to copy constructors, then as per the pass by value mechanism a local or duplicate copy of the arguments should be created and it is over this copy that local updations are performed. But then there is a problem with copy constructors. Copy constructors (function) accept as argument the source object. Now, if it is to be a by value syntax, then a local copy of the source object should be created. The very purpose of copy constructors is to copy the state of the source object on the destination object. Now, if copy constructors were to accept arguments by value, then for a local or temporary copy of the object to be created, the copy constructor for the argument that is an object should be called. This is nothing but the copy constructor function getting called within itself (direct recursion) and this recursion is infinite or based on no condition. Thus, if copy constructors are allowed to accept arguments by value, then it leads to indefinite recursion. Hence, the reasoning behind it is not allowing the copy constructors to accept arguments by reference.

Another point to be noted with copy constructors is that the copy constructor should not simply copy the source object pointer onto the destination object pointer [address]. This may solve our purpose of copying the source object contents into destination object, since both the source as well as the destination object variable would just be an alias for the source object's address. This might simulate the copying operation, but this can lead to serious consequences. The consequence is that assuming that the destination object is the most recently constructed object in comparison with the source object, the destructor of the destination object that would reclaim the objects memory allocation in this case would delete the storage space associated with both the target and the source object. Thus, further references to the memory location via the source object will be undefined. This serious consequence is also referred to as dangling pointer situation n and can cause serious run time errors.

Note: If both the copy constructor and the overloaded assignment operator are provided in a class then based on the use either the copy constructor or the overloaded assignment operator is called. Thus in case where an object's content is to be copied to another object at the point of object creation using one of the following syntaxes mentioned below calls the copy constructor of the class. Copy constructor application syntax is as follows:

```
IntArray o2(o1) ; or Array o2 = o1;
```

After having declared objects of the class, then trying to copy or assign the state of one object to the other using an = operator leads a call to the overloaded assignment operator of the class. Objects can be prevented from being copied or assigned to one another. All

we need to do is to make the copy constructor or the overloaded assignment operator of the respective class private members. Doing so prevents such member functions being invoked from outside the class, even via objects of the class. Private members are not accessible from outside the class, even via objects of the class.The overall C++ code that implements the array class overloading, including the operator overload functions and user instantiations is shown in Figure 6.2.

```
"intarray.h"
class IntArray
{
friend ostream & operator << (ostream &, const IntArray &);
friend istream & operator >> (istream &, IntArray &);
public:
IntArray(int=20);
IntArray(const IntArray &);
~IntArray();
const IntArray & operator =(const IntArray &);
bool operator ==(const IntArray &)const;
bool operator '=(const IntArray &)const;
int & operator[](int);
const int & operator [](int) const;
private:
int *aptr;
int size;
};
"intarray.cpp"
#include"intarray.h"
#include<iostream.h>
IntArray::IntArray(int sz)
{
size = sz;
aptr = new int [size];
++ arraycount;
for (int i=0;i< size; i++)
aptr[i]=0;
}
IntArray::~IntArray()
{
delete [ ] aptr;
};
IntArray::IntArray (const IntArray & source)
{
size = source.size;
aptr = new int [size];
for (int i=0;i<size;i++)
aptr [i]= source.aptr[i];
}
const IntArray & IntArray::operator = (const IntArray & rt)
{
```

Figure 6.2: Continued

```
if (&right != this)
{
if(size !=rt.size)
{
delete [] aptr;
size = rt.size;
aptr = new int [size];
}
for (int i=0;i<size;i++)
aptr [i]= rt.aptr[i];
}
else
return * this;
}
bool IntArray::operator == (const IntArray & r) const
{
if (size ! = r.size)
return FALSE;
for (int i=0;i<size;i++)
if (aptr[i] !=r.aptr[i])
return FALSE;
return TRUE;
}
bool IntArray:operator != (const IntArray &) const
{
return ! ( *this == right);
}
ostream & operator << (ostream & o,const IntArray & a)
{
for (int i=0;i<a.size;i++)
o << a.aptr[i];
return o; }
istream & operator >> (istream & i,IntArray & a)
{
for (int i=0;i<a.size;i++)
i >> a.aptr[i];
return i; }
int & operator [ ] (int sscript)
{
if (sscript >=0 && sscript < size)
return aptr[sscript];
else
abort();
}
const int operator [ ] (int sscript) const
{
if (sscript >= 0 && sscript < size)
return aptr[sscript];
else
abort();
}
```

Figure 6.2: Continued

```
"main.cpp"
#include<iostream.h>
#include"intarray.h"
int main()
{
IntArray a1(6),a2,a3(6),a4(5);
//check the intial states of the array objects
cout<<a1<<"\n"<<a2<<"\n"<<a3<<"\n"<<a4<<"\n";
//read input values for the array objects
cin>>a1>>a2>>>a3>>a4;
//assign one array object to another
a3=a1;
cout<<a3;
//check if two arrays are equal or not
if (a1==a3)
cout<<"\n Array Objects are Equal"<<endl;
if(a1!=a4)
cout<<"\n Array Objects are Not Equal"<<endl;
//copy constructor to create copy a5 from a1
IntArray a5(a1);
//Array Indexing- can try out for out of bounds as well
cout<<"\n Array Indexing"<<a1[3]<<"\n;
return 0;
}
```

Figure 6.2: C++ code for `IntArray` class overloading.

6.7 Conversion Constructors

The next types of constructors are the conversion constructors. Conversion is an inherent requirement while programming. Even with structured programming languages, casting operations were supported to allow conversion of one data type to the other. At any or all stages of programming, we would require type conversion. With object-oriented languages, the conversion operation would involve objects and might involve conversion between user-defined and built-in data types. Such conversion need not be defined by the programmer, if the conversion operation involves only built-in data types.

This is because the compiler is very well aware of how to perform conversions among built-in types. In fact, conversion involving built-in types are often referred to as casting. This lenience is available only when conversion involves built-in types. With user-defined object conversion, the programmer must specify the conversion behaviour. Functions that perform user-specified conversions are referred to as conversion constructors. In fact to be specific, single argument constructors that convert objects [user-defined or built-in type] into objects of a particular class are specifically referred to as conversion constructors. The vice versa operation of converting an object of one class to object of other class or built-in type is referred to as conversion operator or the cast operator function. The signature of the cast operator function for creating a temporary int object from an user-defined object of class `X` is as follows:

```
X:: operator int () const;
```

This syntax converts a user-defined object of class x into built type. The signature to convert user-defined object of class x into another user-defined object of class y is as follows:

```
X:: operator y() const;
```

Well, they are made as constant functions since the conversion is only temporary operation. Thus, it is treated only as a read operation. If o1 is object of type X and the compiler encounters the statement (int) o1, the compiler generates the call o1.operator int (); [based on signature: X: operator int () const]. Thus conversion constructors or cast operators serve the purpose of converting user-defined to predefined or user-defined conversions.

6.8 Overloading of Unary ++/-- Operators

We have seen how to overload the +, = and other operators. In this list of over loadable operators, ++ and - are also present. That is the unary increment and decrement operators can be overloaded. Now, ++/-- can be used either with the pre or postversion. Yes, the issue involved with overloading of increment and decrement operators is in differentiating whether it is used with pre or postmeanings. Let us assume that o is an instance of a class that has 3 integer data members. Let us assume that o is incremented. Now, the compiler will parse o++ and ++o as o.operator ++(), assuming that ++ is overloaded as a member function of the class.

Now, the compiler will be in a fix to differentiate the two member function calls, since their signatures are the same and the compiler will not be able to resolve the member function call or the compiler simply will not know which among the two functions to call, since both the functions have the same name operator ++; same signature in other words. Yes, this is precisely the issue of differentiating function signatures involved with the overloading of ++ and - operator. Well this signature differentiation issue is encountered even when ++ or -- is overloaded as a non-member function. O++ or ++o, both lead to the call or are parsed by the compiler as operator ++ (o).

Thus, to differentiate their signatures, the postincrement version is passed a dummy integer. In the case of member function overload of ++, the statement ++o is parsed as o. operator++(), while the statement o++ is parsed as o. operator ++(0), where 0 is the dummy integer to differentiate the two function calls. In the case of non-member function overload the compiler parses o++ or ++o as operator ++(o,0) and operator ++(o). The 0 argument is used by the compiler to differentiate the signatures of the preincrement and postincrement versions. The dummy 0 need not be passed by the programmer; it is implicitly passed by the compiler.

The programmer just needs to specify the dummy argument type (int) in the member function or non-member function signatures and function header of the respective function definitions. The code for unary operator overloading is shown in Figure 6.3. The pre and postincrement operator functions make use of a helper or utility function to increment the values of the data members by 1. Note how the preincrement operator function first calls the helper function to increment the object state by 1 and then returns the object's address (via the this pointer). Thus, first the increment is performed and then future assignments are handled. In the case of postincrement operator function, the current state of the object is copied to a temporary object and then the increment on the data members is done, resulting in first the assignment operation (old state) and then the increment being performed. State

of object of class A can be reflected by overloading the output operator to redirect the various data members onto the screen to appreciate the effect of pre and postincrement operators, more so when done as a part of an assignment operation.

```
1. class A {
2. private :
3. increment();
4. int no1;
5. int no2;
6. int no3;
7. public:
8. A (int =0,int=0,int=0);
9. ~A();
10. A & operator ++();
11. A & operator ++(int);};
12. A (int n1,int n2,int n3){
13. no1=n1;
14. no2=n2;
15. no3=n3;}
16. A & operator ++(){
17. increment ();
18. return * this; }
19. A & operator ++(int){
20. A temp = * this;
21. increment ();
22. return temp; }
23. void increment (){
24. no1 = no1 + 1;
25. no2 = no2 + 1;
26. no3 = no3 + 1;}
27. int main () {
28. A o1(5,6,7);
29. o1++;
30. ++o1;
31. return 0;}
```

Figure 6.3: Unary operator overloading example.

Review Questions

1. Justify the need for operator overloading in C++.
2. List a few built-in operators in C++ that carry overloaded meaning.
3. List the do's and don'ts of operator overloading.
4. How do you decide whether an operator is to be overloaded as a member or non-member function?
5. Appreciate the use of friend functions in the context of operator overloading.
6. What do you mean by arity of an operator?

7. For the Time class package created in earlier chapter (exercise) overload the input and output operators to read and display time object values.
8. Develop a C++ Matrix package and overload operators of + and - to perform matrix addition, subtraction operations respectively.
9. Develop a C++ class called Complex and simulate complex number arithmetic of addition and subtraction using operator overloading.
10. Why should copy constructor of a class accept arguments only by reference? Justify.
11. Differentiate copy from conversion constructor in C++ using an example.
12. What is the issue to be addressed while overloading unary operators?
13. For the Matrix package created earlier, overload the pre- and post-increment and decrement operators respectively.
14. Discuss the overloaded meanings of << and >> operators in C++.
15. Appreciate the `cout` and `cin` statement by interpreting the way the compiler parses `cout << a` and `cin >> a`.

Chapter 7

Inheritance

In the last two chapters, we have discussed the key OO issues of Encapsulation, Abstraction and Operator Overloading. In this and the following chapter, we shall concentrate on two more key issues of inheritance and polymorphism. Well, the main advantage of OO language is the extent to which it supports reuse and inheritance plays a crucial role in supporting this feature as a part of the language set-up. Inheritance aims to make use of existing class declarations and definitions as a part of new software development wherever possible and thereby minimize the coding effort and time. Inheritance promotes the feature of resuability and is similar in concept to the syntax of standard library functions supported in languages such as C. Here, instead of function level resuability, class level reuse is supported. This chapter primarily concentrates on inheritance discussing the various issues involved, types of inheritance and their associated syntaxes in C++.

7.1 Reusability and Its Benefits

Software engineering observations over the years have been that program development time and code maintenance are lot more easier when software is created or built using existing code. In other words, reusability brings down coding effort or program development time, since programmers do not develop code from the scratch. Rather they make use of classes that are already available [with or without modifications] and their interfaces [functions] to solve the problem on hand [effectively coding]. Since codes that have been already created are used, the program or coding time is brought down (in fact drastically).

Another hidden advantage of reusability is that it facilitates easier maintenance. How? Codes that are reused or distributed have been tested and tried. The chances that such codes are susceptible to errors is very minimal. Thus, user-created programs, which reuse such code, is less-prone to errors or bugs and thus is lot more easier to maintain. Why on earth should we discuss reusability, all of a sudden? Well, inheritance is a key form of reusability. Inheritance in reality is that trait wherein entities inherit (derive/extract/imbibe) properties from other entities.

7.2 Parent–Child Relationship

A child inherits certain properties from the parent. Whether it is for the good or not, there is a separate class of diseases that are referred by medics as Hereditary. Biologists observe that a child imbibes almost 50% of its characteristics from its parent. All engineering or

for that matter any concept existing today is an imitation or an evolution of some natural phenomenon. In programming (OO) terminologies, inheritance is that concept wherein new classes are created from existing classes by extracting the features [attributes/data members] and behaviours [member functions] of existing classes.

Well, by rule of nature, the creator is referred to as the parent and the entity created by the parent is referred to as the child. Similarly, in OO language the existing class is commonly referred to as the parent or base class and the new class is referred to as the child or the derived class. To get into actual programming terminologies, inheritance helps the programmer in creating new classes by absorbing [inheriting] the data members and member functions of existing classes. Thus, the effort involved in the creation of the new class is brought down.

On the lighter side of it, again rule of nature is that the hierarchy does not stop with a child. As years go by the child becomes the parent. In programming terminologies a derived class can itself act as a parent or base class for another class. Nothing but to say that a derived class may be derived further. Programming requirements could be that a class derives or inherits its features from more than one class or there are more than one parent or base class for a derived class. Well, we do not get into correlation with rule of nature, at least here.

But of course, in terms of the properties or characteristics of a child, a grand parent also has some effect. In programming terminologies the case, when a class is created by inheriting features from a single class, is referred to as single inheritance. When a class derives its features from more than one class, it is referred to as multiple inheritance. In this chapter, we shall at length elaborate on both forms of inheritance. Thus, whether it is single or multiple inheritance, a derived class extracts features [data member or member functions] from the base class. Again by nature's law a child may have his or her own unique traits.

7.3 C++ Notion of Inheritance

Derived class if required can have additional data members or member functions, in addition to what it has inherited from the base class. This is to say that derived classes will be larger than that of base classes. Again nature's law, 'whether you like or not you inherit your parents' features'. To put it in simple terms, derived classes will always be of size greater than or equal to that of the base classes. The minimal size of the derived class is when it does not have any unique [additional features], i.e. data members or member functions. In terms of size, most probably derived classes will always be larger than base classes.

It is also common in programming circle to refer base class as superclass and derived class as subclass. Well, in terms of the hierarchy this sounds good. But in terms of the class size, it seems contradicting. It is in fact derived classes that are larger in size than base classes that should be termed as superclasses and base classes as subclasses. Well, we will justify the naming in a short while. The previous discussions on Data Encapsulation or Abstraction, among the three access specifiers 'protected' specifier ensures that such members are visible to derived classes/their objects. In C++ there are three kinds of inheritance associated with the access specifiers, namely private, protected and public inheritance.

In this section, we shall discuss these forms of inheritance. Irrespective of the inheritance type, the final accessibility is as well dependent on the base class member access specifier. With public inheritance, a public base class member becomes public in the derived class also.

Thus, it can be accessed by member functions, friend functions and non-member functions. A protected base class member becomes protected in derived class. Such a member can be accessed directly by member functions and friend functions. A private base class member becomes private in the derived class. The various possibilities are clearly explained in Table 7.1.

Table 7.1: Inheritance types/access specifier

Base class member access specifier	*Type of inheritance*		
	Public	*Protected*	*Private*
Public	Public in derived class (can be asscessed directly by non-static member functions, non-member functions and friend functions)	Protected in derived class (can be accessed directed by all non-static member functions and friend functions)	Private in derived class (can be accessed directly by all non-static member functions and friend functions)
Protected	Protected in derived class (can be accessed directly by all non-static member functions and friend functions)	Protected in derived class (can be accessed directed by all non-static member functions and friend functions)	Private in derived class (can be accessed directly by all non-static member functions and friend functions)
Private	Hidden in derived class (can be accessed by non-static member functions and friend functions but only via the public or protected member functions of the base class)	Hidden in derived class (can be accessed by non-static member functions and friend functions but only via the public or protected member functions of the base class)	Hidden in derived class (can be accessed by non-static member functions and friend functions but only via the public or protected member functions of the base class)

The syntax for specifying the three modes of inheritance discussed above is as follows:

```
Class DerivedClassName :  mode BaseClassName
{
derived class attributes (data members)
derived class properties (member functions)
}
```

For example if `A` is the base class and `B` is a class derived from `A`, and it is in the protected mode that `B` is derived from `A`, the syntax is as follows:

```
Class B : protected A
{
..
..
..
}
```

Inheritance can always be interpreted as (in the form of) a tree like hierarchical structure. The base class exists in a hierarchical relationship with derived classes. A good example for inheritance hierarchy could be: Let us assume the student entity in a university. A typical university or college has students of various types such as Research Student, Postgraduate Student and Graduate Student. Thus, these three types of students could be possible derivations from a base class called Student.

Private members of a base class cannot be accessed by derived classes. And also a derived class does not inherit friend functions of the base class. A base class's public members are accessible throughout the program. However, a base class's private members are accessible only by member functions and friends of the base class. A class's protected members can be accessed by members or friends of base and derived classes. Derived class members can refer by their names. An object of a derived class can be treated as an object of a base class object, but the vice versa treatment causes syntax error. This shall be explained in detail in the next section. Now let us look into a programming example for inheritance. We shall consider the student hierarchy in the following example, code for which is shown in Figure 7.1.

```
"student.h"
class Student
{
protected:
char * fname;
char * lname;
friend ostream & operator << (ostream &, const Student &);
public: Student (char * ="", char * ="");
~Student( );
};
"StudentFunctions.cpp"
#include <iostream.h>
#include "Student.h"
Student :: Student (char * first, char * last)
{
fname = new char [strlen(first) + 1];
strcpy(fname,first);
lname = new char [strlen(last) + 1];
strcpy(lname,last);
}
ostream & operator << (ostream & out, const Student & stud)
{
out << stud.fname <<"\t"<<stud.lname<<"\n";
return out;
}
"Graduate.h"
#include<iostream.h> #include "Student.h" class Graduate : public Student
{
friend ostream & operator << (ostream &, const Graduate &);
public :Graduate (char * = " ", char * = " ", char * =" ");
```

Figure 7.1: Continued

```
Protected :
Char * degree;
};
Graduate :: Graduate (char * fna, char *lna, char * dn) : Student(fna,lna)
{
degree = char new [strlen(dn)+1];
strcpy(degree,dn);
}
ostream & operator << (ostream & out, const Graduate & grad)
{
out << "First Name is \t" << static cast <Student>(grad);
out << "Degree is \t" << grad.degree;
return 0;
}
"main.cpp"
#include<iostream.h>
#include "Graduate.h"
#include "Student.h"
int main ()
{
Student * sptr = 0,S("Karthik","Krishnan");
Graduate *gptr =0,G("Anusha","Srinivasan","B.E");
cout << "Student S's details "<< S <<"\n";
cout << "G's details" <<G<<"\n";
//Trying to view a Graduate student as a Student
sptr = &G; cout << * gptr;
sptr = &S;
gptr = static cast<Graduate *>(sptr);
cout << * gptr;
return 0;
}
```

Figure 7.1: C++ code to illustrate the feature of inheritance.

7.3.1 Pointer Casting

In the above example, we have employed casting operators to convert base class pointer to derived class type and vice versa. These are referred to as casting operations. The earlier discussion on inheritance referred to base class as superclass and derived class as subclass. Even before we get into the above programming example, we shall dwelve on the casting operations. Well, all derived class objects can be treated as base class objects and it is syntactically and semantically valid to do so. It is on this basis that base class is referred to as superclass and derived class as subclass [based on the count of base class objects being more as a result of derived class objects being treated as be class objects].

The process of assigning derived class pointers to base class pointers is referred to as up-casting a pointer and the process of converting a base class pointer to a derived class pointer is referred to as down-casting a pointer. Down casting must be performed very carefully to ensure that the pointer points to an object of the correct type. Now getting back to the programming example, we consider the student hierarchy. We have a base class called Student whose members fname, lname are protected to enable access to derived classes of it. The constructor of the base class allocates memory dynamically for the fname, lname members

and then constructs the object suitably. The destructor deallocates the memory associated with fname, lname members. We have also overloaded the stream insertion (<<) operator to redirect a student's object at one shot.

The class Graduate which represents Graduate students has an explicit member, namely `char * degree` to identify the degree programme. The derived class constructor accepts arguments including the members that would be derived from the base class, namely fname and lname. Thus, the derived class constructor accepts all three arguments, two for the construction of derived class members inherited from the base class and one for its explicit data member. Within the definition of the derived class constructor, we invoke the base class constructor by its name and passing the respective arguments using the member initializer syntax : operator. There is a valid reasoning behind invoking the base class constructor within the derived class constructor.

Whenever derived class objects are created, even before the derived class's explicit data members are constructed, the base class members that are inherited in the derived class should be constructed first. It is with this reasoning that whenever derived class objects are created, automatically the base class constructor is called to construct the "from the base class" derived members of the derived class and then proceed with the construction of explicit derived class members. If the derived class constructor does not invoke the base class constructor using the member initializer syntax, the compiler automatically invokes the base class's default constructor whenever objects of the derived class are created.

A default base constructor not available in such cases causes syntax error. The overloaded stream insertion operator (<<) of the derived class makes use of the overloaded stream insertion operator of the base class student to redirect the "from the base class" derived members of the derived class. For the duration when we wish to treat the derived class objects as a base class object, which is syntactically and semantically valid, we use the `static _cast` [casting at compile time] operator to convert the derived class object (G) to the base class (Student) type.

After this temporary conversion the overloaded << operator of the class Student is invoked to redirect the "from the base class" derived members of the derived class. The derived class's explicit member (degree) is redirected separately (as an individual statement).

Note that the base class members are made protected to enable the derived class to refer them directly by their names. If the base class members were private instead of protected as we have here, then they cannot be accessed directly by the derived class, but can be referred to in the derived classes using the base class public member functions.

7.4 Class Graduate

Public Student instructs the compiler that the class Graduate is a publicly derived class from Student base class. The public and protected members of the base class are inherited as public and protected into the derived class Graduate. The drive routine creates `sptr` (student pointer) or pointer to student class and then initializes the address of student object S to this base class pointer. A similar derived class pointer and object pair is created. The overloaded output operator is applied with each instance of base and derived classes `S` and `G`. Well, this exercise is typically to understand the pointer manipulations of upcasting and downcasting efficiently. First, the base class pointer (`sptr`) is assigned the address of the derived class

object G, then redirection statement on line displays the student (base class) portion of the Graduate (derived Class).

Well, this pointer manipulation is allowed since a derived class object can always be treated as a base class object. The base class pointer (sptr) sees only the base class members of the derived class object. This casting or conversion is syntactically and semantically valid. Then the base class pointer is converted again to a derived class pointer type and this conversion is assigned to a derived class pointer.

```
[gptr = static_cast < Graduate *> (sptr)].
```

This is again valid since the derived class pointer is pointing to the address of a derived class object.

The validity is checked by dereferencing the contents pointed to by the derived class pointer

```
[cout << *gptr;]
```

Well, this casting of a base class pointer type to a derived class pointer type must be performed carefully, else may cause serious run-time errors. Such an operation is allowed syntactically but then semantically it is meaningless. In the above case, the derived class pointer is actually pointing to a derived class objects. Hence there were no run-time or serious problems to be considered.

However, if the objects type do not match, that is a derived class pointer assigned the address of a base class object without any casting, then referring to explicit derived class members [degree] via such a pointer might cause serious logical errors, since there is no such member existing at that point of program execution. This is illustrated in the above example by the cout statement at line. It displays only the base class portion of the derived class object and the explicit derived class member is defaulted to null.

It is up to the programmer to be careful while performing such manipulations, since calling or referring to data members or member functions non-existent can even crash the program in execution. In certain cases whatever values happens to be in the memory (location of the derived class pointer to the base class object). In all, such non-existent member references can prove very dangerous.

The reasoning behind not allowing private members of a base class directly accessible by derived classes is to prevent the data encapsulation being violated since such access might allow further derivations [classes derived from derived class] to access private data and would amount to data encapsulation being violated throughout the hierarchy. Well, you have a way out; go in for protected specifiers in such cases where the base class members should be directly accessible to derived class. The above example can be extended with another possible class Research Student, a derivation from Graduate class with its possible exclusive data member being fellowship amount. This is left as an exercise for the reader.

7.5 Order of Constructor/Destructor Calls with Inheritance

In the earlier discussions on encapsulation, we have seen one example to understand the order in which constructors and destructors get invoked. Well, in this section, we would like to extend those concepts and understand the constructor/destructor calls in an inheritance (hierarchy)

involved program scenario. All those concepts explained in the earlier section are valid here as well. The only new issue here would be to resolve the ordering of base class constructors and destructor calls when a derived class object is created or instantiated.

When a derived class object is created, since a derived class inherits its base class members first and then has its own or explicit derived class members, as we already mentioned it is first base class constructor(s) that should be called and so it is and then the derived class constructor [to construct the explicit derived class members]. As is the principle that most recently constructed objects are destructed first, the destructor for the derived class object and then the base class object is invoked.

The base class constructors and assignment operators are not inherited by derived class, but still derived classes can call constructors and overloaded assignment operators of base classes. If the derived class constructor is not defined by the programmer, when a derived class object is created, the compiler which invokes the default constructor of derived class in turn invokes the base class's default constructor even before the construction of the derived class's explicit data members.

In the case of multiple inheritance when a class derives its properties from more than one base class, the order in which the base class constructors are called depends on the inheritance order specification and not on the order in which the base class constructors are specified in the member initializer syntax of the derived class constructor. That is, let us assume a class `C` is publicly derived from both class `A` and `B`, whose syntax is as follows:

```
Class C: public class A, public class B
{
;
;
}
```

Let us assume that the derived class constructor definition [with no emphasis on the number of arguments] looks like `C() : B(),A()`.

When an object of class C is created it is the ordering specified in the inheritance specification `[A, B]` that determines the order of the base constructors call and not the ordering of the member initializer syntax. That is to say that first the constructor of the base class `A` is called and then the constructor for the base class `B` is called, and not in the order of `B`'s constructor followed by `A`'s as is specified in the member initializer syntax. Whatever be the case, destructors always get invoked in the reverse order of their construction.

7.6 Inheritance and Composition Issues

Another issue to be resolved with inheritance-based OO programming is that a base class or derived class can have objects of other class as its members or a base or derived class can compose objects of other classes as its members. This feature, which is referred to as composition is differentiated from inheritance by the fact that composition signifies a 'has a' relationship, while inheritance signifies a 'is a' relationship. This naming is based on the fact that in composition an enclosing class composes or has objects of other class as its members. In inheritance, a derived class object is an instance of or can be treated as a base class object.

When composition and inheritance are exploited together in an OO program, that is both base class and derived class contain objects of other classes, when an object of a derived class is created, the constructor for the base class member objects (base class constructor), and then the constructors of derived class member objects (derived class constructors) are invoked.

Member objects are constructed again based on the order in which they are declared in the enclosing or composing class and not on the ordering in which the member initializer syntax of the composing class's constructor specification. Destructors are called in the reverse order of constructor calls. Getting back to our earlier programming example on student hierarchy, let us assume we have the following driver function to drive the classes Student and Graduate, code for which is shown in Figure 7.2.

```
#include "Student.h"
#include "Graduate.h"
int main ( )
{
Student S1("Sai","Siddarth");
Graduate G1("Kripa","Shankar","B.S");
{
Student S2("Shri","Vastsan");
}
Graduate G2("Ganesh","Ramani","B.E");
return 0;
}
```

Figure 7.2: Order of constructor/destructor calls with inheritance.

The above statements shown in Table 7.2 clearly indicate the order in which base class and derived class constructors/destructors are invoked when a derived class object is created.

Table 7.2: Output for code in Figure 7.2

The order of constructor and destructor calls in the above example is as follows:
Base Class Constructor for S1 [Sai Siddarth]
Base Class constructor for G1 [Kripa Shankar]
Derived Class constructor for G1 [BS]
Base Class Constructor for S2 [Shri Vatsan]
Derived Class Destructor for S2 [Shri Vatsan]
Base Class Constructor for G2 [Ganesh Ramani]
Derived Class Constructor for G2 [B.E]
Derived Class Destructor for G2[B.E]
Base Class Destructor for G2[Ganesh Ramani]
Derived Class Destructor for G1[BS]
Base Class Destructor for G1[Kripa Shankar]
Base Class Destructor for S1[Sai Siddarth]

7.7 Base Class Members Redefinition

A derived class inherits both data members as well as member functions from the base class. The derived class can have its own explicit data members or member functions. In certain occasions, the derived class may decide to redefine member functions that have been inherited by the derived class. That is member functions in the derived and base class have the same name or same function identifier.

The derived class may or may not redefine member functions that are inherited from the base class. This concept of redefining member functions of the base class in the derived class is referred to as overriding of base class member functions in the derived class. Function overriding is a key concept in inheritance. We shall explore function overriding in the following sections. The first simple question to be resolved is how does function overriding differ from function overloading.

Well, in function overloading, the functions have to necessarily differ in their signatures. It is based on the signature that an overloaded function call gets resolved. In the case of function overriding, there is no such syntactic requirement that the base class member function and its derived class version (overridden function) differ in their signatures. In fact, most probably the signatures of the base class and the overriding derived class version have the same signature. The correct function gets called based on the object type [base class or derived class].

When an overridden function is mentioned by name with a derived class object, it is the derived class version that is invoked automatically. If one requires the base class version of the member function within the derived class, the scope resolution operator preceded by the base class name and followed by the member function name is used to access the base class version from the derived class. Normally in function overriding, the derived class version of an overridden function makes use of the base class version to achieve the tasks related to the "from the base class" derived members of the derived class.

It is only the additional tasks required for the explicit derived class members that a new code is specified. This prevents code redundancy and promotes easier code maintenance. Our earlier Student–Graduate example will now be modified by excluding the overloaded stream insertion (`<<`) operator. Instead, we will define a function called `display()` in the base class student [that displays the first name and last name] and the `display()` function is overridden in the derived class.

The overridden version invokes the base class version of `display()` to redirect the `fname`, `lname` [base class members] of the derived class object [Graduate]. The overridden version in the derived class redirects the explicit data member, namely degree. The modified member function definitions are as shown in Figure 7.3.

The class structures are quite similar to the earlier discussion on student hierarchy. Base class member function display redirects the first name and last name of the Student object. The derived class Graduate overrides the display function inherited from the Student class. The overridden version invokes the base class version to redirect the first name and last name members inherited from the Student class and then redirects the explicit derived class member (degree) using a separate `cout` statement.

The driver function test drives the base and derived classes and invokes the display with respect to each instance to verify that the correct version gets invoked. The reasoning behind allowing derived class objects to be treated as base class objects is that derived class has members corresponding to or required for the base class members. However, base class objects

```
"Student.h"
Class Student
{
protected :
char * fname;
char * lname;
public : Student (char * ="", char * ="");
~Student ( );
void display ( ) const;
};
"Student.cpp"
#include <iostream.h>
#include "Student.h"
Student :: Student (char * fn, char * ln)
{
fname = new char [strlen (fn) + 1];
strcpy(fname,fn);
lname = new char [strlen (ln) + 1];
strcpy(lname,ln);
}
Student :: ~Student ( )
{
delete [] fname;
delete [] lname;
}
void Student :: display( ) const
{
cout <<"First name is \t" << Q fname <<"\n";
cout <<"Last name is \t" << lname <<"\n";
}
"Graduate.h"
#include "Student.h"
Class Graduate : public Student
{
protected :
char * degree;
Graduate ( char * = , char * = , char * = );
~Graduate ( );
void display ( ) const;
};
Graduate :: Graduate (char * fnam, char *lnam, char * deg)
: Student(fnam,lnam)
{
degree = new char [strlen(deg) + 1];
strcpy(degree,deg);
}
Graduate :: ~Graduate ( )
{
delete [] degree;
```

Figure 7.3: Continued

```
}
void Graduate :: display ( ) const
{
Student :: display ( );
cout << "Degree is \t" << degree <<"\n";
}
"main.cpp"
#include<iostream.h>
#include"Graduate.h"
#include"Student.h"
int main ( )
{
Student S1("Siddarth", "Jonnathan");
Graduate G1("Sudharshan", "Rangarajan", "M.Tech");
S1.display ( ); // Base class Version display
G1.display ( ); // Derived class version of display
return 0;
}
```

Output:

First Name is Siddarth
Last Name is Jonnathan
First Name is Sudarshan
Last Name is Rangarajan
Degree is M.Tech

Figure 7.3: C++ code for student class based on inheritance.

cannot be treated as derived class objects since the explicit derived class members would then be undefined. In simple terms, base class object = derived class object is allowed (syntactically and semantically valid), but derived class object = base class object is not allowed (syntactically valid, but not semantically!).

The reason is that in the second assignment the additional explicit derived class members will not have valid values. The earlier assignment statement is allowed since the derived class object has the required base class members for the assignment operation to proceed. In other words, in the first case the derived class object on the RHS of the assignment statement should be compatible or of similar type to the LHS base class object. Thus, this case wherein a derived class object is to be converted or treated as a base class object is allowed while the vice versa operation is dangerous, though it may be syntactically allowed, but definitely dangerous from a semantic perspective. Such manipulations require intermediate casting operations as well. In the following section, we shall elaborate on multiple inheritance and focus on virtual base class, to resolve one of the key issues involved in multiple inheritance.

7.8 Multiple Inheritance

It is a hierarchy set-up in which a class derives its properties from more than one base class. Well, the issues pertaining to the constructor/destructor calls has already been explained. Let us consider the following inheritance hierarchy where the class `C` is derived from class `A` and class `B`. Let us go through the definitions of these classes which are given in Figure 7.4.

```
"Example.cpp"
#include <iostream.h>
Class A
{
public :
A (int a)
{
val = a;
}
~A( )
{
cout <<"Destructor \n";
}
protected :
int val;
void print ( ) const
{
cout <<"Value is"<<val<<" \n";
};
Class B
{
public:
B (char c)
{
id = c;
}
~B( )
{
cout << "Destructor \n";
}
protected:
char id;
void print ( ) const
{
cout <<"Value is"<< id <<"\n";
};
class C : public A, public B
{
public :
C(int=0, char='',double=0.0);
~C ( );
friend ostream & operator << (ostream &, const C &);
private : double iden;
};
C :: C( int i, char j, double k) : A(i), B(j)
{
iden = k;
}
C :: ~ C( )
{
```

Figure 7.4: Continued

```
cout <<"Destructor of C \n";
}
ostream & operator <<( ostream & out, const C & obj)
{
out << "Integer value is "<< obj.val <<"\n";
out << "Character value is"<< obj.id <<"\n";
out << "Double value is "<< obj.iden <<"\n";
return out;
}
int main ( )
{
A o1(50), * aptr = 0;
B o2('B'), * bptr = 0;
C o3(10,'C',1.1);
o1.print();
o2.print();
cout << o3;
aptr = &o3;
aptr→print();
bptr = &o3;
bptr→print();
return 0;
}
```

Output:

```
Value is 50
Value is B
Integer Value is 10
Character Value is C
Double Value is 1.1
Value is 10
Value is C
```

Figure 7.4: C++ code for multiple inheritance.

In the case of multiple inheritance a derived class object can be treated as an object of either of the base classes as has been done above. Class `A` is identified by an integer identifier, `B` is identified by a character identifier, and `C` which inherits from `A` and `B` has its explicit member of a double identifier. Corresponding constructors and destructors are provided to suitably initialize the data members. Note that the constructor of derived class `C` accepts arguments inclusive of the base class(es) `A` and `B`. This being an example to understand the concept of multiple inheritance not much member functions are required here.

The `print` function of base classes `A` and `B` redirect their respective data members. Note that we have overloaded the output operator for the derived class `C`. However, an alternative solution would be to override the `print` function of class `C` within which the respective base class versions be invoked to redirect the inherited base class members. We will defer such programming at least till we resolve the need for virtual base classes, the topic of interest in our next section. The `driver` function creates normal instances of each of the three classes `(o1,o2,o3)` and two pointer instances, one `(aptr)` to base class `A` and the other `(bptr)` to base class `B`.

To understand the concept that a derived class object (inherited from more than one base class) can be treated as an instance of either of the classes, the driver function assigns the address of o3 to aptr trying to treat derived class object o3 as an instance of base class A (line). Then bptr is assigned the address of o3 to treat the derived class object as an instance of base class B. In each of the above cases, the print function is invoked to display the object identifier. aptr→print() displays the value 10 (integer identifier of 03), a property inherited from base class A. bptr→print() displays the value C (character identifier of o3), a property inherited from base class B.

As should be clear member function invokes of print() on each of the two normal instances of the base class o1 and o2 displays the state of o1 and o2. State of o3 is displayed making use of the overloaded redirection operator. This is our normal syntax of invoking a member function on a object is in no way different in interpretation with respect to inheritance. Well, with the above discussion, readers should be clear of the fact a derived class object that inherits properties from more than one base class, can be treated as an instance of either of the base classes. Having explored multiple inheritance in terms of feature inheritance, we are now in a position to focus on the issue of virtual base class, one of the key aspects of multiple inheritance scenario.

7.9 Virtual Base Class

The virtual base class declaration is allowed to overcome one of the key duplication issues involved with multiple inheritance. We shall go through the following example to understand the duplication issue involved with multiple inheritance. Let us assume we have the following class structure shown in Table 7.3 available. Well, in this section, we are more interested in understanding the need for virtual base class, a concept and hence we will not be particular about the syntax behind each of these declarations, which should be clear by now and can be very easily incorporated.

Table 7.3: Virtual inheritance situation

Class A → Data Member I
Class B (derived from A) →Explicit member J
Class C (derived from A)→Explicit member K
Class D (derived from B,C)→Explicit member L

The information within the parentheses clearly identifies the parent class in each of the derivations and the information on the right of the arrow identifies the respective derived class's explicit member. Well, what is the issue at all? That should be the intriguing question disturbing all readers who have understand the previous discussions on inheritance and multiple inheritance. In fact, the issue should have become clear by now! But, still let us for a while interpret the structure of each of the above classes in terms of data members.

Class A is existing as a normal class (concrete class) and has one data member named I. Class B, a derivation from Class A has two data members, one its explicit data member (J) and the other being the inherited I from class A. Class C, also a derivation from class A has two data members, one its explicit data member (K) and the other being the inherited I from class A. The issue with multiple inheritance is as follows. We have a class D, that is a derivation from two classes, namely B and C. D in all will have five data members, one

its explicit data member (L), two inherited from class B(I and J), the last two inherited from class C (I and K). Now the issue should be clear. D in terms of the final class structure has two data members named I and references to I via an instance of D would be irresolvable or the compiler is in a fix to differentiate which I it is. Yes, virtual base class is the solution for such inheritance scenarios. To prevent such ambiguous references, the virtual keyword is used to declare what are called as virtual base classes. In this case, classes which will serve as base classes for further (possibly) multiple inheritance situation are derived in the virtual mode. The virtual keyword precedes the public, protected or private mode of derivation specification. That is, in the above programming example, the derivations should be as shown in Figure 7.5 (of course to avoid the duplication effect). The dots indicate the data member(s)/member function(s) that can possibly belong to each of the above classes. Thus, virtual base class overcomes the ambiguous (duplicated members) scenario by maintaining only one copy of members in the derived class that inherits properties from more than one base class, each of which is a derivation from another common base class.

```
Class B: virtual public A
{
...
}
Class C :virtual public A
{
...
}
Class D: public B, public A
{
...
}
```

Figure 7.5: Abstract base class declaration.

Review Questions

1. Discuss the benefits offered by inheritance feature of C++.
2. Define the terms base and derived classes.
3. Appreciate the access specifiers of members in classes derived from a base class.
4. Differentiate single from multiple inheritance. What is the issue involved with multiple inheritance?
5. Appreciate the usage of casting operations with inheritance programming situations.
6. Develop a university package in C++ with options to display details of Student, Faculty and Staff. A few of the other operations to be supported include display of CGPA for Students, display of salary details for Faculty and Staff. Use inheritance feature of C++.
7. Differentiate inheritance from class composition.
8. Differentiate an abstract base from a virtual base class.
9. Appreciate the benefits of inheritance in terms of long-term software development.
10. Discuss constructors and destructors with respect to derived classes and highlight the order of construction and destruction with a programming example.

CHAPTER 8

Virtual Functions and Polymorphism

Virtual Functions and Polymorphism help in designing and implementing systems that are more extensible. Polymorphism helps in treating objects of all classes of a hierarchy generically as base class objects. Virtual Functions, the syntactic construct to implement polymorphism provides a means or way of dealing with objects of different types in a program. An alternative solution to virtual functions is switch statement to take the appropriate action based on the object type. Each object type correlates with the case labels of the switch statement. The bottlenecks associated with the switch logic, explicit testing of object types is required. An accidental omission of one of the cases would lead to one object type being not recognized.

Another disadvantage is that if a new class or object is added to the hierarchy, then correspondingly an equivalent case must be inserted in the switch logic to handle the new class's objects. Thus, addition or deletion of classes to handle new types requires the switch statements to be modified, which would be time-consuming or error-prone. Virtual functions, which are the syntactic construct to implement polymorphism, eliminate the need for switch logic. Virtual functions in comparison with switch logic contain less branching and thereby facilitating testing, debugging, program maintenance, and bug avoidance.

8.1 Polymorphism

Polymorphism is thus the ability of objects of different classes related by inheritance to respond differently to the same message or member function call. The same message (in terms of signatures) sent to many different types of objects thus takes many forms. Polymorphism, the dictionary meaning is an entity that can exist in multiple forms. In polymorphism, when a request is made through a base class pointer, the correct overridden function in the appropriate derived class is chosen.

To differentiate a polymorphic from non-polymorphic behaviour, let us assume that a non-virtual member function is defined in a base class and the same function is overridden in the derived class also. If such a member function is called through a base class pointer to the base class object, always the base class version is called. If the member function is referenced through a derived class pointer, the derived class version is used. This in other words, is non-polymorphic behaviour. This is also referred to as static or compile time binding wherein all member function calls are associated with the objects at compile time itself. Such member function calls are resolved statically.

Thus, with non-polymorphic behaviour it is the pointer type and not the actual object type to which it points to that decides the member function call. This is the reasoning behind a member function reference through a base class pointer to derived class object always invoking the base class version, when in fact the object to which it points to (derived class) member function should be invoked. Let us consider the following programming example:

```
Student S, * sptr = &S;
Graduate G * gptr = &G;
sptr→display ( ); // invokes Base class version
gptr→display ( ); // invokes derived class version
sptr = &G;
sptr→display ( ); // invokes base class version
```

In the above example, we consider the same student hierarchy explored in the earlier chapter. Student is the base class and Graduate is a class derived from Student. Both the base and derived classes have their own versions of function `display()`. Assuming `display()` is not a virtual function, referencing display through a base class pointer `[sptr]` always calls the base class version. The calls `sptr→display()` and `gptr→display()` call the intended base class and derived class versions [member function call is decided by the pointer type and not by the actual object type to which it points to].

The call `sptr→display()` after having assigned the address of a derived class object to `sptr` in the previous statement invokes the base class version of display. Logically speaking `sptr` is pointing to an object of derived class type and the member function reference `sptr→display()` should actually invoke the derived class version of `display()`. For such a polymorphic behaviour to be supported it requires dynamic binding or binding function calls at run-time.

Well, virtual functions are the syntactic provisions in C++ to implement polymorphic behaviour or run-time binding of function calls. That is functions that need to support polymorphic behaviour should be explicitly declared by the programmer as virtual. However, to invoke a derived class version through a base class pointer pointing to a derived class object [function being non virtual]. This is just an alternative or a way out in cases where virtual functions are not supported. For example:

```
sptr→Graduate::  display();
```

or

```
gptr→Student:d:  display ();
```

Through the use of virtual functions member function calls can be made to cause different actions depending on the object type receiving the respective function call.

8.2 Virtual Functions

To support polymorphism or to treat al objects related by inheritance as objects of the base class and there by let the program determine dynamically at run-time which version of derived class to be used, we declare functions as virtual. To enable this kind of polymorphic behaviour, the member function that needs to support polymorphism is declared virtual in the base class.

For example, with respect to the student inheritance hierarchy the `display()` functions should be declared as virtual in the base class. The function prototype or header (only) is preceded by the keyword virtual to make the respective function virtual and thereby support polymorphic behaviour. The exact syntax to declare a virtual function is as follows:

```
virtual void display();
```

There is no need for the virtual keyword to be repeated again in the function header of the function definition. The above function `display()` can very well be made a constant function as well, since it requires only read and not write permissions.

A function that has been declared as virtual in the base class remains virtual throughout the inheritance hierarchy, even though it is not declared virtual in a class that overrides it. Thus, a function that has been declared virtual in the base class (explicitly) is implicitly treated as a virtual function in all its derived classes as well. Though functions are implicitly declared as virtual in the successive derived classes that override, to promote program clarity it is a good programming practice that functions be explicitly declared as virtual even in classes that overrides it. And also when a derived class decides not to override or redefine a virtual function, the derived class inherits its immediate base class's virtual function version.

When a virtual function is invoked by referencing an object by name and the . operator, the call is resolved at compile time and the virtual function that is called is the one inherited or overridden by the class of that particular object. They dynamic or polymorphic behaviour is supported only when a base class pointer referring to a derived class object references a virtual function overridden or inherited in one of the derived classes.

8.3 Abstract Classes

All along classes that were defined were instantiated at some stage or the other of an OO program. Such classes that are instantiated or objects or instances of which are created is referred to as concrete classes. But in certain cases, classes may exist solely as a part of an inheritance hierarchy and no objects or instances of such a class intend to be created. This class is purely existing for the purpose of inheritance to distribute its members to classes that shall be derived from it. Such classes are called abstract classes and no instances or objects of an abstract class can be created. In fact, they are also referred to as Abstract Base Class, since they are existing only as a base class [for future derivations]. In fact, abstract classes are generic while concrete (normal) classes are specific in their existence.

Abstract classes are normally at the top of an inheritance hierarchy. As and when we get deeper and deeper into the hierarchy, it is in fact derived classes that are more specific instances of the base class. A class can be declared to be an abstract class by declaring one or more of its member functions to be virtual and not just virtual, but pure virtual! A pure virtual function is one such function that is initialized to 0 in its declaration. An example of pure virtual function declaration is virtual void `display () =0`. For a pure virtual function, there should be no definition in the base class. There should be at least one pure virtual function in a class to be treated as an abstract class.

A class that is derived from an abstract class [has at least one pure virtual function] and the derived class does not provide a definition for that pure virtual function (s) in it, then the function becomes pure virtual even in the derived class and thus consequently the derived

class also becomes an abstract class and cannot be instantiated. It can only be used to derive further classes from it. Note that attempts to create objects or instantiate an abstract class causes syntax error.

Thus, virtual functions and polymorphism help the programmer in implementing generic behaviour and let the specifics to be decided at run-time. New types of objects that will respond to existing message can be added without modifying the base system. Abstract class defines interface for the various members in a class hierarchy. The pure virtual functions in an abstract class are defined in its derived class. All objects in the inheritance hierarchy can make use of the same interface through polymorphism.

Though abstract classes cannot be instantiated, pointers and references to abstract classes can be created and thereby support polymorphic behaviour/manipulations of derived class objects. Polymorphism is most often used for layered software systems. In operating systems development, physical devices operate differently from others. Regardless of the type of the physical device, read or write commands from and to devices have certain uniformity. However, each message needs to be interpreted specifically in the context of the device deriver.

An OO OS might use an abstract base class to provide an interface appropriate to all device drivers. Through inheritance, derived classes are created and all these classes operate uniformly. These capabilities (interfaces) offered by the device drivers are provided as pure virtual functions in the abstract base class. Implementations of virtual functions are provided in the derived classes that correspond to the specific type of device drivers. Having had sufficient theoretical discussion behind virtual functions, let us now explore a programming example to understand the syntax and functioning of virtual functions. We consider the employee set-up in this example, code for which is as shown in Figure 8.1.

```
"employee.h"
class Employee
{
public:
Employee (const char*,const char*);
~ Employee();
virtual double salary()const =0;
virtual void display() const;
private: char *firstname;
char * lastname;
};
"functions.cpp"
#include"employee.h"
Employee::Employee(const char *fn,const char*ln)
{firstname=new char[strlen(fn)+1];
strcpy(firstname,fn);
name=new char[strlen(ln)+1];
strcpy(firstname,fn);
}
Employee::~Employee
```

Figure 8.1: Continued

```
{delete [] firstname;
delete [] lastname;
}void Employee::display()const
{
cout<<firstname<<lastname;
}
"foreman.h"
#include"employee.h"
class Employee::public Foreman
{
public:
Foreman(const char*,const char*,double);
private: double pay;
};
"Foreman.cpp"
#include"foreman.h"
Foreman::Foreman(const char*f,const char *l,double p)
::Employee(f,l)
{
pay=p;
}
double Foreman::salary()const
{
cout<<"Employee's Pay is "<<pay;
}
"Manager.h"
#include"employee.h"
class Employee::public Manager
{
public:
Manager(const char*,const char*,double,double);
private:
double basic;
double perks;
};
"Manager.cpp"
#include"Manager.h"
Manager::Manager(const char*fn,const char *ln,double bp,double pk)
::Employee(fn,ln)
{
basic=bp;
perks=pk;
}double Manager::salary()const
{
cout<<"Employee's Pay is "<<basic+perks;
}
"main.cpp"
#include<iostream.h>
#include"employee.h"
```

Figure 8.1: Continued

```
#include"foreman.h"
#include"Manager.h"
int main()
{
Foreman f("Shankar","Kumar",4587.00);
Manager m("Ashok","Singhal",6500.85,3000);
Employee *ep;
f.display();
m.display();
ep=&f;
ep->display();
ep=&m;
ep->display();
return 0;
}
```

Figure 8.1: Virtual functions, C++ code.

8.4 Code Interpretation

In the above example in Figure 8.1, the base class employee (abstract class) contains data members' first and last name. Member function display that is expected to display the first name and last name details of an employee object is made virtual while salary is a pure virtual function. That is to say that future derivations from Employee class may or may not override the behaviour of display function (depending on their requirement) but every future derived class must necessarily redefine or override the salary function. Thus the salary computation logic for the varied employees is assumed to be unique and different from one another. Note the driver routine creates two instances of class Foreman and Manager and displays the salary details. A pointer to the base class (ep) is assigned the address of the either derived class objects (f and m), and because the member functions have been declared to be virtual, the appropriate functions (in the order of Foreman's followed by that of Manager's) are called.

Polymorphism and virtual functions are helpful to the programmer when all possible classes are not known in advance. New classes are accommodated by dynamic binding (late binding) or binding at run-time. An object's type need not be known at compile time for a virtual function call to be compiled. At run-time the virtual function call is matched with the appropriate member function of the called object. Dynamic binding also helps in maintaining code secrecy when software is distributed to end users. Such distributions contain only header and objects files and not the implementation. Source code (of member functions) are not revealed. End users use inheritance to derive classes from those that are distributed.

Another important issue to be resolved with polymorphism is related to destructors. If an object is destroyed by explicitly applying the delete operator to a base class pointer to the derived class object, the base class destructor is called, as a result of matching by pointer type. Always the base class destructor is called irrespective of the type of the object to which the base class pointer actually points. This happens even though the derived class's destructor differs in its name or function identifier. Well, this issue is overcome by declaring the destructor of the base class to be virtual.

With such a declaration in place, automatically the derived class's destructor also becomes virtual, even though it differs in its name. With such virtual base class destructors, on deleting a base class pointer to a derived class object, the destructor for the appropriate class derived is called. Well, after destroying the derived class explicit members, automatically the base class constructor is called to destroy the base class portion of the derived class object.

The reason why this issue has been resolved is that otherwise the base class destructor would destruct the base class portion of the derived class object, leaving the derived class object's explicit members undestroyed and also the principle that "destructions are in the reverse order of construction" gets violated since the base class destructor gets called before the derived class destructor, (even if we were to assume that the derived class destructor gets called somehow). Hence, the need for virtual destructors to ensure proper and as per principle destruction.

Well, the next issue that should strike or disturb any oo Programmer is whether constructors should be made virtual and of course the next issue can constructors be made virtual. The answer is that constructors cannot be made virtual; doing so leads to syntax errors. Again there is reasoning behind this syntax. Constructors are required at the point of object creation for initialization of data members to valid state. Well, dynamic binding or polymorphism is in terms of behaviours associated with objects that have been already created. Hence constructors cannot be made virtual and there is no need for them to be virtual as well.

8.5 Internals Involved with Virtual Functions and Polymorphism

When the C++ compiler encounters a class that has one or more virtual functions, the compiler creates or builds a `vtable` or virtual function table for that class. This `vtable` is used to select appropriate functions every time a virtual function of that class is to be called. Let us consider the employee hierarchy again here to illustrate the contents of `vtable` and how virtual function calls actually get resolved, code snippet of which is shown in Figure 8.2. Let us consider the following class structure. Again here we will not be very specific with the member function

```
Class Employee
{
char * fname;
char * lname;
public :
virtual void f1() const
{
cout <<"Function F1 \n";
}
virtual void f2() const
{
cout <<"Function F1 \n";
}
virtual void f3() const =0;
virtual void f4() const =0;
};
```

Figure 8.2: Sample class specification to illustrate virtual functions.

definitions/their names. We will be more interested to understand the intricacies behind the virtual function calls resolution.

The `vtable` is actually a collection of function pointers, the number of function pointers being equal to the total number of virtual and pure virtual functions in the base class. When the base class is compiled, the `vtable` consisting of four function pointers is created (we exclude the virtual destructor in this discussion). The first function pointer points to the implementation of the function `f1()` that displays the string function `F1`.

The second function pointer points to the definition of the function `f2()` that displays the string function `F2`. Note that functions `f1` and `f2` are virtual while `f3` and `f4` are pure virtual. That is `f1` and `f2` may or may not be overridden in the derive class. But `f3` and `f4` should be necessarily overridden in derived classes, else the derived class would become abstract class. We have discussed this already.

Getting back to `vtable`, the third and fourth function pointers that are for pure virtual functions are each set to `0` since these pure virtual functions do have implementation or function definition in the base class. Any class that has one or more zero pointers in the `vtable` is an abstract class. Classes without any `0` pointers in their `vtable` are concrete classes. Since virtual functions are propagated throughout the inheritance hierarchy, the compiler builds the `vtable` for all the derived classes as well.

Derived classes in which functions `f1` and `f2` are not redefined or overridden, the `vtable` function pointers are set to the copies of the `f1` and `f2` implementation in the base class or base class `vtable`. Derived classes in which `f1` and `f2` are overridden, the function pointers point to the respective function definitions. The remaining function pointers [which are for pure virtual functions] of each derived class `vtable` points to their respective function definitions. Polymorphism is achieved via a data structure that consists of three levels of pointers.

The function pointers discussed above form the first level. These pointers point to the actual functions that are to be executed when a virtual function is invoked. Whenever an object of a class with virtual functions is instantiated, the compiler automatically attaches a pointer to the `vtable` for the respective class in front of the instantiated object. These form the second level pointers. The third pointer is the handle on the object that receives the virtual function.

Let us assume the following function call of `bptr→f3()` after having assigned to `bptr` (base class pointer) the address of a derived class object. When the compiler encounters the above statement, it determines that the call is being made off a base class pointer and `f3` is a virtual function. The `vtable` pointer associated with the call is dereferenced to reach the respective derived class's `vtable`. Then the required number of bytes [in this case 8] is skipped to reach the function `f3()'s` function pointer in the `vtable`. This is with the assumption that a function pointer consumes 4 bytes of memory. Then this function pointer is dereferenced to form the actual function call to be invoked.

All these data structures are manipulated internally by the compiler and completely hidden from the programmer. The `vtable` and `vtable` pointers consume some memory. The additional execution time required for virtual function calls is because of the pointer dereferencing operations and memory accesses that occur on every virtual function call. A pictorial description of the virtual function resolutions is shown in Figure 8.3.

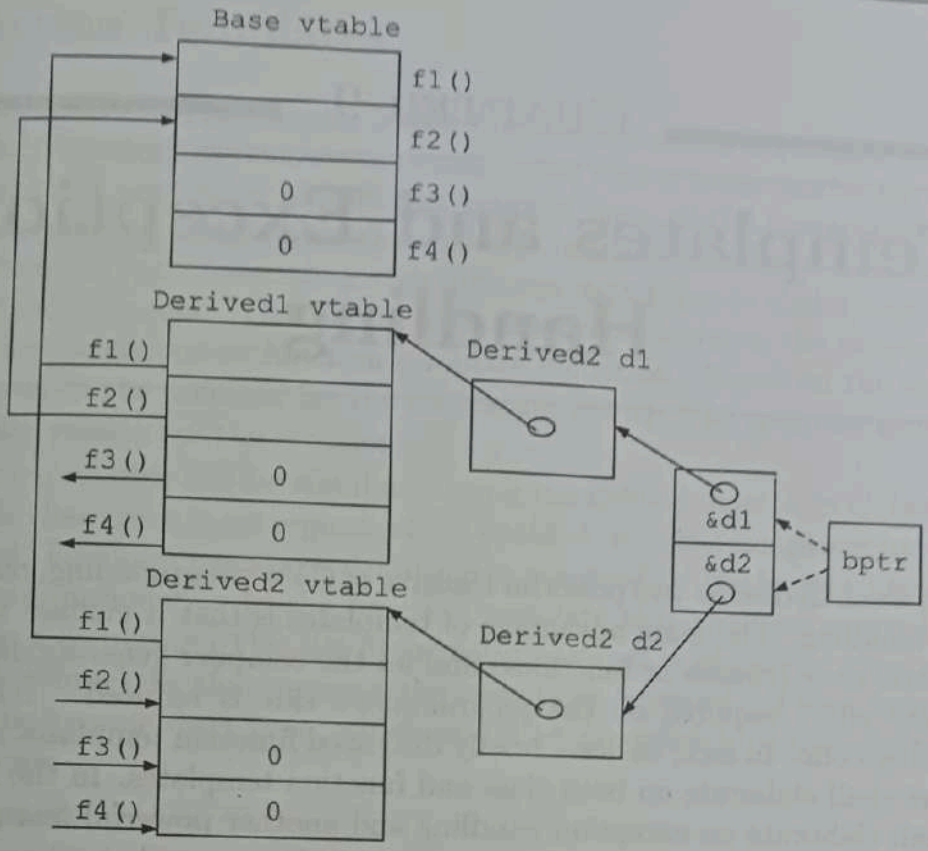

Figure 8.3: Virtual functions—Internals' illustration.

Review Questions

1. Define the term polymorphism and explain the support of polymorphism in C++.
2. Define the term abstract base class and discuss the syntax to create abstract base class in C++.
3. Can abstract base classes be instantiated? If not, what is the purpose of having such classes?
4. Appreciate the notion of polymorphism and virtual functions from a day to day computers related device usage point of view. Example with respect to the various media types used such as floppy, pen drive, cdrom, support polymorphic operations of open, read, write and close. Emphasis must be on the broad outlay of classes and member functions without getting into the details of physical read and write effects.
5. Explain in detail the data structure used to support virtual functions in C++.
6. Differentiate static from dynamic binding of objects.
7. For the university package created in Chapter 7 (exercise) support polymorphism with respect to operations that are common to different entities of the university such as printing the details of student/employee, insurance details, etc.
8. Differentiate a virtual from a pure virtual function.
9. Justify the need for virtual destructors in C++.
10. Give a few examples of programming situations in real life(day to day usage of computers) where polymorphic behaviour is required.

CHAPTER 9

Templates and Exception Handling

In this chapter, we shall discuss two powerful features of C++ programming, namely templates and exception handling. The main advantage of templates is that it enables the programmer to develop generic code (classes or functions) and let the compiler generate data type specific code. Thus, the effort required on the programmer's side is reduced. Templates help in developing generic code. In fact, we have briefly discussed function templates in Chapter 3. In this chapter, we shall elaborate on both class and function templates. In the later half of the chapter, we shall elaborate on exception handling and another powerful feature of C++ that helps programmer decide the way exceptions or abnormal program behaviour to be handled.

9.1 Introduction

One requirement for function templates would be to develop a function called print array that should display the contents of either an integer or float or character array passed to it as an argument. In other words, the generic specification that displays the contents of the array, not relating to any specific integer or character or float array. This generic specification is referred to as function template. Based on the requirement or the specific data type with which a function template is called or invoked, the compiler on behalf of the programmer generates the data type specific code or the function definition, which handles the specific data type. Such compiler-generated versions of function definition are referred to as template functions. The reasoning behind calling them as template functions is that these are functions that are generated from templates (function templates to be precise).

On similar lines, generic class specifications or behaviours are referred to as class templates, and data type specific class specifications are referred to as template classes. One example for class template requirement would be to create a generic stack class and then instantiate or create specific integer, or float or character stack. To correlate the concept of templates with realistic example, function and class templates are similar to stencils from which differing shapes can be created or traced. Template functions and template classes are nothing but the separate tracings that are all of the same shape but can differ in their colour. Here, the similarity in shape of different coloured tracings is the genericness [template] and the various colours available are equivalent to the data type specific versions generate from the generic templates. In fact, templates promote the concept of software reusability to a great extent. We will first explore the concept of function templates and then proceed on to class templates.

9.2 Function Templates

We have already mentioned in Chapter 3 that function templates are similar to function overloading concept. Function overloading is normally the choice when similar but not equivalent operations need to be performed on different types of data. If the operations are identical or equivalent across different data types, the choice is function templates. Function overloading is the choice when the logic associated with different data types is slightly different, but definitely not identical or similar. Getting on with function templates, the programmer specifies a generic function behaviour or function template definition. Based on the arguments types (data type) passed to the function (at the invocation point). The compiler generates separate data type specific versions of the function.

To a certain extent this can be simulated using the defined macros in C, but macros suffer from the setback that there is no type checking enabled, whereas templates have in them type checking enabled. Function `template` definition is preceded by the keyword template followed by a list of formal parameters to the function template that shall be referred to within the function `template` definition. This list of formal parameter is of generic type and is specified within the angled brackets that succeed the `template` keyword. These formal parameters correlate with the generic data type, which would be predefined, or user-defined. The exact syntax is as follows:

```
Template <class T>
```

where T is some predefined or user-defined generic data type.

The formal type parameters of a template definition are used to specify the argument types to functions, functions return type and for declaring variables for use within the function definition. Here, in the function definition, we are referring to the generic behaviour. The function definition follows the function template header and the definition is quite similar to that of a normal function definition with the only difference that only generic data types and not specific data types are manipulated within the function template definition. Let us now develop a generic swap function `template` and then leave it to the compiler to generate data type specific code to swap integers or characters or floating point inputs, code for which is as shown in Figure 9.1.

The generic swap function `template` declares a single formal parameter `T` (`T` to be some valid identifier) for the type of data to be swapped by this function `template`. When the compiler detects an invocation of swap function, the type of data with which the swap function is invoked replaces the generic data type `T` throughout the function `template` definition of swap. This compiler substitute or generate code for a specific data type is the `template` function. For better understanding, the invocation of function swap with integers [template function for integer data type passed to a function `template swap(T)` generates the following integer data type specific version of the function `template swap`:

```
void swap (int & input1, int & input 2)
{
int temp;
temp = input1;
input1 = input2;
input2 = temp;
}
```

```
template <class T>
void swap (T & inp1, T & inp2)
{
T temp;
Temp = inp1;
inp1 = inp2;
inp2 = temp;
}
#include<iostream.h>
int main ( )
{
int num1 = 20, num2 =30;
float no1 = 5.5, no2 = 6.6;
char ip1 ='A', ip2 ='B';
cout << "Swapping Two Integers \n";
cout << "Before Swapping Values are \n";
cout << num1 <<"\t"<<num2<<"\n";
swap (num1,num2);
cout << "After Swapping Values are \n";
cout << num1 <<"\t"<<num2<<"\n";
cout << "Swapping Two Floating Point Numbers \n";
cout << "Before Swapping Values are \n";
cout << no1 <<"\t"<<no2<<"\n";
swap(no1,no2);
cout<< "After Swapping Values are \n";
cout<< no1 <<"\t"<<no2<<"\n";
cout<< "Swapping Two Characters \n";
cout<< "Before Swapping Values are \n";
cout<< ip1 <<"\t"<<ip2<<"\n";
swap(ip1,ip2);
cout<< "After Swapping Values are \n";
cout<< ip1 <<"\t"<<ip2<<"\n";
return 0;
}
```

Figure 9.1: C++ code to demonstrate function templates.

Similarly, for other invocations of swap with `float` and `char` data type the compiler generates the code for the respective template function from the generic function template definition by suitably replacing the generic data type `T` with the specific data type (`char` or `float`). This is left as an exercise for the reader. Every different formal type parameter of a function template definition should be specified or identified in the formal parameter list of the function template definition header. The name of a formal parameter type in the template header can be mentioned only once and cannot be duplicated. However, with template functions, these formal parameter types need or will not remain unique. The instantiation or invocation of function templates is illustrated in the definition of function `main()` where we have three calls to swap, first call accepting two integers, second call accepting two float point numbers, and the third call accepting two character data type inputs. As is shown the two entities [integer or `char` or `float`] are swapped.

The `call swap (&num1,&num2);` causes the compiler to replace `T` in the generic function `template swap (T &, T&);` with integer and generate the swap template function associated for integer arguments. Similarly, the calls to other data type specific inputs are han-

dled by the compiler. Thus, the above example saves the programmer effort in developing three separate overloaded functions with the following prototypes of `void swap: void swap (int &,int &); void swap(float &,float &); void swap (char &,char &);` all of which would have the same code and differ only in their data types. Templates though support the concept of software reusability and reduces the programmer's effort by generating code (data type specific), multiple copies of template functions and template classes are still instantiated in a program despite the fact that the function or class template is specified only once.

We have earlier said that function templates have the concept of function overloading supported after the template functions have been generated. Thus, there is a considerable amount of memory consumption involved with templates feature of C++. Note that function templates can also be overloaded to have different generic behaviours associated with the same function name. The way templates are handled by the C++ compiler is as follows. It has already been mentioned that when a function is invoked, the compiler performs a matching process to locate the function definition that matches in signature (function header) with the called functions signature.

This methodology is applicable with template based c++ programs as well. First, the compiler checks for such precise match. If this comparison fails, the compiler proceeds checking if a function `template` from which template functions can be generated possibly match the signature of the function call [in number of arguments and the individual data type]. If the compiler is able to locate such a template match, the compiler generates the appropriate `template` function and invokes it to handle the function call. In cases where the compiler cannot find a valid match or multiple matches are encountered a suitable compiler error is generated. Let us look at another example for function templates. A generic sort function to sort either an array of integers or characters or floating point numbers shall be developed in the following example, code for which is shown in Figure 9.2.

9.3 Class Templates

In this section, we will explore the class templates feature of C++. Stack is one important data structure that is so very useful in expression evaluation. A stack is a constrained version of a linked list, new nodes(data) can be added or removed or insertion / deletion is only at the top. This is also referred to as the LIFO (Last In, First Out) data structure. This differs from QUEUE (FIFO—First In, First Out) in that deletion from the front while insertion is at the back. Based on the realistic Q syntax of newly arriving people at the back or rear of the Q, while the server or the service person caters to or services the person at the front end [deletion]. One important application of stacks is function call mechanism.

For example, when a function call is made, the called function must know the return address [caller address] to resume program execution. Return addresses are normally pushed onto the stack. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return to its appropriate caller. Recursive function calls are supported by stacks similar to non-recursive function calls. Having elaborated on stacks and their applications, let us now get back to the development of a generic stack class template from which either stack of integers or floats or characters can be [template classes] created. We shall implement the stack data structure using arrays, code for which is as shown in Figure 9.3.

```
template < class T>
void bubblesort (T arr[], int size)
{
for (int pass =1,pass<size;pass++)
for (int count =0;count<size-1;count++)
if( arr[count]>arr[count+1] )
{
T temp;
temp = work[count];
work[count]=work[count+1];
work[count+1]= temp;
}
}
#include<iostream.h>
#define SIZE 5
int main ( )
{
int array1[SIZE] = {5,4,3,2,1 };
float array2[SIZE] = {5.5,4,4,3.3,2.2,1.1};
char array3[SIZE] = 'E','D','C','B','A';
cout << "Sorting in Progress \n";
cout << "Integer Array Being Sorted \n";
bubblesort(array1,SIZE);
cout << "The sorted Integer Array is \n";
for (int i=0;i<SIZE;i++)
cout << array1[i];
cout << "Floating Point Array Being Sorted \n";
bubblesort(array2,SIZE);
cout << "The sorted Floating Point Array is \n";
for (int i=0;i<SIZE;i++)
cout << array2[i];
cout << "Character Array Being Sorted \n";
bubblesort(array3,SIZE);
cout << "The sorted Character Array is \n";
for (int i=0;i<SIZE;i++)
cout << array3[i];
return 0;
}
```

Figure 9.2: Function `template` to sort an array of integers.

```
template <class T>
class Stack
{
public :
Stack (int =20);
~Stack ();
bool push (const T & );
```

Figure 9.3: Continued

```
bool pop ( T & );
private : int size;
int top;
T * stackpointer;
bool empty( ) const
{
return (top == -1);
}
bool isfull ( ) const
{
return (top == size -1);
}
};
template <class T>
Stack<T> :: Stack (int sz)
{
size = sz >0?sz:20;
top = -1;
stackpointer = new T [size];
}
template <class T>
Stack<T> :: ~Stack( )
{
delete [] stackpointer;
}
bool Stack<T> :: push (const T & pval)
{
if (!(isfull()))
{
stackpointer [++top] = pval;
return true;
}
return false;
}
bool Stack<T> :: pop (T & popval)
{
if (!(isempty()))
{
popval = stackpointer[top-];
return true;
}
return false;
}
#include <iostream.h>
int main ( )
{
float f = 2.2;
int val = 2;
Stack<float> sof (4);
while (sof.push(f))
```

Figure 9.3: Continued

```
{
cout << "pushed \t"<<f<<"\n";
f+ =1.5;
}
cout << "Sorry ! The Float Stack is Full \n";
cout << "Popping from the Float Stack \n";
while (sof.pop(f))
cout << f;
Stack<int> soi (4);
while (soi.push(val))
{
cout << "pushed \t"<<val<<"\n";
val =1.5;
}
cout << "Sorry ! The Int Stack is Full \n";
cout << "Popping from the Int Stack \n";
while (soi.pop(val))
cout <<val;
return 0;
}
```

Figure 9.3: C++ stack class template code.

9.4 Stack Class Template Code Interpretation

The stack class template contains at its data members size of the stack (size), a cumulative index (top) that refers to the current top index or element of stack. The `int * stackpointer` data member of the class refers or points the first element or the base address of the array or the first array element. The constructor for the class template stack (defaults the size of the stack to 20) validates the size value passed to it. If the size value passed is not >=0, then the size is set to the default value of 20.

At the point of construction since the stack would be empty [no elements], the top data member [stack pointer index] is set to -1 to indicate that the stack at the point of construction is empty. The constructor then allocates the space required for the respective stack type. The destructor of Stack class template deallocates the space allocated for the stack pointer in the constructor using the delete operator. The class template has two helper or utility functions, namely `isfull`, `isempty` to check for if the stack is currently full in which case more elements cannot be pushed or not. `Isempty` is used to test if the stack is empty or not. In case of empty stack, elements cannot be removed from the stack (popping). An empty stack condition occurs when the top index =-1 and full stack occurs when top index = size -1.

The class template stack should support two basic operations of push and pop. Push member function is used to [push or insert values onto the stack. The push member function accepts as argument the value to be pushed onto the stack. A value of generic type is pushed onto the stack after checking for the `isfull` condition. If the stack is not full (`!isfull()`), then the value `pval` is pushed onto the stack by the statement `stackpointer [++top] = pval`. This top index is preincremented since the value to be pushed should be stored in the next index of the array. At the end of a successful push operation the push member function returns a true result to indicate that the push operation was successful. Successive push

operations are performed only if an immediately preceding push operation was successful (for a new push to be allowed, the earlier push should be successful).

The pop function is used to delete or remove elements in the last in first our fashion. The pop member function accepts an argument of generic type T to store the popped value. Note that this argument cannot be made constant since for the popped value to be stored in T, one should have write or assignment operations over the argument. Both push and pop member functions accepts arguments by reference to avoid the duplicating effect in case of value pass.

The pop member function checks for the stack being empty condition. If the stack is not empty, we proceed with the pop operation by decrementing the top index [top –]. Note that this is a postdecrement operation, since we require first the element at the top of the stack to be popped and then proceed to the next element. Thus, here the element `stack[top]` is retrieved first and then later the top index is decremented so that after the topmost element has been popped, the new top element of the stack would be earlier top index -1. The retrieved value is stored in the `popval` argument. The driver routine creates stacks of the integer and floating point data types respectively and invokes the push and pop functions to push/retrieve elements from the stack.

9.5 Exceptions

The exhaustive features supported by any programming language and C++ in specific brings along the possibility of several types of errors and erroneous conditions that the program has to encounter and handle. The feature in C++ that supports this mechanism of handling errors and exceptions is what is referred to as Exception Handling. One possible and most often encountered exception in programming is the divide by zero exception (the code for which is shown in Figure 9.4), in which case most languages come out with a prompt of Invalid division operation and terminating further program execution. C++'s exception handling differs from other languages in the fact that the error handling code is clearly demarcated from the main processing code. But in languages such as error handling code often gets mixed up or interspersed with the main program code, contributing to poor maintenance and understanding of the code.

C++ exception handling allows programs to identify and handle errors rather letting them happen and result in serious consequences. It is generally the choice when the program can recover from the error situation and the recovery procedure that handles the error is referred to as the exception handler. It is primarily designed to handle synchronous errors such as divide by zero that occur as the program executes the concerned statement or instruction. Asynchronous errors such as those involving disk reads, mouse clicks, etc. are not handled by exception handlers. The exception handling feature of C++ is so designed such that in case of no exceptions in the program, then the overhead as a result of exception handling code is minimal to the extent possible.

9.6 The `try`, `throw` and `catch` Syntax

Exception handling in C++ incorporates three syntactic constructs or keywords, namely `try`, `throw` and `catch`. The `try` construct or block as it is often referred to encompasses the program that is susceptible to errors or exceptions. Once an exception is generated, it is

```
#include<iostream.h>
class Example
{
public:
Example()
{
message=new char[30];
strcpy(prompt, "Invalid Divide by 0 Attempted \n");
}
char * message()
{
return prompt;
}
private:
char*prompt;
};
int main ()
{
int no1,no2;
float ans;
try
{
if(no2==0)
throw Example();
ans=no1/no2;
}
catch(Example e)
{
cout<<"Exception has occurred \n"<<e.message();
}
cout<<"Program Execution Successful, result is \n"<<ans;
return 0;
}
```

Figure 9.4: C++ code to handle divide by zero exceptions.

thrown (sent out) for an appropriate handler or exception handler to process and handle the exception. The entity or block that handles the exception is referred to as the `catch` block. Ideally, a `try` block is followed by several `catch` blocks, so as to say that each handler takes of specific exception situations. In the event of an exception being generated, the appropriate `catch` handler is identified and code belonging to that `catch` block is executed.

Situations where an exception has been thrown and none of the existing handlers can handle it, function `terminate` is called, which in turn invokes the abort function. Cases where exceptions are not generated, the entire `catch` block(s) following the `try` block is ignored and program execution resumes from the point after the last `catch` block. Exceptions are thrown within the `try` block or from a function that either directly or indirectly contains a `try` block. The statement or point at which an exception is thrown is referred to as the `throw` point. Once an exception is thrown control cannot return back to the `thrown` point. Information about the exception such as type or nature can be passed on to the exception handlers or `catch` blocks. Yes, exception details are passed as arguments to the appropriate `catch` handlers, which based on a parameter match invoke the intended exception handling routine.

Let us now see an example to handle the divide by zero exception. The code is as shown in Figure 9.4 and the class Example that involves a division behaviour (function) is equipped to handle the divide by zero situation.

As can be seen in the above code, the `try` block throws an exception of the user-defined class type (Example), when the denominator is zero, which is caught by the appropriate `catch` handler (only one in this case which accepts as parameter an object of class Example). The program then executes the handler resulting in the display of the message Exception has occurred followed up by the specific prompt message that it was An attempt to divide by zero that occurred. In the case of no exceptions being generated, normal execution of the code follows, resulting in the display of the computed quotient value.

9.7 The `throw` Construct

The keyword `throw` in C++ denotes the throw point and signals the occurrence of the exception. Along with the `throw` keyword, one operand, which could be an object or some other type can be thrown. It is this object or data type that is communicated to the external handlers to appropriately get invoked and handle the exception in the intended manner. Once an exception is thrown, it is generally handled by the handler or the `catch` block that is closest to the `try` block in which the exception occurred (of course the thrown data type or exception type needs to match the `catch` parameter). All the exception handlers or `catch` blocks for a specific `try` block are specified immediately after the `try` block.

Once an exception is thrown, control from the `try` block is exited and shifted to the appropriate `catch` routine or handler that can handle the exception. Situations where the `throw` statement lies deep within a `try` block or nested in a function call, still the appropriate `catch` handler is invoked and activate, once an exception is generated. As an example, out of bounds array subscription could be one exception situation where the indexing statement checks for the bounds to fall within the limits or not. When this condition fails, an exception is thrown and the `catch` handler that handles it is executed.

9.8 The `catch` Construct

As stated above, exception handlers are generally part of the `catch` blocks. A `catch` block contains the keyword `catch` followed by an optional parameter list (similar to functions) that identifies the type of exception. When a thrown exception is caught or there is a parameter match in terms of the exception type, the code that forms the definition of the `catch` block is executed. This is referred to as an exception being caught and handled appropriately. The `catch` block can contain either named or unnamed parameters. In the case of named parameters, it can be referenced within the `catch` block, while in the other situation, only the type (exception type) would suffice to identify the exception and then have an appropriate handler executed.

The `catch` handler that matches a thrown exception type or catches an exception is the first one that lies after the currently active `try` block. A `catch` block such as `catch(...)` or ellipsis catches all types of exceptions or is a generic exception handler. Thus, such a `catch` block is best placed after all possible `catch` blocks. Situations where an ellipsis `catch` block

precedes all other possible handlers, would always have the generic handler executed, ignoring the other what could be case specific exception handlers.

`catch` blocks are searched in the order in which they are listed after the `try` block and the first
one that yields a match is executed. Once the appropriate `catch` block is executed, program execution switches to the first statement after the last `catch` block associated with the currently active `try` block. At any given point in time only one `catch` block gets executed, after which the other following `catch` block, if any are ignored. Cases where there is no match of the thrown exception with the handler types, function `terminate` which indirectly calls abort is called, resulting in program termination. This signifies the fact that the thrown exception cannot be handled by any of the user-defined handlers and hence control is transferred to the system-defined terminate procedure to stop further program execution. Since there can be the situation of several handlers that can handle a specific exception, the order of listing of the `catch` blocks is crucial to the entire process of exception handling.

Similar to the `catch` ellipsis syntax, a `catch` block that catches a base class type object would yield a match for the corresponding derived class object types as well. This is based on the argument that, derived class objects can always be treated as instances of the base class object and such a base class object `catch` block should generally constitute the trailing half of the exception handler codes. Alternatively, instead of having separate classes for different exception, a generic exception class with different private data members associating the different exception types could also be used. In fact, it is this approach that is adopted by the system-defined exception handler classes. There is no conversion (promotion or demotion) of the thrown data types in the process of exception matching and handling.

A `catch` block though similar in nature to a `switch` statement differs in the sense that every `catch` block has its own unique and distinct scope. However, with `switch` constructs, all the cases fall under the same scope of the outlying `switch`. Hence the need for `break` statement within switch-case constructs is alleviated with respect to exception handlers or `catch` blocks. Also, handlers cannot access variables or objects within the `try` block, because once an exception occurs control is exited the `try` for ever and transferred to a matching `catch` block. Throw point cannot be reached back from a `catch` block using a `return` statement.

Exception handlers differ in their approach of handling the exception. Some might decide to terminate program execution, while others might perform recovery operation and resume execution after the last handler. There can also be handlers which convert exceptions for other handlers to handle and process the exception. Ideally, the `catch` block is expected to perform resource clean-up (in the bare minimum). One such operation could be reclaiming memory spaces allocated using the new operator with the delete operation.

In certain cases, a handler or a `catch` block might also decide to rethrow an exception. That is when it is not able to process the exception, it might throw the same exception (rethrow) with a throw (just the keyword throw after an earlier throw operand) for other handlers to process the exception. In such cases, the `catch` blocks associated with the next following `try` block are the one that can possibly handle rethrown exception. Also, `catch` blocks themselves can throw new exceptions to be handled and processed by encompassing `try` associated `catch` handlers. Consider another example for situation handling where the user wishes to have an appropriate handler invoked in cases of failed memory allocation attempts.

One alternative is to check for non-null condition of the allocated block. The code shown in Figure 9.5 handles failure by the new operator to allocate the requisite space by invoking

the what function of the predefined exception class. Note the usage of the new header file to make use of the features supported by the **bad_alloc** class that is employed to generate the message related to failure of new to obtain the requisite blocks of memory. Alternatively, users can as well make use of the **set_new_handler** routine with argument as the customized handler function or routine that is to be called, instead of a case specific **try** and **catch** block syntax. C++ supports the **exception.h header file** with a what message interface support to identify the appropriate error message depending on the exception generated. exception is the base class with several possible derivations associated with errors to check for runtime error, overflow, underflow errors, etc. Interested readers must pursue the respective header file for further application and usage.

```
#include<iostream.h>
#include<new.h>
int main()
{
int * p [200];
try
{
for (int i=0;i<100;i++)
{
p[i]=new int[10000000];
cout<<"Allocated space for p["<<i<<"]";
}}catch (bad alloc exception)
{
cout<<"Exception Occurred \n";
cout<<exception.what();
}
return 0;
}
```

Figure 9.5: Exception handling in cases of new failure.

Review Questions

1. Discuss the need for templates-based programming.
2. Differentiate function from class templates.
3. Develop a function template MAX in C++ that computes the maximum of three integers or three floating points of three characters (based on ascii values).
4. Develop a function template SORT in C++ that sorts either an array of integers or characters or floating point numbers.
5. Develop a Matrix class template in C++ that supports floating or integer matrix arithmetic of addition and subtraction.
6. Develop a Linked List class template in C++ with support for operations insertion, deletion and search of an element. A linked list is basically a collection of elements with the first element pointing to the next and so on, The last element links to NULL. Note in a linked list elements need not be in contiguous memory locations as is the case with arrays.