Chapter One

1. Algorithms and problem solving

1.1. Programming Language

A language is a systematic set of rules for communicating ideas. With a natural language, like English, the communication is between people and the language is used in both spoken and written. A programming language is a language for communication between a person and computer. The content of communication is known as a program. Programs are set of instructions, which enable a computer to perform a required operation.

Several hundred programming languages are in use today, each varying in their specific syntax, but showing similarities in purpose. Input/output instructions, arithmetic instructions, logic instructions, control instructions, data movement instructions and specification instructions are such common purpose instructions. Together with the improvement of hardware and processing speed and memory size increase computer languages changed to languages that were easier for humans to understand. So generations of programming languages are developed to correspond with the generations of computer hardware.

Generations of Programming Language

Thus, programming languages fit into five generations: First generation languages, second-generation languages, third generation languages, fourth-generation languages and fifth-generation languages. These generations of programming languages can also be categorized into two broad categories: -low and high-level languages. Low-level languages are machine-dependent; that is, they are designed to be run on a particular computer. In contrast, high-level languages (for example, COBOL and BASIC) are machine-independent and can be run on a variety of computers.

The first two generations were low-level and the rest high-level of programming languages. The higher the level of the language, the easier it is to understand and use. For example, in a fourth-generation language you need only instruct the computer system what to do, not necessarily how to do it.

When programming in one of the first three generations of languages, you have to tell the computer what to do and how to do it. What comprises a new generation is less clear; therefore, languages after the fourth generation are referred to as very high-level languages.

With each new level, **fewer instructions** are needed to tell the computer to perform a particular task. A program written in a second-generation language that computes the total sales for each sales representative, and then lists those over quota, may require 100 or more instructions; the same program in a fourth-generation language may have fewer than 10 instructions.

The ease with which the later generations can be used is certainly appealing, but the earlier languages also have their advantages. All generations of languages are in use today.

First generation (Machine languages, 1940's): Difficult to program in; they are dependent on machine languages of the type of computer being used. However, machine language allows the programmer to interact directly with the hardware, and it can be executed by the computer without the need for a translator.

Machine language is the ultimate low-level language. To communicate with the first generation computer programmers had to write programs in machine language: - the 0's and 1's of binary code. Programming in 0s and 1s needed reducing statements such as add, subtract and divide into a series of 0's and 1's. Programs written in this way are machine-dependent. There is no universal machine language as different hardware designs dictate different data and memory locations, and hence different bit strings. Programmer must remember exact memory addresses used to store data and instructions. They execute very quickly and use memory very efficiently, as no conversion from source code is necessary. Writing the programs is difficult, tedious, time consuming and hard to debug. A high level of understanding of hardware specifics is required. All programs written using other languages are translated into machine language before they can be executed.

End users who wanted applications have to work with specialized programmers who could understand, think and work directly in the machine language of a particular computer. So programming in machine is slow and labor intensive process.

Second generation (Assembly languages, appeared in the early 1950's): Use symbolic names for operations and storage locations. Assembly language is easier to use than machine language but it is still difficult to understand. A systems program called an **assembler** translates it into machine language. Different computer architectures have their own machine and assembly languages, which means that programs written in these languages are not portable to other, incompatible systems. Many programmers still use assembly languages because they still give them close control over the hardware.

Instead of using 0's and 1's programmers substituted language like acronyms and words such as add, sub (subtract) in programming statements.

- Costly in terms of programmer time
- Difficult to read and debug, and
- Difficult to learn
- Highly used in system software development

Programs are easier to create and to debug than machine language, but are still prone to errors and tedious and execute faster than high-level languages.

Third generation (High-level languages, emerged from 1950's - 1970's): Use English-like instructions like print and mathematicians were able to define variables with statements such as Z=A+B. Many high-level languages are **portable**. Each has syntax rules that must be followed. Such languages are much easier to use than assembly language but they can take considerable time to learn. Two types of systems programs exist for translating them into machine language: interpreters and compilers.

In High-level languages, instructions closely resemble human language and mathematical notation. Programmers do not require detailed knowledge of hardware specifics as these languages extensively use symbolic representations of memory addresses and library functions. The use of common words (**reserved words**) within instructions makes them easier to learn. Assembly language code segments may be added to most high-level programs, where faster execution speed is required by the programmer. There are two types of language-translator programs used to translate into machine code, Interpreters and Compilers.

Fourth-generation (Since late 70's): Have simple, English-like syntax rules; commonly used to access **databases**. As we mentioned earlier, the third-generation programming languages discussed in the previous section are all **procedural languages** because the programmer must list each step and must use logical control structures to indicate the order in which instructions are to be executed. Fourth-generation languages (4GLs), on the other hand, are **nonprocedural languages**. These languages can be compared to the way in which you might instruct someone to cook a meal. The **nonprocedural method is simply** to state the needed output: fix a meal of chicken, rice, and salad. Using the procedural method, on the other hand, involves specifying each step - from preparing the shopping list to washing the dishes.

Obviously, the nonprocedural method is easier to write, but you have less control over how each task is actually performed. For example, the dishes might be washed by hand or in a dishwasher. When using nonprocedural languages, the methods used and the order in which each task is carried out are left to the language itself; the user does not have any control over it. In addition, 4GLs sacrifice computer efficiency in order to make programs easier to write; hence they require more computer power and processing time. As the power and speed of hardware have increased and its cost has decreased, the use of 4GLs has spread. Because fourth-generation languages have a minimum number of syntax rules, people who have not been trained as programmers can use such languages to write application programs, as they need them. This saves time and fees professional programmers for more complex tasks. There are several categories of

4GLs languages. The most common once are: query languages, report generators, and application generators and graphic languages.

Query languages allow the user to retrieve information from databases by following simple syntax rules. For example, you might ask the database to locate all customer accounts that are more than 90 days overdue. Example of query languages is SQL, which has become a de facto standard.

Report generators produce customized reports using data stored in a database. The user specifies the data to be in the report, how the report should be formatted, and whether any subtotals and totals are needed. For example, you might ask the system to create a list, arranged by account number, of all the company customers located in Debre Markos.

The user can access the database when using query languages and report generators, but the database cannot be altered with these tools. With **application generators**, however, the user writes programs to allow data to be entered into the database. The program prompts the user to enter the needed data. It also checks the data for validity.

Application generators contain preprogrammed modules that can generate entire applications, greatly seeding development. A user can specify what needs to be done, and the application generator will create the appropriate code for input, validation, update, processing, and reporting.

Graphic languages that retrieve data from files or databases and display them in graphic format. Users can ask for data and specify how they are to be charted. Some graphic software can perform arithmetic or logic operations on data as well. SAS, Harvard Graphics, and Lotus Freelance Graphics are popular graphics tools.

Fifth-generation (1990's): Used in artificial intelligence and expert systems; also used for accessing databases. Fifth-generation languages (5GLs) are also nonprocedural languages and are also commonly used to query databases. Because these languages are still in their infancy, only a few are currently commercially available. They are closely tied to artificial intelligence and expert systems.

Fifth-generation languages are the "natural" languages whose instructions closely resemble human speech "Get me Abebe's sales figures for the 2009 financial year" is a typical instruction. Very powerful hardware and software are required to execute such programs because of the complexity involved in interpreting commands entered in human language. This means computers can in the future have the ability to think for themselves and draw their own inferences using programmed information in large databases. Complex processes like understanding speech would appear to be trivial using these fast inferences and would make the software seem highly intelligent.

1.2. Problem solving using computers

Problems: Undesirable situations that prevent an organization from fully achieving its purpose, goals and objectives

- True problem situations, either real or anticipated, that require corrective action
- Unexploited Opportunities to improve a situation despite the absence of complaints
- Directives to change a situation regardless of whether anyone has complained about the current situation

Problem solving is the process of transforming the description of a problem into the solution of that problem by using our knowledge of the problem domain and by relying on our ability to select and use appropriate problem-solving strategies, techniques, and tools.

Problem Solving: is basic intellectual process that has been refined and systemized for the various challenges people face.

The need to problem solving is to achieve the following basic issues which abbreviated as PIECES.

- The need to improve performance
- The need to improve information and data
- The need to improve economics (minimize cost and increase profit)
- The need to improve control/security
- The need to improve efficiency
- The need to improve service to customers, suppliers, partners, employees, etc.….

Problem Solving Approaches

Computer software development may be done in two different ways. The first way is called **top down development** and the alternate approach is called **bottom up development**.

Top down design: is a systematic approach based on the concept that the structure of the problem should determine the structure of the solution and what should be done in lower level. This approach will try to disintegrate a larger problem into more smaller and manageable problems to narrow the problem domain.

Bottom up design: is the reverse process where the lowest level component are built first and the system builds up from the bottom until the whole process is finally completed.

In reality, large systems are usually developed using a combination of the both methods. Top-down usually works well for small systems but for large complex systems, the combination is the best approach and in practice, what actually happens.

1.3. Basics of program development

The vehicle for the computer solution to problem is a set of explicit and unambiguous instructions called programs expressed in programming language.

Quality programming is necessary for the economic and correct solution of problems. This only comes from good program design. The programs we design need to be:

Reliable: the program should always do what it is expected to do and handle all types of expectations

Maintainable: the program should be in a way that it could be modified and upgraded when the need arises.

Portable: it needs to be possible to adapt the software written for one type of computer to another with minimum modification.

Efficient: the program should be designed to make optimal use of time, space and other resources.

1.4. Software Engineering

Software Engineering has been used to describe building of software systems which are so large or so complex that they are built by a team or teams of engineers.

Programming Life cycle

The method (approach) that software Engineers use in solving problems in computer science is called the **software development method**. Another name that is commonly used for the software development method is **software life cycle**.

The software life cycle has the following components

1. Preliminary Investigation
2. Analysis
3. Design
4. Implementation
5. Testing
6. Maintenance

Preliminary Investigation

- **Defining the problem** (examines documents, work papers, and procedures; observe system operations; interview key users of the system). Suggesting a solution (often improving an existing one or building a new information system)
- **Feasibility study** (determine whether the solution is feasible)

- Technical **feasibility:** whether implementation is possible with the available or affordable hardware, software and other technical resources.
- Economic **feasibility:** whether the benefits of the proposed solution outweigh the costs
- Operational **feasibility:** whether the proposed solution is desirable within the existing managerial and organizational framework.

Analysis (Requirement gathering)

– Try to understand the business in general (activities done, how it is done, etc.)

– Define the specific information requirements: who needs what information, where, when, and how. Present a detail description of the functions the new system must perform

– In this phase we identify:

- Inputs to the problem and their form (Filing status: Single, Married, divorced)
- Outputs expected from the solution and their form (Intermediate results or Final results – Completed tax information)
- Constraints (What are the limits on the data? Income cannot be a negative number)
- Assumptions (Problem deals in $US, Only for 2005)
- Formulas (For determining the area, it is length x width)

Design

Based on the requirements specified in the analysis phase algorithms are developed at this stage. Develop a series of steps with a logical order which, when applied to the input would produce the specified output.

- Consider alternative technology configurations to develop the system (hardware, software, security capability of the system, network alternatives etc.)
- Management and control of the technical realization of the system: detailed program specification,
- Detail system specifications for the functions identified in the analysis: managerial, organizational and technological components of the system solution (input, output, user interface, database design, processing, manual procedures, controls: input, processes, output and procedural controls; security, documentation, training, organizational change)

Implementation

During implementation, we translate (code) each step of the algorithm in the design specifications into a statement in the particular language selected, and end up with a computer program.

Testing (Will the system produce the desired results?)

- **Unit testing** (Program testing): test each unit of program in the system separately
- **System testing**: test the functions of the system as a whole (performance, capacity for file storage, recovery and restart capabilities)
- **Acceptance testing**: system is evaluated by users and management

Maintenance

Changes in hardware, software, documentation, or procedures to correct errors, meet new requirements, or improve processing efficiency.

- Fix bugs found by the customer
- Make modifications
- Add new features

Documentation: Written descriptions of, specifications, development, and actual code of program. Documentation is valuable because:

1. It is probable that you may come back to this program sometime in the future in order to use it again, or you may want to use part of it in developing the solution for another similar problem.
2. If you have developed this program for a client's use, he or she will need some information so that in your absence the program can be used.
3. If you are working for a company and have developed this program as an addition to the company's software library, some other programmer may be assigned to maintain it or to make additions to it.
4. It is possible that sometime after you have developed the program, you may discover some errors in it and you may be required to correct them.

The significance of proper program documentation in the software life cycle cannot be overemphasized. Program documentation consists of

1. A concise problem specification;
2. Descriptions of inputs, expected outputs, constraints, and applicable formulas;
3. A pseudo code or flowchart for its algorithm;
4. A source program listing;
5. A self-documenting code: a program containing meaningful identifiers as well as clarifying comments;
6. A hard copy of a sample of data and output of test run of the program; and
7. A user's guide explaining to nonprogrammer users how the program should be used.

1.5. Algorithm development

An algorithm is procedure for solving a problem in terms of

- the action to execute (what to do) and
- the order in which these actions are executed(done)

An algorithm needs to be

- Precise and unambiguous (no ambiguity in any instruction and in the order of execution)
- Simple
- Correct
- Finite (has to have an end)
- Produce expected output
- Efficient: in time, memory and other resources

An algorithm can be expressed in many ways. Some of these methods are **narrative, flowchart and pseudocode.**

Narrative: often used to narrate the algorithm, can be understood by any user who may not have any knowledge of computer programming. Too wordy, too unambiguous and can be interpreted in different ways.

Example: Accept salary of the employee. Calculate bonus as 10% of salary and add it to salary. Accept service year of employee. If the service year is greater than 10, give additional 100 birr as bonus. Display the bonus of the employee.

Flowcharts: a diagram consisting of labeled symbols, together with arrows connecting one symbols to another.

Basic flowcharting symbols are:

Terminal point: marks the beginning or end of a program

Process: indicates any arithmetic computations or data movement

Decision: Indicates a yes/no decision to be made by the program

Input/output: is used to show input or output data

Flow line: is used to show the direction of logical flow

On-page Connector: used to connect two points without drawing a flow line

Annotation flag: used to add clarifying comments or descriptions

Inter-page Connector: used as exit or entry from a flowchart on one page to a flowchart in another page

Predefined Process: indicates a process defined elsewhere

Example: A certain company plan to give a 10% bonus to each of its employees at the end of every year. If an employee has been working 10/more years at the company, she/he is to get an additional birr 100. Draw a flowchart of an algorithm to calculate and print the bonus for a given employee.

Solution:

Analysis: the problem is to compute bonus for employee. To do so, the salary and number of service years of the employee must be known. Let the salary be denoted by SAL, service be denoted by YEAR and bonus by BONUS. To compute the bonus, we multiply the salary by 10% and assign the result to BONUS. Thus, BONUS = SAL * 0.10. If the employee has served at least 10 years, birr 100 is added to the computed bonus of the employee and the result is displayed. Otherwise the originally computed bonus is displayed.

Algorithm: uses English to write operations in a group.

1. Read employee's salary and year of service
2. Calculate the employee's bonus
3. If years of service is greater than or equal to 10 then increase the bonus.
   Otherwise only print the bonus

Eg.1. A flowchart for calculating bonus

2) A flow chart for calculating interest amount

3) A program that identifies a larger and smaller number from two numbers

4) Calculate grade for ten students based on the scale:

Pseudo-code:

It is much similar to real code. We use verbs to write pseudo-code. Capitalize important words that show actions.

E.g. 1) A pseudo-code to calculate interest rate

```
ACCEPT Name, Principal, Rate
Interest=Principal X Rate
DISPLAY Name, Interest
```

2) A pseudo-code to calculate bonus

```
ACCEPT Name, Salary
Bonus=SalaryX0.1
ACCEPT Serviceyear
IF Serviceyear>10 Then
   Bonus+100
ENDIF
DISPLAY Bonus
```

3) A pseudo-code that calculates grade

```
ACCEPT Mark, Name
IF Mark>80 Then
   Grade←A
ELSE IF Mark>=70 Then
   Grade←B
ELSE IF Mark>=60 Then
   Grade←C
ELSE IF Mark>=50 Then
   Grade←D
ELSE
   Grade←F
ENDIF
DISPLAY Grade, Name
```

Most algorithms incorporate three programming conventions

- Sequence
- Decision and

- Repetition

Sequence

```
STOCK=STOCK+ QUANTITY
```

Decision

```
IF HOURS>40
    THEN
        PERFORM OVERTIME
    ELSE
        PERFORM REGULAR
ENDIF
```

Repetition

```
INITIALIZE SUM TO ZERO
WHILE SUM<100
    GET DATA
    ADD DATA TO SUM
ENDWHILE
```

1.6. The C++ Compilation Process

C++ is a 3rd-generation language. These types of languages must be translated into a machine language in order to be executed by a CPU. The process of translating high-level language into machine language is called the compilation process.

The compilation process consists of the following steps.

Program source code is entered into a file using a text editor. After the code has been entered, a compiler program is started that translates the source into an object code file. The object code file is linked with other object code files that come with the compiler and an executable file (or program) is created. The linker "combines" all object code files into an executable file. Typically, the object files created by your source files are linked with object files that are packaged into libraries. In order to execute the program, a program called the loader copies the executable file into the memory of the computer and sends an execute command to the CPU. It should be noted that errors can occur during each step.

1.7. The syntax and semantics of programming languages

Syntax

The Syntax of a programming language consists of the rules for the correct use of the language. This involves the correct grammatical construction and arrangement of the language, correct spelling, hyphenation, inflection and so on. The syntax of a programming language has to be strictly adhered to.

Semantics

The semantics of a programming language deal with the meanings given to syntactically correct constructs of the language. Usually, semantics is defined in terms of the program's run-time behavior. What happens when the program is executed with a certain set of inputs, what statements are executed, what values are assigned to the variables, and what output is produced? Thus syntax has nothing to do with "meaning" or run-time behavior of a program. A program could be syntactically correct though it is meaningless.

1.8. Errors in programming language

There are different kinds of errors that encounter in writing a program:

- Compile-time errors: identified by the compiler, includes
  - missing brackets
  - undeclared variables (e.g. misspelling variable name in one place)
  - incorrect use of single or double quotes
  - invalid identifier names
- Run-time errors: attempts by the program to do something illegal while executing, can include
  - trying to divide by zero (will crash)

- using or printing a variable value before initializing it (usually will not crash, just give very weird results)

– Logic errors: the implementation doesn't correctly solve the problem, includes any form of miscalculation or incorrect ordering of instructions.

Worksheet 1

For each of the problems below, develop a flow chart

1. Receive a number and determine whether it is odd or even.
2. Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
3. Receive 3 numbers and display them in ascending order from smallest to largest
4. Add the numbers from 1 to 100 and display the sum
5. Add the even numbers between 0 and any positive integer number given by the user.
6. Find the average of two numbers given by the user.
7. Find the average, maximum, minimum, and sum of three numbers given by the user.
8. Find the area of a circle where the radius is provided by the user.
9. Swap the contents of two variables using a third variable.
10. Swap the content of two variables without using a third variable.
11. Read an integer value from the keyboard and display a message indicating if this number is odd or even.
12. Read 10 integers from the keyboard in the range 0 - 100, and count how many of them are larger than 50, and display this result.
13. Take an integer from the user and display the factorial of that number

Chapter 2

2. Basics concepts of C++ programming

2.1 Structure of C++ program

Structure of a C++ program includes:

- ✓ **Documentation section**
- ✓ **Linking section**
- ✓ **Definition section**
- ✓ **Global declaration section**
- ✓ **Class declaration**
- ✓ **Function**
- ✓ **main() functions**
  {
  Initializations;
  Executable parts;
  }
- ✓ **Function1()**
  {
  Initializations;
  Executable parts;
  }…n

- **Documentation Section:** include comment parts, which are ignored by compiler. The comments are used to describe the functionality of each statement in the program, its copyright, author, and date of compilation and purpose of the program to the user. C++ supports single line (// comment) and multiline (/* comments*/).

  Syntax: //This statement is a single line comment

  /*this is multiline comment*/

- **Link section:** here we can link the necessary files and libraries to current files. Using #include directive we can link the necessary libraries to current files.

  Syntax: #include<file or library name>

  Example: #include<iostream.h>

- **Definition Section:** It is possible to define symbolic constants. Symbolic constants are normal identifiers which value cannot be altered.

  Syntax: #define identifier constant

- **Global declaration Section:** variables declared under this section are available throughout the program.
- **Class declaration Section:** defines the class declaration.
- **Functions:** The function main, which has been written in the program, is an example of function. A function is defined as group of instructions, which are assigned a name and accessed by the name.

Syntax:

```
return type function name
()
        {
Statement of function;
        }
```

- ✓ A function is identified by the compiler with the help of **parenthesis ( )** after the function name. Without the parenthesis the compiler thinks it as a variable or complaints that it is undefined symbol. So to define a function, parenthesis is must.
- ✓ The statement inside the brace {} forms the body of the function and specifies the task of the function. Return type specifies the type of data the function has to send to the calling function after the execution. It may be the result of the task performed in the function.

- **Main ( ) function:** the execution of the program starts from main function. Every program must have only one main function. All other functions are called either directly or indirectly from main. The initialization or other executable statements are included with in the main function.
- **Initialization part:** the variables that are used in the program should be initialized here. These variables are not available to all the functions. In C++ we can initialize the variables at any part in the function.
- **Executable part:** A statement declared under executable part performs some tasks. Every statement under global declaration, initialization and executable parts should be terminated with the semi colon ';'. A semi colon acts as a statement terminator like full stop in English.

Syntax:

```
#include<iostream.h>
Int main ()
{
Statements;
Other functions;
}
```

2.2 A simple C++ program

Look the program given below:

```
//This program accepts to integer numbers and prints out their sum.

#include<iostream.h>
Int main ()
{
int a,b,sum;
cout<<"Enter the value of a and b \n";
cin>>a>>b;
sum=a+b;
cout<<"\n Sum="<<sum;
}
```

- **#include<iostream.h>:** this statement is used to include a library file, which contain**s** the necessary built in functions that are used in the program. The iostream.h is a library file which have the built in function for **input** and **output** operations like **cin** and **cout** respectively in above example. The < and > angle bracket are not a part of a file name, but are used to indicate that this is a standard C++ library file.
- **Int:** the **int** specifies the function need not return any value. In C++ every caller function expects a return from the called function. If you specify a 'int' before the function, the function does not expect any return from the called function.
- **{…}:** we should specify the body of the function in between the open and close flower braces.
- **cout:** It is an output statement available in iostream.h library file, used to display message .It is used for prompting message and displaying the values of the variable. To use cout you must include the following directive at the beginning of your program

  #include <iostream.h>

  The basic form for writing output to the screen is

  Syntax:

  cout<<"the statement to be displayed"; (for displaying message) or

  cout<< the name of the variable to be displayed; (for displaying the value of a variable)
- **cin:** It is an input statement available in iostream.h library file, used to accept values of a variable.

  Syntax:

  cin>>the name of the variable which its' value is going to be accepted;
- **int a,b,sum;** is a declaration of an integer variables.

Special characters

There are a variety of characters in C++ that are used for special purposes, such as single-quotes, double-quotes, backslashes, etc. To include one of these as a "regular" character in an output stream, or to assign them as a character value, we have to let the compiler know that we don't want the "special" use of the character. To do so, we use an escape sequence - which usually simply means preceding the desired character with a backslash \

Some of the common escape sequences are listed below:

Escape sequence	Meaning
\n	newline (like endl)
\t	tab
\\	backslash
\'	single quote
\"	double quote
\nnn	use ASCII value

For example:

```
cout << "Here comes a double-quote: \" there it went ... " << endl;
```

The ASCII value for a character is an integer interpretation of the bit pattern used to store the character.

For example, the ASCII values for the characters 'A' through 'Z' are 65 through 90.

Syntax and Semantics

The Syntax of a programming language consists of the rules for the correct use of the language. This involves the correct grammatical construction and arrangement of the language, correct spelling, hyphenation, inflection and so on. The syntax of a programming language has to be strictly adhered to.

Character set: The character set of a language is simply that set of symbols from which programming languages are composed. Because the character set is a fundamental element of programming languages, especially of data representation in computers efforts were made to generate standards. As a result two alphabets have emerged to become the de facto standards.

Semantics

The semantics of a programming language deal with the meanings given to syntactically correct constructs of the language. Usually, semantics is defined in terms of the program's run-time behavior: What happens when the program is executed with a certain set of inputs, what statements are executed, what values are assigned to the variables, and what output is produced.

Thus syntax has nothing to do with "meaning" or run-time behavior of a program. A program could be syntactically correct yet meaningless. The program below (code fragment) is syntactically correct but does not have any meaning at runtime (never terminates).

```
sum=0;
while (sum!=-1)
sum=sum+10;
```

Yet syntax is a prerequisite to meaningful expression. Thus, a programming language must have a good syntactic definition before it can properly support the development of meaningful programs.

2.3 Data Types, Variables, and Constants

A. Data types

Every operation in the program manipulates with different types of data. Data can be integer, a fraction or a string. C++ provides you to define a variable, which can hold different types of data.

B. Variable

A variable is a location to store data. A variable is a symbolic name that holds different data types of data. It represents a place in the memory where particular data is stored. To access the data, the name of the variable is referred.

Most of the languages support the common data type variables viz. character type, integer type, and real numbers. C++ also supports all the basic data types besides user defined data types.

The data types in C++ are classified as:

- **Primary**
- **User defined(structures, Unions, Classes, enumerated)**
- **Derived(Arrays, Functions and pointers)**

Primary data set

It is classified by two ways.

1. **Integral type**
   a. **Integer (int)**: is used to declare the integer variables. Integer variables are used to store the natural numbers like 1,202, 343 …etc. we can define this type of variables using **int**. The integer type variable occupies 2 bytes of memory.

      Syntax:

      int variablename;

   b. **Character (char):**it is used to declare the character variable. Character variable are used to store characters like 'A', 'X', '4'…etc. We can define character through the key word **char** and followed by the character variable. These types occupy one byte space in the memory.

      Syntax:

      Char variablename;

2. **Floating point type:**

   It stores numbers with decimal places. In integer we can't stores numbers with fractional part. In such cases where fractional parts are necessary float serves good purpose. This data type is useful when you want to store length, width and salary etc.

   Fundamentally C++ supports 2 types of floating point types.

   a. **float**: It is used to declare the real variables(real numbers). Like 1.23, 3.05…etc. It has 7 digits of precision. It occupies 4 bytes of memory.

      Syntax:

      float real_variable;

   b. **double:** it occupies 8 bytes of memory twice as much as type float. Stores floating point numbers with much larger range and precision. It is used when the value of a floating number more than that held by float data type.

      Syntax:

      double real_variable;

Range of primary data set

Type	Size	Range
Int	2 bytes	-32768 to +32767
char	1 byte	-128 to +127
float	4 bytes	$3.4x10^{-38}$ to $3.4x10^{+38}$
double	8 bytes	$1.7x10^{-308}$ to $1.7x10^{+308}$

In primary data set the range is limited as defined .If we need to access the values out of the range it is possible through the type modifiers.

Modifiers

We have two sets of modifiers.

- Long and short
- Signed and unsigned

Type	Size	Range
long int	4 bytes	-2147483648 to +2147483648
long double	10 bytes	$3.4x10^{-4932}$ to $3.4x10^{+4932}$
unsigned int	2 bytes	0 to 65535
unsigned char	1 byte	0 to 255
unsigned short int	2 byte	0 to 65535
unsigned long int	4 byte	0 to 4294967295

Variable naming

- *The name of the variable needs to be meaningful, short and should not contain any embedded space or symbols like?!@#$%^&*(){}[ ] . , ; ”/\. However, underscores can be used wherever a space is required; for example, Basic_salary.*
- *No two variables should have the same name; for example, to accept four numbers, one variable name cannot be used rather four different variables need to be used.*
- *A variable name must begin with an alphabet, which may be followed by a sequence of alphabets or digits (0-9)*
- *Keywords or reserved words cannot be used as variable names*

Constants

Constants are data elements whose values do not change during program execution. It is any expression that has a fixed value. They are of two types:

- **Alphanumeric constants**: comprises of A-Z, a-z, 0-9, symbols like!@#$ %^&*() {}[ ] . , ; : ’ ” / \. Each letter, symbol or number used is called a character. It cannot be used for arithmetic calculations. A set of character is called string. Strings are enclosed with double quotes where as characters are enclosed with single quotes.

- **Numeric constants:** are of two type
  - Integer: whole numbers
  - Float : numbers with decimals

2.4 Keywords and Identifiers

I. Keywords

Certain words are reserved by C++ for specific purposes and may not be used as identifiers. These are called reserved **words** or **keywords.**

The keywords used in C++ are summarized in the table given below:

Asm	continue	Float	new	signed	try
break	delete	friend	private	static	union
Case	Do	Goto	protected	struct	unsigned
Catch	Double	If	public	switch	virtual
Char	Else	Inline	register	template	int
Class	Enum	Int	return	this	volatile
Constant	Extern	Long	short	throw	while

II. Identifiers

Programming languages use names to refer to the various entities that make up a program. Names are a programming convenience, which allow the programmer to organize what would otherwise be quantities of plain data into a meaningful and human-readable collection. As a result, no trace of a name is left in the final executable code generated by a compiler. For example, a temperature variable eventually becomes a few bytes of memory which is referred to by the executable code by its address, not its name.

C++ imposes the following rules for creating valid names (also called identifiers). A name should consist of one or more characters, each of which may be a letter (i.e., 'A'-'Z' and 'a'-'z'), a digit (i.e., '0'-'9'), or an underscore character ('_'), except that the first character may not be a digit. Upper and lower case letter are distinct. For example:

```
salary      // valid identifier
salary2     // valid identifier
2salary     // invalid identifier (begins with a digit)
salary      // valid identifier
Salary      // valid but distinct from salary
```

C++ imposes no limit on the number of characters in an identifier. However, most implementation do.

But the limit is usually so large that it should not cause a concern (e.g., 255 characters).

2.5 Operators

An operator is a symbol that operates on one or more expressions producing a value that can be assigned to a variable. Till now we have dealt with 3 operators

'<<' output operator used with cout [insertion operator]

'>>' input operator used with cin [extraction operator]

The operators used in C++ are discussed below:

1.5.1. Arithmetic Operator

C++ provides five basic arithmetic operators.

Operator	Name	Example
+	Addition	10+2 //gives 12
-	Subtraction	10-2 //gives 8
*	Multiplication	10*2 //gives 20
/	Division	10/2 //gives 5
%	Remainder	10%2 //gives 0

Except for remainder (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are real then the result will be a real (or double to be exact). When both operands of the division operator (/) are integers then the division is performed as an integer division and not the normal division we are used to. Integer division always results in an integer outcome (i.e., the result is always rounded down).

For example:

```
9 / 2        // gives 4, not 4.5!
-9 / 2       // gives -5, not -4!
```

Unintended integer divisions are a common source of programming errors. To obtain a real division when both operands are integers, you should cast one of the operands to be real:

```
int  cost = 100;
int  volume = 80;
double  unitPrice = cost / (double) volume;   // gives 1.25
```

The remainder operator (%) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example 13%3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1. It is possible for the outcome of an arithmetic operation to be too large for storing in a designated variable. This situation is called an overflow. The outcome of an overflow is machine-dependent and therefore undefined. For example:

```
unsigned char k = 10 * 92;  // overflow: 920 > 255
```

It is illegal to divide a number by zero. This results in a run-time division-by-zero failure which typically causes the program to terminate.

2.5.1 Relational operators

Relational operators are used to test the relationship between two variables, or between a variable and a constant.

Operator	Name	Example
==	Equal to	4==4 //gives 1
!=	Not equal to	4!=4 //gives 0
>=	Greater than or equal to	4>=4 //gives 1
<=	Less than or equal to	4<=4 //gives 1
>	Greater than	4>4 //gives 0
<	Less than	4<4.1 //gives 1

2.5.2 Logical Operators

C++ provides three logical operators for combining or negates logical expression.

Operator	Name	Example
!	Logical negation	!(3==3) //gives 0
&&	Logical and	3<4 && 6<6 //gives 0
||	Logical or	3<4 && 6<6 //gives 1

The compound condition combined with the AND(&&) operator evaluates to true only if all the individual conditions evaluate to true. The logical OR(||) is use when at least one of the conditions must be true in order to satisfy the compound condition. The logical NOT(!) reverses the result of a condition.

2.5.3 Increment or decrement operator

The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.

```
int k=5;
```

Operator	Name	Example
++	Auto increment(prefix)	++k+5// gives 11
++	Auto increment(postfix)	K++ +5// gives 10
--	Auto decrement(prefix)	--k+5 //gives 9
--	Auto decrement(postfix)	k-- +5// gives 10

Prefix increment and pre decrement operator increments or decrements a variable and executes the statement. Postfix increment and decrement operator increments or decrements a variable after the variable is executed.

2.5.4 Assignment operators

The assignment operator is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue. An **lvalue** (standing for **l**eft **value**) is anything that denotes a memory location in which a value may be stored. The only kind of lvalue we have seen so far in this book is a variable.

Operator	Example	Equivalent to
=	N=5	
+=	N+=5	N=N+5
-=	N-=5	N=N-5
*=	N*=5	N=N*5
/=	N/=5	N=N/5
%=	N%=5	N=N%5

2.4 Debugging and programming errors

When we attempt to produce an efficient program we should take sufficient care to maintain clarity and readability of a program. The program errors are called **bugs.** The art of locating and eliminating bugs or errors is called **debugging.**

Types of Programming bugs or errors

1. **Compile errors**: are errors indicating program syntax errors, disk or memory access errors and command line errors. For example
    - missing expected) , (,{,} and comma
    - Array bounds missing], array must have at least one element; array of references is not allowed and array size too large.
2. **Linker errors:** Errors that occur when a function is miss spelled, hexadecimal overflow etc. Unable to find the required object files in the current directory to link etc. That is when main as maid or cout is spelled as cuot and so on.
3. **Run time errors:** errors that occur at run time. It is a process where the program is compiled, linked and executed. Errors that may occur at execution time such as divide errors, stack over flow, null pointer termination etc. Result is not as expected due to logical errors in the program.

Chapter Three

Data Types, variables and Constants

Data types

Every operation in the program manipulates with different types of data. Data can be integer, a fraction or a string. C++ provides you to define a variable, which can hold different types of data.

Variable

A variable is a location to store data. A variable is a symbolic name that holds different types of data. It represents a place in the memory where particular data is stored. To access the data, the name of the variable is referred.

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.

Most of the languages support the common data type variables. Character type, integer type, and real numbers. C++ also supports all the basic data types besides user defined data types.

Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

bool	For variable to store boolean values(True or False)
char	For variables to store character types.
int	for variable with integral values
float and **double** are also types for variables with large and floating point values	

The sizeof operator

As we know that different types of Variables, constant, etc. require different amounts of memory to store them The sizeof operator can be used to find how many bytes are required for an object to store in memory. the sizeof operator determines the amount of memory required for an object at compile time rather than at run time.

//example

```
#include <iostream>
using namespace std;
int main() {
 cout << "Size of char is " << sizeof(char) << endl;
 cout << "Size of int is " << sizeof(int) << endl;
 cout << "Size of float is " << sizeof(float) << endl;
 cout << "Size of short int is " << sizeof(short int) << endl;
 cout << "Size of long int is " << sizeof(long int) << endl;
 cout << "Size of double is " << sizeof(double) << endl;
 cout << "Size of wchar_t is " << sizeof(wchar_t) << endl;
 return 0;
}
```

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at

the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

Example:

```
int i;      // declared but not initialized
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i; // declaration
i = 10; // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10; //initialization and declaration in same step
int i=10, j=11;
```

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;
i=10;
j=20;
int j=i+j;/*compile time error, cannot redeclare a variable in same scope*/
```

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases it's between the curly braces, in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which are once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the main() function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

Example: Only declared, not initialized

```
include <iostream>
using namespace std;
int x; // Global variable declared
int main()
{
 x=10; // Initialized once
 cout <<"first value of x = "<< x;
 x=20; // Initialized again
 cout <<"Initialized again with value = "<< x;
}
```

Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example:

```
include <iostream>
using namespace std;
int main ()
{
 int i=10;
 if(i<20) // if condition scope starts
  {
```

```
    int n=100; // Local variable declared and initialized
  }      // if condition scope ends
 cout << n;// Compile time error, n not available here
}
```

Types of Data Types

The data types in C++ are classified as

- Primary
- User defined (structures, Unions, Classes, enumerated)
- Derived (Arrays, Functions and pointers)

Primary data set

It is classified by two ways.

1. Integral type

Integer (int): is used to declare the integer variables. Integer variables are used to store the natural numbers like 1,202, 343 …etc. we can define this type of variables using **int**. The integer type variable occupies 2 bytes of memory.

Syntax: `int variablename;`

Character (char): it is used to declare the character variable. Character variable are used to store characters like 'A', 'X' …etc. We can define character through the key word **char** and followed by the character variable. These types occupy one byte space in the memory.

Syntax: `Char variablename;`

2. Floating point type:

It stores numbers with decimal places. In integer we can't stores numbers with fractional part. In such cases where fractional parts are necessary float serves good purpose. This data type is useful when you want to store length, width and salary etc.

Fundamentally C++ supports 2 types of floating point types.

float: It is used to declare the real variables (real numbers). Like 1.23, 3.05…etc. It has 7 digits of precision. It occupies 4 bytes of memory.

Syntax: `float real_variable;`

double: it occupies 8 bytes of memory twice as much as type float. Stores floating point numbers with much larger range and precision. It is used when the value of a floating number more than that held by float data type.

Syntax: `double real_variable;`

Range of primary data set

Type	Size	Range
Int	2 bytes	-32768 to +32767
char	1 byte	-128 to +127
float	4 bytes	$3.4x10^{-38}$ to $3.4x10^{+38}$
double	8 bytes	$1.7x10^{-308}$ to $1.7x10^{+308}$

In primary data set the range is limited as defined. If we need to access the values out of the range, it is possible through the type modifiers.

Modifiers

We have two sets of modifiers.

- Long and short
- Signed and unsigned

Type	Size	Range
long int	4 bytes	-2147483648 to +2147483648
long double	10 bytes	$3.4x10^{-4932}$ to $3.4x10^{+4932}$
unsigned int	2 bytes	0 to 65535
unsigned char	1 byte	0 to 255
unsigned short int	2 byte	0 to 65535
unsigned long int	4 byte	0 to 4294967295

Below mentioned are some important points you must know about the modifiers,

- **long** and **short** modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of **short**.

- Size hierarchy: short int < int < long int
- Size hierarchy for floating point numbers is: float < double < long double
- **long float** is not a legal type and there are no **short floating point** numbers.
- **Signed** types includes both positive and negative numbers and is the default type.
- **Unsigned**, numbers are always without any sign, that is always positive.

Variable naming

- The name of the variable needs to be meaningful, short and should not contain any embedded space or symbols like?!@#$%^&*(){}[ ] . , ; ”/\. However, underscores can be used wherever a space is required; for example, Basic_salary.
- No two variables should have the same name; for example, to accept four numbers, one variable name cannot be used rather four different variables need to be used.
- A variable name must begin with an alphabet, which may be followed by a sequence of alphabets or digits (0-9)
- Keywords or reserved words cannot be used as variable names.

Constants

Constants are data elements whose values do not change during program execution. It is any expression that has a fixed value. They are of two types:

Alphanumeric constants: comprises of A-Z, a-z, 0-9, symbols like!@#$ %^&*() {}[ ] . , ; : ’ ” / \. Each letter, symbol or number used is called a character. It cannot be used for arithmetic calculations. A set of character is called string. Strings are enclosed with double quotes whereas characters are enclosed with single quotes.

Numeric constants: are of two type

- Integer: whole numbers
- Float: numbers with decimals

A variable which does not change its value during execution of a program is known as a constant variable. Any attempt to change the value of a constant will result in an error message. A constant in C++ can be of any of the basic data types, const qualifier can be used to declare constant as shown below:

```
const float PI = 3.1415;
```

The above declaration means that PI is a constant of float types having a value 3.1415.

Examples of valid constant declarations are:

```
const int RATE = 50;
const float PI = 3.1415;
const char CH = 'A';
```

Keywords and Identifiers

Keywords

Certain words are reserved by C++ for specific purposes and may not be used as identifiers. These are called reserved **words** or **keywords.**

The keywords used in C++ are summarized in the table given below:

asm	continue	float	new	signed	try
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
Catch	double	if	public	switch	virtual
Char	else	inline	register	template	void
Class	enum	int	return	this	volatile
constant	extern	long	short	throw	while

Identifiers

Programming languages use names to refer to the various entities that make up a program. Names are a programming convenience, which allow the programmer to organize what would otherwise be quantities of plain data into a meaningful and human readable collection. As a result, no trace of a name is left in the final executable code generated by a compiler. For example, a temperature variable eventually becomes a few bytes of memory which is referred to by the executable code by its address, not its name.

C++ imposes the following rules for creating valid names (also called identifiers).

A name should consist of one or more characters, each of which may be a letter (i.e., 'A'-

'Z' and 'a'-'z'), a digit (i.e., '0'-'9'), or an underscore character ('_'), except that the first character may not be a digit. Upper and lower case letter are distinct. For example:

```
salary     // valid identifier
salary2    // valid identifier
2salary    // invalid identifier (begins with a digit)
_salary    // valid identifier
Salary     // valid but distinct from salary
Emplolyeesalary       // valid identifier
Employee_Salary      // valid identifier
Employee Salary   // invalid identifier
```

Very important: The C++ language is **case sensitive**, that means an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example the variable **RESULT** is not the same as the variable **result** nor the variable **Result**.

Operators

An operator is a symbol that operates on one or more expressions producing a value that can be assigned to a variable. Till now we have dealt with 3 operators

'<<' output operator used with cout [insertion operator]

'>>' input operator used with cin [extraction operator]

The operators used in C++ are discussed below:

Arithmetic Operator

C++ provides five basic arithmetic operators.

Operator	Name	Example
+	Addition	10+2 //gives 12
-	Subtraction	10-2 //gives 8
*****	Multiplication	10*2 //gives 20
/	Division	10/2 //gives 5
%	Remainder	10%2 //gives 0

Except for remainder (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are real then the result will be a real (or double to be exact). When both operands of the division operator (/) are integers then the division is performed as an integer division and not the normal division we are used to. Integer division always results in an integer outcome (i.e., the result is always **rounded down**).

For example: **`9 / 2`** **`// gives 4, not 4.5!`**

`-9 / 2` **`// gives -5, not -4!`**

Unintended integer divisions are a common source of programming errors. To obtain a real division when both operands are integers, you should cast one of the operands to be real:

```
int cost = 100;
int volume = 80;
double unitPrice = cost / (double) volume; // gives 1.25
```

The remainder operator (%) expects integers for both of its operands. It returns the remainder of integer-dividing the operands. For example 13%3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1. It is possible for the outcome of an arithmetic operation to be too large for storing in a designated variable. This situation is called an overflow. The outcome of an overflow is machine-dependent and therefore undefined.

Example: **`unsigned char k = 10 * 92; // overflow: 920 > 255`**

It is illegal to divide a number by zero. This results in a run-time division-by-zero failure which typically causes the program to terminate.

Relational operators

These operators establish a relationship between operands. The relational operators are: less than (<) , greater than (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).

Relational operators are used to test the relationship between two variables, or between a variable and a constant.

You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any variable, whereas equivalent operator is used to compare values, like in if-else conditions, Example

```
int x = 10; //assignment operator
x=5; // again assignment operator
if(x == 5) /* here we have used equivalent relational operator,
for comparison */
{
 cout <<"Successfully compared";
}
```

Operator	Name	Example
==	Equal to	4==4 //gives 1
!=	Not equal to	4!=4 //gives 0
>=	Greater than or equal to	4>=4 //gives 1
<=	Less than or equal to	4<=4 //gives 1
>	Greater than	4>4 //gives 0
<	Less than	4<4.1 //gives 1

Logical Operators

C++ provides three logical operators for combining or negates logical expression. The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision

making.

Operator	Name	Example
!	Logical negation	!(3==3) //gives 0
&&	Logical and	3<4 && 6<6 //gives 0
||	Logical or	3<4 || 6<6 //gives 1

The compound condition combined with the AND (&&) operator evaluates to true only if all the individual conditions evaluate to true. The logical OR (||) is use when at least one of the conditions must be true in order to satisfy the compound condition. The logical NOT (!) reverses the result of a condition.

Increment or decrement operator

The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable.

```
int k=5;
```

Operator	Name	Example
++	Auto increment(prefix)	++k+5// gives 11
++	Auto increment(postfix)	K++ +5// gives 10
--	Auto decrement(prefix)	--k+5 //gives 9
--	Auto decrement(postfix)	k-- +5// gives 10

Prefix increment and pre decrement operator increments or decrements a variable and executes the statement. Postfix increment and decrement operator increments or decrements a variable after the variable is executed.

Assignment operators

The assignment operator is used for storing a value at some memory location (typically denoted by a variable). Its left operand should be an lvalue, and its right operand may be an arbitrary expression. The latter is evaluated and the outcome is stored in the location denoted by the lvalue. An **lvalue** (standing for **l**eft **value**) is anything that denotes a memory location in which a value may be stored. The only kind of lvalue we have seen so

far in this lecture note is a variable.

Operator	Example	Equivalent to
=	N=5	
+=	N+=5	N=N+5
-=	N-=5	N=N-5
*=	N*=5	N=N*5
/=	N/=5	N=N/5
%=	N%=5	N=N%5
&=	N&=n	N=N&n
|=	N|=n	N=N|n
^=	N^=n	N=N^n
<<=	N<<=3	N=N<<3
>>=	N>>=3	N=N>>3

Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. Operators in higher levels take precedence over operators in lower levels.

Precedence Table:

Level	Operator	Order
Highest	sizeof()++ -- (pre fix)	Right to left
	* / %	Left to right
	+ -	Left to right
	< <= > >=	Left to right
	== !=	Left to right
	&&	Left to right
	||	Left to right

	? :	Left to right
	= ,+=, -=, *=, /=,^= ,%=, &= ,|= ,<<= ,>>=	Right to left
	++ -- (postfix)	Right to left
	,	Left to right

Example:

a = = b + c * d

c * d is evaluated first because * has a higher precedence than + and = =.

The result is then added to b because + has a higher precedence than = =

And then == is evaluated.

- Precedence rules can be overridden by using brackets.

E.g. Rewriting the above expression as:

a = = (b + c) * d causes + to be evaluated before *.

- Operators with the same precedence level are evaluated in the order specified by the column on the table of precedence rule.

E.g. a = b += c, The evaluation order is right to left, so the first b += c is evaluated followed by a = b.

Chapter Four

Decision Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

- A running program spends all of its time executing instructions or statements in that program.
- The order in which statements in a program are executed is called **flow** of that program.
- Programmers can control which instruction to be executed in a program, which is called **flow contro**l.
- This term reflects the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement.
- Flow control in a program is typically **sequential**, from one statement to the next.
- But we can also have execution that might be divided to other paths by **branching statements**. Or perform a block of statement repeatedly until a condition fails by **Repetition** or **looping**.

Flow control is an important concept in programming because it will give all the power to the programmer to decide what to do to execute during a run and what is not, therefore, affecting the overall outcome of the program.

Sequential Statements

Such kinds of statements are instruction in a program which will be executed one after the other in the order scripted in the program. In sequential statements, the order will be determined during program development and cannot be changed.

Selection Statements

Selection statements are statements in a program where there are points at which the program will decide at runtime whether some part of the code should or should not be executed. There are two types of selection statements in C++, which are the "**if statement**" and the "**switch statement**".

No	Statement & Description
1	**if statement** An 'if' statement consists of a boolean expression followed by one or more statements.
2	**if...else statement** An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
3	**switch statement** A 'switch' statement allows a variable to be tested for equality against a list of values.
4	**nested if statements** You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
5	**nested switch statements** You can use one 'switch' statement inside another 'switch' statement(s).

The if Statement

- It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied.
- The different forms of the 'If" statement will be used to decide whether to execute part of the program based on a condition which will be tested either for TRUE or FALSE result.

- The different forms of the "If" statement are:
  - The simple if statement
  - The If else statement
  - The if else if statement

The simple if statement

- The simple if statement will decide only one part of the program to be executed if the condition is satisfied or ignored if the condition fails.
- The General Syntax is:

```
if (expression)
statements;
```

- In any **if** statement, first the **expression** will be evaluated and if the outcome is non zero (which means TRUE), then the **statements** is executed. Otherwise, nothing happens (the statement will not be executed) and execution resumes to the line immediately after the **if** block.
- To make multiple statements dependent on the same condition we can use a compound statement, which will be implemented by embracing the group of instructions within the left "{" and right "}" French bracket.

Example:

```
            if(age>18)
                  cout<<" you are an adult";
```

Example:

```
if (balance > 0)
  {
            interest = balance * creditRate;
            balance += interest;
  }
```

- Most of the time "expression" will have relational expressions testing whether something is equal, greater, less, or different from something else.
- It should be noted that the output of a relational expression is either True (represented by anything different from zero) or False (represented by Zero).

- Thus any expression, whose final result is either zero or none zero can be used in" expression".

Example:

```
      int x;
    cin>>x;
    if(x)
    cout<<"You are an adult";
```

In the above example, the value of variable x will be an input from the user. The "if" statement will be true if the user gives anything different from zero, which means the message "**you are an adult**" will be displayed on the screen. If the user gives zero value for x, which will be interpreted as False, nothing will be done as the if statement is not satisfied. Thus, expression can be:

- Relational expression,
- A variable,
- A literal constant, or
- Assignment operation, as the final result is whatever we have at the right hand side and the one at the left hand side is a variable.

The If else statement

- Another form of the "if" is the "if …else" statement.
- The "if else" statement allows us to specify two alternative statements:
  - One which will be executed if a condition is satisfied and
- Another which will be executed if the condition is not satisfied.
- The General Syntax is:

```
if (expression)
     statements1;
else
     statements2;
```

First "expression" is evaluated and if the outcome is none zero (true), then "statements1" will be executed. Otherwise, which means the "expression" is false "statements2" will be executed.

Example:

```
                    if(balance > 0)
{
     interest = balance * creditRate;
     balance += interest;
}
else
{
     interest = balance * debitRate;
     balance += interest;
}
```

Example 1:

```
if(age>18)
     cout<<" You are an adult";
else
    cout<<" You are a kid";
```

Example 2:

```
int x;
cout<<" Enter a number: ";
cin>>x;
if(x%2==0)
       cout<<" The Number is Even";
else
       cout<<" The Number is Odd";
```

Example 3:

```
int x;
cout<<" Enter a number: ";
cin>>x;
if(x%2)
       cout<<" The Number is Odd";
else
       cout<<" The Number is Even";
```

- The above three examples illustrate the "if else" statement.
- The last two examples will do the same thing except that the expression in the Example 3 is changed from relational to arithmetic. We know that the result from the modulus

operator is either 0 or one if the divider is 2. Thus the final result of the expression in Example 3 will either be 1 for odd numbers or be 0 for even numbers.

- 'if' statements may be nested by having an if statement appear inside another if statement. For instance:

```
if(callHour > 3)
{
	if(callDuration <= 5)
		charge = callDuration * tarif1;
	else
		charge = 5 * tarif1 + (callDuration - 5) * tarif2;
}
else
	charge = flatFee;
```

Example:

```
	#include<iostream.h>

	#include<conio.h>

	int main()

	{

		int testScore;

		cout<<"\nEnter your test score:";

		cin>>testScore;

		if(testScore < 70)

			cout<<"\n You did not pass:"; // then block

		else

			cout<<"\n You did pass"; //else block

		return 0;

	}
```

- The above sample program can be diagrammatically expressed as follows:

The "if else if" statement

- The third form of the "if" statement is the "if …else if" statement.
- The "if else if" statement allows us to specify more than two alternative statements each will be executed based on testing one or more conditions.
- The General Syntax is:

```
if (expression1)
statements1;
else if(expression2)
statements2;
.
.
.
else if(expressionN)
statementsN;
else
statements
```

First "expression1" is evaluated and if the outcome is none zero (true), then "statements1" will be executed. Otherwise, which means the "expression1" then "expression2" will be

evaluated: if the output of expression2 is True then "statements2" will be executed otherwise the next "expression" in the else if statement will be executed and so on.

If all the expressions in the "if" and "else if" statements are False then "statements" under else will be executed.

The "if…else" and "if…else if" statements are said to be exclusive selection as if one of the condition (expression) is satisfied only instructions in its block will be executed and the rest will be ignored.

Example:

```
       if(score >= 90)
       cout<< "\n your grade is A";
                    else if(score >= 80)
       cout<< "\n your grade is B";
                    else if(score >= 70)
       cout<< "\n your grade is C";
                    else if(score >= 60)
       cout<< "\n your grade is D";
                    else
       cout<< "\n your grade is F";
```

In the above example, only one of the five cout statements will be executed and the rest will be ignored. But until one of the conditions is satisfied or the else part is reached, the expressions will be tested or evaluated.

Nesting If statements within another if statement

- One or more if statements can be nested with in another if statement.
- The nesting will be used to test multiple conditions to perform a task.
- It is always recommended to indent nested if statements to enhance readability of a program
- The General Syntax might be:

```
if (expression1)
     {
        if (expression2)
              statementsN;
       else
```

```
          statementsM;
     }
else
    {
       if (expression3)
           statementsR;
      else
         statementsT;
   }
```

- StatementsN will be executed if and only if “expression1” and “expression2” are evaluated and if the outcome of both is none zero (TRUE).
- StatementsM will be executed if and only if “expression1” is TRUE and “expression2” is FALSE.
- StatementsR will be executed if and only if “expression1” is FALSE and “expression3” is TRUE.
- StatementsT will be executed if and only if “expression1” is FLASE and “expression2” is FALSE.

An example on nested if statements:

```
#...
int main ()
{
       int testScore, age;
       cout<<” \n Enter your test score:”;
       cin>>testScore;
       cout<< “\n enter your age:”;
       cin>>age;
       if (testScore >= 70)
       {
              if (age < 10)
                     cout<< “\n You did a great job”;
              else
                     cout<< “\n You did pass”;
       }
       else
              cout<< “\n You did not pass”;
       return 0;}
```

- The above program can be expressed diagrammatically as follows.

The Switch Statement

- Another C++ statement that implements a selection control flow is the switch statement (multiple-choice statement).
- The switch statement provides a way of choosing between a set of alternatives based on the value of an expression. The general form of the switch statement is:
- The switch statement has four components:
    - Switch
    - Case
    - Default
    - Break
- Where **Default** and **Break,** are Optional.
- The General Syntax might be:

```
switch(expression)
   {
      case constant1:
            statements;
            .
      case constant n:
            statements;
      default:
            statements;
    }
```

- Expression is called the **switch tag** and the constants preceding each case are called the **case labels**.
- The output of "expression" should always be a constant value.
- First expression is evaluated, and the outcome, which is a constant value, will compared to each of the numeric constants in the case labels, in the order they appear, until a match is found.
- Note, however, that the evaluation of the **switch tag** with the **case labels** is only for **equality**
- The statements following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement).
- After one case is satisfied, execution continues until either a **break** statement is encountered or all intervening statements are executed, which means until the execution reaches the right French bracket of the switch statement.
- The final default case is optional and is exercised if none of the earlier cases provide a match. This means that, if the value of the "expression" is not equal to any of the case labels, then the statements under **default** will be executed.
- Now let us see the effect of including a break statement in the switch statement.

Scenario One

```
switch (N){
      case 1: x=10;
      case 2: x=20;
      case 3: x=30;
}
Even if N is 1 or 2 x will
have 30
```

Scenario Two

```
switch(N){
     case 1: x=10; break;
     case 2: x=20; break;
     case 3: x=30; break;
}
X will have 10, 20 or 30 based on the
value of N
```

- The break terminates the switch statement by jumping to the very end of it.
- There are, however, situations in which it makes sense to have a case without a break.

  For instance:

```
// Program to built a simple calculator using switch Statement
#include <iostream.h>
int main()
{
    char o;
    float num1, num2;
    cout << "Enter an operator (+, -, *, /): ";
    cin >>o;
    cout << "Enter two operands: ";
    cin >> num1 >> num2;
    switch (o)
    {
        case '+':
```

```
            cout << num1 << " + " << num2 << " = " << num1+num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1-num2;
            break;
        case 'X':
        case '*':
            cout << num1 << " * " << num2 << " = " << num1*num2;
            break;
        case '/':
            cout << num1 << " / " << num2 << " = " << num1/num2;
            break;
        default:
      // operator is doesn't match any case constant (+, -, *, /)
            cout << "Error! operator is not correct";
            break;
    }
    return 0;
}
```

The - operator entered by the user is stored in o variable. And, two operands 2.3 and 4.5 are stored in variables num1 and num2 respectively.

Then, the control of the program jumps to

cout << num1 << " - " << num2 << " = " << num1-num2;

Finally, the break statement ends the switch statement. If break statement is not used, all cases after the correct case is executed.

Because case 'x' has no break statement (in fact no statement at all!), when this case satisfied, execution proceeds to the statements of the next case and the multiplication is performed.

Switch evaluates expression and compares the result to each of the case values.

Relational and Boolean operators can be used in switch tag if and only if the expected output is either 0 to represent False or 1 to represent True as that is the only possible output from such operators.

The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

```
//Another example for switch statement
#include <iostream>
using namespace std;
int main () {
   // local variable declaration:
   char grade = 'D';

   switch(grade) {
      case 'A' :
         cout << "Excellent!" << endl;
         break;
      case 'B' :
      case 'C' :
         cout << "Well done" << endl;
         break;
      case 'D' :
         cout << "You passed" << endl;
         break;
      case 'F' :
         cout << "Better try again" << endl;
         break;
      default :
         cout << "Invalid grade" << endl;
   }
   cout << "Your grade is " << grade << endl;
   return 0;}
```

Chapter-Three

Flow of Control

Introduction

- A running program spends all of its time executing instructions/statements.
- **Flow** of a program: the order in w/c statements are executed.
- **Flow control**: Programmers can control which instruction to be executed.
- Flow control in a program is typically **sequential**, from one statement to the next.
- Flow control is an important concept in programming b/c it will give all the power to the programmer to decide what to do to execute during a run and what is not.

Sequential Statements

- Are executed one after the other in the order.
- Are written in the source code.

Example:

```
#include<iostream.h>
     int main (){
     float x, y, z;
     cin>>x;
     y=x*10;
     z=y*y;
     cout<<y<<"\t"<<z;
     }
```

Selection Statements

- are statements in a program where there are points at which the program will decide at runtime whether some part of the code should or should not be executed.
- Two types of selection statements in C++.
  - **if statement**
  - **switch statement**

if Statement

- It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied.
- The different forms of the 'if" statement will be used to decide whether to execute part of the program based on a condition which will be tested either for TRUE or FALSE result.
- enables you to test for a condition.
- Different forms of the "if" statements are:
  - **simple if** statement
  - **if else** statement
  - **If…else if** statement

simple if statement

- decide only one part of the program to be executed if the condition is satisfied or ignored if the condition fails.

- Syntax:

  if (expression)
  statements;

- Example:

```
if (y > 0)
    y+=2;
    cout<<"y is equal to:"<<y;
```

if else statement

- if else statement is used when there are alternative statements:
- some which need to be executed when the condition holds and
- some which are to be executed when the condition does not hold.
- Syntax:

```
if (expression)
statements;
else
statements;
```

Example:

```
cout<<"Enter a number: ";
cin>>x;
if(x%2)
    cout<<"The Number is Odd";
else
    cout<<"The Number is Even";
```

if …else if statement

Syntax:

```
 if (expression1)
statement1;
else if (expression2)
statement2;
      .
      .
      .
else if(expressionN)
statementsN;
else
statements
```

Example:

```
if (x<0)
cout<<"x is Negative";
else if (x>0)
cout<<"x is Positive";
Else
cout<<"x is Zero";
```

Nested if statement

Syntax:

```
if (expression1)
   {
     if (expression2)
          statementsN;
    else
       statementsM;
   }
else
  {
    if (expression3)
       statementsR;
   else
     statementsT;
  }
```

Example:

```
if (1 <= day && day <= 7)
{
if (day == 1 || day == 7)
cout << "This is a weekend day";
else
cout << "This is a weekday";
}
else
cout << "Not a legal day";
```

Switch statement (1)

- Is multiple-choice statement.
- Provides a way of choosing between a set of alternatives based on the value of an expression.
- Has four components:
  - **Switch**
  - **Case**
  - **Default(optional)**
  - **Break(optional)**

Switch statement (2)

Syntax:

```
switch (expression)
{
case valueOne: statement;
break;
case valueTwo: statement;
break;
....
case valueN:
statement;
break;
default: statement;
}
```

Switch statement (3)

Example:

```
cout<<"Enter Day";
cin>>day;
switch(day)
{
case 1 : cout << "Sunday";
break;
case 2 : cout << "Monday";
break;
case 3 : cout << "Tuesday";
break;
case 4 : cout << "Wednesday";
break;
case 5 : cout << "Thursday";
break;
case 6 : cout << "Friday";
break;
case 7 : cout << "Saturday";
break;
default : cout << "Not
Aplicable!";
break;
}
```

Switch statement (4)

```
cout<<"Enter Grade";
cin>>grade;
switch(grade)
{
case 'A':
        cout<<"Excellent";
        break;
case 'B':
        cout<<"Very Good";
        break;
case 'C':
        cout<<"Fair";
        break;
default:
        cout<<"Poor"; }
```

Switch statement (5)

- If none of the constant present in case labels is equal to the value of the variable or expression in switch statement, default label is executed.

Repetition Statements

- Control a block of code to be executed repeatedly for a fixed number of times or until a certain condition fails.
- There are three C++ repetition statements:
  - **for** Statement/loop
  - **while** statement/loop
  - **do…while** statement/ loop

For Statement / loop

- used to repeatedly execute a block of instructions until a specific condition fails.
- **Syntax:**

```
for(expression1 ; expression2 ; expression3)
    statements;
```

Where:

expression1 → **initialization**

expression2 → **condition** and

expression3 → either **increment** or **decrement**.

- **Or**

```
for(initialization ; condition ; increase/decrease)
    statement;
```

Steps of execution: for loop (1)

1. Initialization is executed. (will be executed only once)
2. Condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.
3. Statement is executed.
4. Finally, whatever is specified in the increase or decrease field is executed and the loop gets back to step two.

```
for(initialization ; condition ; increase/decrease)
```

Steps of execution: for loop (2)

- While making one or more of the three expressions null, the semi colons CAN NOT be ignored.
- **Example:**

for (;n<10;) //if we want neither initialization nor increase/decrease

for (;n<10;n++) //if no initialization is needed.

for (; ;) //is an infinite loop unless and otherwise there is if statement inside the loop.

for loop (3)

```
//Example1
int i,n;
for(i=1;i<=6;i++)
{
for(n=6;n>=i;n--)
cout<<"*";
cout<<"\n";
}
```

```
//Example2
int i,n,sum=0;
cout<<"Enter Num";
cin>>n;
for(i=1;i<=n;i++)
{
sum+=i;
}
cout<<“Sum=“<<sum;
```

for loop (4)

```
// Example: For loop to calculate factorial
#include <iostream.>
int main()
{
    int n, factorial = 1;
    cout << "Enter a positive integer: ";
    cin >> n;

    for (int i = 1; i <= n; ++i) {
        factorial *= i; //factorial = factorial*i;
    }
    cout<< "Factorial of "<<n<<" = "<<factorial;
    return 0;
}
```

While statement/loop

- Provides a way of repeating a statement or a block as long as a condition holds.
- Syntax:

while(expression)

statements;

```
//Example
#include<iostream.h>
int main(){
int n,i=1,sum=0;
cout<<"Enter Num";
cin>>n;
while(i<=n)
      {
      sum+=i;
      i++;
      }
cout<<"Sum="<<sum;
}
```

do…while statement (1)

- Similar to the while statement, except that its body is executed first and then the loop condition is examined.
- The body of the loop will be executed **at least once**. Then the condition will be tested.
- If condition is TRUE, the loop is repeated; otherwise, the loop ends.

do…while statement (2)

- Syntax:

```
do
{
 statement;
}
while(expression);
```

- First statement is executed and then expression is evaluated.

do…while Statement (3)

```
//Example
#include<iostream.h>
int main()
{
int n, number=1,sum=0;
cout<<"Enter Number";
cin>>n;
do
{
sum+=number;
number++;
}
while(number<=n);
cout<<"Sum="<<sum;}
```

Difference b/n while & do…while Loop

- The while loop checks the condition first and then executes the loop whereas do-while loop executes the loop first and then checks the condition so it is executed at least once even if the condition is false.

Nested Loops

- Loops may be nested, with one loop sitting in the body of another.
- The inner loop will be executed in full for every execution of the outer loop.
- Example:

```
#include<iostream.h>
int main(){
for(int i=1;i<8;i++)
{
for(int j=i;j>0; j--)
{
cout<<i;
}
cout<<"\n";
}}
```

Pitfalls of repetition statements

- **Infinite loop:**

```
for(int i=1;i<=10;i--)
```

- **Off-By-One Bugs (OBOB):**

for(int i=1;i<=10;i++): How many times the loop will be executed ???

Jump statements

- **C++ has four statements that cause an unconditional branch.**

  1. **goto**
  2. **break**
  3. **continue**
  4. **return**

- You may use return and goto anywhere in your program.
- You may use break and continue statements in conjunction with any of the loop statements.

goto statement

- In C++, a label is just a name followed by a colon (:).
- The label is placed to the left of a legal C++ statement, and a jump is accomplished by writing **goto** followed by the label name.

Example:

```
#include <iostream.h>
int main() {
int counter = 0;
loop: counter ++; // loop is label
cout << "counter=" << counter ;
if (counter < 5)
goto loop;// jump to label loop
cout << "Complete!"; }
```

continue statement

- The **continue** statement causes a loop to begin again at the top of the loop.
- Used when there is a need to skip some statement/s inside the loop.

```
while (test expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        continue;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        continue;
    }
    statements/
}
```

```
//Example:
#include <iostream.h>
int main()
{
for (int i = 1; i <= 10; i++)
        {
        if ( i == 6 || i == 9)
        {
            continue;
            }
        cout<<i<<"\n";
    }
  }
```

break statement

- The **break** statement causes the immediate end of a loop or switch.
- Execution jumps to the closing brace.

//Example:

```
#include <iostream.h>
int main()
{
 for (int i = 1; i <= 10;
 if ( i == 6)
{
    break;
   }
   cout<<i<<"\n";
 }
}
```

return

- used to return from a function.
- If return has a value associated with it, that value is the return value of the function.
- If no return value is specified it is the same as the } that ends a function.

```
int main(){
  // statements;
return 0;

}
```

exit statement (1)

- You can break out of a program by using the standard library function **exit().**
- This function causes the immediate termination of the entire program.
- The general form is exit (int return_code);
- The value of the return_code is generally
  - zero**(0)** to indicate a normal termination of a program
  - other value(usually **1**) for abnormal termination (to indicate some sort of error).

exit statement (2)

Example:

```
#include <iostream.h>
int main()
{
int b;
cout << "Enter the bit value 0,1:" << '\n';
cin >>b;
switch( b)
{
case 0:
cout << "Bit value is 0" << '\n';
break;
case 1:
cout << "Bit value is 1" << '\n';
break;
default:
exit(0);
}
}
```

- The control structures exit() function is used to terminate program execution of a "switch" statement if the input value is other than 0 or 1.

Thank You !!!

Chapter Five

Loop Statement

Repetition Statements

- Repetition statements control a block of code to be executed repeatedly for a fixed number of times or until a certain condition fails.
- There are three C++ repetition statements:
  - The **For** Statement or loop
  - The **While** statement or loop
  - The **do…while** statement or loop

The for statement / loop

- The "for" statement (also called loop) is used to repeatedly execute a block of instructions until a specific condition fails.
- The General Syntax is:

```
for(expression1; expression2; expression3)
       statements;
```

The for loop has three expressions:

- **expression1**: is one or more statements that will be executed only once and before the looping starts.
- **Expression2**: is the part that decides whether to proceed with executing the instructions in the loop or to stop. Expression2 will be evaluated each time before the loop continues. The output of expression2 should be either non zero (to proceed with the loop) or zero (to stop the loop) to represent true and false output respectively.
- **Expression3**: is one or more statements that will be executed after each iteration.

Thus, first **expression1** is evaluated and then each time the loop is executed, **expression2** is evaluated. If the outcome of **expression2** is non-zero then **statements** is executed and **expression3** is evaluated. Otherwise, the loop is terminated.

In most programs, the "for loop" will be used for such expressions where **expression1** is **initialization**, **expression2** is **condition** and **expression3** is either **increment** or **decrement**.

The general format can be expressed as follows for the sake of clarity:

```
for(initialization; condition ; increase/decrease)
      statement;
```

Steps of execution of the for loop:

1. Initialization is executed. (will be executed only once)
2. Condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.
3. Statement is executed.
4. Finally, whatever is specified in the increase or decrease field is executed and the loop gets back to step two.

Example: Guess the output of the following code:

```
int main()
{
       for(int i=10;i>0;i--)
       {
       cout<<n<<“,”;
       }
       cout<< “FIRE!”;
       return 0;
}
```

- Even though it is not recommended, expression1, expression2 and expression3 can be optional or can be ignored. This means that they can take NULL statement.
- But making **expression2** null means that the loop will not terminate. In such cases one can include an “if” statement inside the “for” loop which will test a condition and break out from the loop using the **break** statement.
- While making one or more of the three expressions null, the semi colons CAN NOT be ignored.

Example:

for (;n<10;) //if we want nether initialization nor increase/decrease

for (;n<10;n++) //if no initialization is needed.

for (; ;) //is an infinite loop unless and otherwise there is if statement inside the loop.

It is declared above that expression1 and expression3 can be one or more statements. The composite statements should be separated by a comma. This means, optionally, using the comma operator (,) we can specify more than one instruction in any of the two fields included in a "for" loop.

Example:

```
for(n=0,i=100;n!=i; n++,i--)
             {
              //whatever here
             }
```

In the above example, n=0 and i=100 will be part of expression1 and will be executed only once before the loop starts. In addition, n++ and i—will be part of expression3 and will be executed after each looping/iteration.

Example 1: //the following for statement adds the numbers between 0 and n

```
       int Sum=0;
       for(int i=0; i<=n;i++)
            Sum=Sum+i;
```

Example 2: //the following for statement adds the even numbers between 0 and n

```
       int Sum=0;
       for(int i=0; i<=n;)
          {
               Sum=Sum+i;
               i+=2;
          }
```

Example 3: //the following for statement adds the even numbers between 0 and n

//where all the three expressions are null.

```
       int Sum=0;
       int i=0;
       for( ; ; )
          {
              If(i<=n)
                      break;
                 else
                    {
                      Sum=Sum+i;
                      i++;
```

```
                    }
            }
```

In the above example, the initialization is at the top before the looping starts, the condition is put in if statement before the instructions are executed and the increment is found immediately after the statements to be executed in the loop.

NB: even though there is the option of making all the three expressions null in a "for" loop, it is not recommended to make the condition to take null statement.

```
 // Example: For loop to calculate factorial
#include <iostream.>
int main()
{
    int n, factorial = 1;
    cout << "Enter a positive integer: ";
    cin >> n;

    for (int i = 1; i <= n; ++i) {
        factorial *= i;   // factorial = factorial * i;
    }
    cout<< "Factorial of "<<n<<" = "<<factorial;
    return 0;
}
```

The while Statements

- The while statement (also called while loop) provides a way of repeating a statement or a block as long as a condition holds / is true.
- The general form of the while loop is:

```
while(expression)
       statements;
```

- First expression (called the loop condition) is evaluated. If the outcome is non-zero then statement (called the loop body) is executed and the whole process is repeated. Otherwise, the loop is terminated.

- Suppose that we wish to calculate the sum of all numbers from 1 to some integer value n. this can be expressed as:

**Example 1:// ** adds the numbers between 0 and any given number n

```
int i=1;
sum = 0;
while(i <= n)
       sum += i++;
```

Example 2: //adds the numbers between 0 and 100

```
       number=1;
       sum=0;
                     while(number <= 100)
                      {
                  sum += number;
                  number++;
                   }
```

Example 3: /*the following while loop will request the user to enter his/her age which should be between 0 and 130. If the user enters a value which is not in the range, the while loop test the value and request the age again until the user enters a valid age. */

```
cout<<"\n enter your age [between 0 and 130]:";
cin>>age;
while(age < 0 || age > 130)
   {
              cout<<"\n invalid age. Plz try again!";
              cin>>age;
   }
```

Do…while loop

- The do statement (also called the do while loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined.
- In do…while loop, we are sure that the body of the loop will be executed at least once. Then the condition will be tested.
- The general form is:

```
do
  {
```

```
        statement;
   }
while(expression);
```

First statement is executed and then expression is evaluated. If the outcome of the expression is nonzero, then the whole process is repeated. Otherwise the loop is terminated.

Example: //our previous example (Eg3) in the while loop might be changed as:

```
age=-1;
do
  {
     cout<<"\n enter your valid age [between 0 and 130]:";
     cin>>age;
  }
while(age < 0 || age > 130);
```

Example: //what do you think is the outcome of the following code:

```
       unsigned long n;
       do
               {
              cout<<"\n enter number (0 to end):";
              cin>>n;
              cout<<"\n you entered:"<<n;
               }
       while(n != 0);
```

Pitfalls in witting repetition statements

There are some pitfalls worth mentioning in repetition statements. These pit falls are the most common programming errors committed by programmers

Infinite loop: no matter what you do with the while loop (and other repetition statements), make sure that the loop will eventually terminates.

Example 1: //Do you know why the following is an infinite loop?

```
      int product = 0;
     while(product < 50)
       product *= 5;
```

Example 2: //Do you know why the following is an infinite loop?

```
int counter = 1;
while(counter != 10)
```

```
        counter += 2;
```

- In the first example, since **product** is initialized with zero, the expression "product*=5" will always give us zero which will always be less than 50.
- In the second example, the variable counter is initialized to 1 and increment is 2, counter will never be equal to 10 as the counter only assumes odd values. In theory, this while loop is an infinite loop, but in practice, this loop eventually terminates because of an overflow error as counter is an integer it will have a maximum limit.

<u>Off-By-One Bugs (OBOB)</u>: another thing for which you have to watch out in writing a loop is the so called Off-By-One Bugs or errors. Suppose we want to execute the loop body 10 times. Does the following code work?

Example 1:

```
 count = 1;
  while(count < 10)
    {
       …
       count++;
   }
```

No, the loop body is executed nine times. How about the following?

Example 2:

```
count = 0;
while(count <= 10)
{
       …
       count++;
}
```

No this time the loop body is executed eleven times. The correct is

Example 3:

```
count = 0;
while(count < 10)
{
       …
       count++;
}
```

OR

```
ount = 1;
while(count <= 10)
{
       …
       count++;}
```

Nested loop Statements

A loop inside another loop is called a nested loop. The number of loops depend on the complexity of a problem. Suppose, a loop, outer loop, running n number of times consists of another loop inside it, inner loop, running m number of times. Then, for each execution of the outer loop from 1...n, the inner loop runs maximum of m times.

Types of nested loops

There can be many types of nested loops in C++ but the mostly used nested loops are

- Nested while loop
- Nested do-while loop
- Nested for loop

Note: There can also be very variation of nested loops where a while loop can be inside a for loop, a for loop can be inside a do-while loop and many more.

Nested while loop

A while loop inside another while loop is called nested while loop.

Syntax of Nested while loop

```
while (condition1)
{
    statement(s);
    while (condition2)
    {
        statement(s);
        ... ... ...
    }
    ... ... ...
}
```

Flowchart for Nested while loop

fig: Flowchart for nested while loop

Example of Nested while loop

C++ program to print the number pattern.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{
    int i=1,j;
    while (i <= 5)
    {
        j=1;
        while (j <= i)
```

```
        {
            cout <<j;
            j++;
        }
        cout << endl;
        i++;
    }
    return 0;
}
```

Nested do-while loop

A do-while loop inside another do-while loop is called nested do-while loop.

Syntax of Nested do-while loop

```
do
{
    statement(s);
    do
    {
        statement(s);
        ... ... ...
    }while (condition2);
    ... ... ...
}while (condition1);
```

Flowchart of Nested do-while loop

Fig: Flowchart for nested do-while loop

Example of do-while loop

C++ program to print the given star pattern.

```
*
**
* **
****
*****
```

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int i=1,j;
    do
    {
        j=1;
        do
        {
            cout << "*";
            j++;
        }while(j <= i);
        i++;
        cout << endl;
    }while(i <= 5);
    getch();
    return 0;
}
```

Nested for loop

A for loop inside another for loop is called nested for loop.

Syntax of Nested for loop

```
for (initialization; condition; increment/decrement)
{
    statement(s);
    for (initialization; condition; increment/decrement)
    {
        statement(s);
        ... ... ...
    }
    ... ... ...
}
```

Flowchart of Nested for loop

Fig: Flowchart for nested for loop

Example of Nested for loop

C++ program to find the sum of 2 matrices.

```
#include <iostream.h>
#include <conio.h>
int main()
{
    int a[10][10], b[10][10], s[10][10];
    int i,j,row, column;
    cout <<"Enter size of row:";
    cin >> row;
    cout <<"Enter size of column:";
    cin >> column;
    cout <<"Enter elements of matrix A" << endl;
    for(i=0;i<row;i++)
    {
        for(j=0;j<column;j++)
        {
            cin >> a[i][j];
        }
    }
    cout <<"Enter elements of matrix B" << endl;
```

```
    for(i=0;i<row;i++)
    {
        for(j=0;j<column;j++)
        {
            cin >> b[i][j];
        }
    }
    cout << "Sum of A and B" << endl;
    for(i=0;i<row;i++)
    {
        for(j=0;j<column;j++)
        {
            cout << a[i][j] + b[i][j] << "   ";
        }
        cout << endl;
    }
    return 0;
}
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

No	Control Statement & Description
1	**break statement** Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**goto statement** Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The continue statements

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It is an error to use the continue statement outside a loop.
- In while and do while loops, the next iteration commences from the loop condition.
- In a "for" loop, the next iteration commences from the loop's third expression.

Example:

```
       for (int n=10; n>0; n--)
              {if(n==5)
                     continue;     //causes a jump to n–
                    cout<<n<< ",";
              }
```

When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For example, in the following set of nested loops, the continue statement applies to the "for" loop, and not to the "while" loop.

Example:

```
 while(more)
      {
      for (i=0; i<n; i++)
      {cin>>num;
             if(num<0)
                  continue; //causes a jump to: i++
      }
       }
```

The break statements

- A break statement may appear inside a loop (**while**, **do**, or **for**) or a switch statement. It causes a jump out of these constructs, and hence terminates them.
- Like the continue statement, a break statement only applies to the "loop" or "switch" immediately enclosing it. It is an error to use the break statement outside a loop or a switch statement.

Example:

```
  for(n=10;n>0;n--)
```

```
        {        cout<<n<< “,”;
                if (n = = 3)
                  {cout<< “count down aborted!!”;
                       break;
                   }}
```

GOTO Statements

In C++ programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

Syntax of goto Statement

```
goto label;
... .. ...
... .. ...
... .. ...
label:
statement;
... .. ...
```

In the syntax above, label is an identifier. When goto label; is encountered, the control of program jumps to label: and executes the code below it.

Example: goto Statement

```
// This program calculates the average of numbers entered by user.
// If user enters negative number, it ignores the number and
// calculates the average of number entered before it.
```

```
# include <iostream.h>
int main()
{
    float num, average, sum = 0.0;
    int i, n;

    cout << "Maximum number of inputs: ";
    cin >> n;
    for(i = 1; i <= n; ++i)
    {
        cout << "Enter n" << i << ": ";
        cin >> num;
        if(num < 0.0)
        {
           // Control of the program move to jump:
            goto jump;
        }
        sum += num;
    }
jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
}
```

Worksheet

1. Write for, do-while, and while statements to compute the following sums and products.

    a. 1+2+3+…+100

    b. 5+10+15+…+50

    c. 1+1/2+1/3+1/4+…1/15

    d. 1*2*3*…*20

2. write an application to print out the numbers 10 through 49 in the following manner

10 11 12 13 14 15 16 17 18 19

20 21 22 23 24 25 26 27 28 29

30 31 32 33 34 35 36 37 38 39

40 1 42 43 44 45 46 47 48 49

3. A prime number is an integer greater than one and divisible only by itself and one. The first seven prime numbers are 2, 3, 5, 7, 11, 13, and 17. Write a program that displays all the prime numbers between 1 and 100.

4. Write a C++ program that counts the number of digits in an integer number. For example; 23,498 has five digits.

5. Write a C++ application that can compute the letter grade of a student after accepting the student's mid and final mark. The program should only accept mid result [0-40] and final [0 - 60]. If the data entered violates this rule, the program should display that the user should enter the mark in the specified range. The program is also expected to run until the user refuses to continue.

6. Write a C++ program that accepts a positive number from the user and displays the factorial of that number. Use for loops to find the factorial of the number.

7. Write a C++ code that computes the sum of the following series.

Sum = 1! + 2! + 3! + 4! + …n!

The program should accept the number from the user.

8. Using the ASCII table numbers, write a program to print the following output, using a nested for loop. (Hint: the outer loop should loop from 1 to 5, and the inner loop's start variable should be 65, the value of ASCII "A").

A

A B

A B C

A B C D

A B C D E

9. Write a C++ program that displays the following output using their ASCII values.

a

bc

def

gehi

jklmn

opqrst

10. Write a C++ program that will print the following shapes.

```
A.          B.          C.          D.
*           * * * * *       *           *
* *         * * * *       ***         ***
* * *       * * *       *****       *****
* * * *     * *       *******         ***
* * * * *   *       *********           *
```

12. Write a weather-calculator program that asks for a list of the previous 10 days' temperatures, computes the average, and prints the results. You have to compute the total as the input occurs, then divide that total by 10 to find the average. Use a while loop for the 10 repetitions.

13. Write a C++ program that accepts marks of five students and then displays their average. The program should not accept mark which is less than 0 and mark greater than 100.

14. Develop a calculator program that computes and displays the result of a single requested operation.

E.g. if the input is

15 * 20, then the program should display 15 * 20 equals 300

If the operator is not legal, as in the following example

24 ~ 25 then the program displays ~ is unrecognized operator

As a final example, if the denominator for a division is 0, as in the following input: 23 / 0 then the program should display the following:

23 0 can't be computed: denominator is 0.

15. Use either a switch or an if-else statement and display whether a vowel or a consonant character is entered by the user. The program should work for both lower case and upper case letters.

16. Write a C++ code to display only even numbers found between 0 and 20.

17. Write a C++ application that extracts a day, month and year and determine whether the date is valid. If the program is given a valid date, an appropriate message is displayed. If instead the program is given an invalid date, an explanatory message is given. Note: to recognize whether the date is valid, we must be able to determine whether the year is a leap year or not.

An example of the expected input/output behavior for a valid date follows

```
Please enter a date (dd mm yyyy) : 30 4 2006
30/4/2006 is a valid date
Please enter a date (dd mm yyyy) : 1 13 2006
Invalid month: 13
Please enter a date (dd mm yyyy) : 29 2 1899
Invalid day of month 29
```

If the year is a leap year, then February will have total of 29 days. Otherwise, it will have 28 days. If the year is not a century year and is evenly divisible by 4, then the year is a leap year. If the year is a century year (years whose last digits are 00) and is evenly divisible by 400, then the year is a leap year.

Chapter Six

6. Arrays and String Manipulations

6.1 Define an array

Arrays

In previous chapters, you declared a single int, char, or another object. You often want to declare a collection of objects, such as 20 int.

What Is an Array?

An array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array.

One dimensional array

You declare an array by writing the type, followed by the array name and the subscript. The subscript is the number of elements in the array, surrounded by square brackets. For example,

```
int age[25];
```

Declares an array of 25 integers, named age. When the compiler sees this declaration, it sets aside enough memory to hold all 25 elements. Because each integer requires 2 bytes, this declaration sets aside 50 contiguous bytes of memory.

Array Elements: You access each of the array elements by referring to an offset from the array name. Array elements are counted from zero. Therefore, the first array element is arrayName[0]. In the age example, age[0] is the first array element, age[1] the second, and so forth.

More generally, SomeArray[n] has n elements that are numbered SomeArray[0] through SomeArray[n-1].

Using an integer array.

```
//Arrays
#include <iostream.h>
int main()
{
   int myArray[5];
   int i;
   for ( i=0; i<5; i++)  // 0-4
   {
      cout << "Value for myArray[" << i << "]: ";
```

```
        cin >> myArray[i];
    }
    for (i = 0; i<5; i++)
        cout << i << ": " << myArray[i] << "\n";
  return 0;
}
```

Line 6 declares an array called myArray, which holds five integer variables. Line 8 establishes a loop that counts from 0 through 4, which is the proper set of offsets for a five-element array. The user is prompted for a value, and that value is saved at the correct offset into the array.

The first value is saved at myArray[0], the second at myArray[1], and so forth. The second for loop prints each value to the screen.

```
NOTE: Arrays count from 0, not from 1. This is the cause of many bugs in programs written by C++ novices. Whenever you use an array, remember that an array with 10 elements counts from ArrayName[0] to ArrayName[9]. There is no ArrayName[10].
```

6.2. Declare an array and initialize it

Initializing Arrays

You can initialize a simple array of built-in types, such as integers and characters, when you first declare the array. After the array name, you put an equal sign (=) and a list of comma-separated values enclosed in braces. For example,

```
int IntegerArray[5] = {10, 20, 30, 40, 50 };
```

Declares IntegerArray to be an array of five integers. It assigns IntegerArray[0] the value 10, IntegerArray[1] the value 20, and so forth. If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write

```
int IntegerArray[] = {10, 20, 30, 40, 50 };
```

You will create exactly the same array as you did in the previous example.

You cannot initialize more elements than you've declared for the array. Therefore,

```
int IntegerArray[5] = {10, 20, 30, 40, 50, 60};
```

Generates a compiler error because you've declared a five-member array and initialized six values. It is legal, however, to write

```
int IntegerArray[5] = {10, 20};
```

If you don't initialize an array member, its value will be set to 0.

Declaring Arrays

Arrays can have any legal variable name, but they cannot have the same name as another variable or array within their scope. Therefore, you cannot have an array named myCats[5] and a variable named myCats at the same time.

Accessing Arrays

To access members of the array, use the subscript operator.

Example

```
        int theNinethInteger = MyIntegerArray[8];
```

6.3. Manipulate multidimensional arrays

It is possible to have arrays of more than one dimension. Each dimension is represented as a subscript in the array. Therefore, a two-dimensional array has two subscripts; a three-dimensional array has three subscripts; and so on. Arrays can have any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

A good example of a two-dimensional array is a chessboard. One dimension represents the eight rows; the other dimension represents the eight columns.

```
                SQUARE Board[8][8];
```

Initializing Multidimensional Arrays

You can initialize multidimensional arrays. You assign the list of values to array elements in order, with the last array subscript changing while each of the former holds steady. Therefore, if you have an array

```
        int theArray[5][3]
```

You initialize this array by writing

```
        int theArray[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 }
```

For the sake of clarity, you could group the initializations with braces. For example,

```
int theArray[5][3] = {  {1,2,3},
{4,5,6},{7,8,9},{10,11,12},{13,14,15} };
```

The compiler ignores the inner braces, which make it easier to understand how the numbers are distributed. Each value must be separated by a comma, without regard to the braces. The entire initialization set must be within braces, and it must end with a semicolon.

6.4. Strings: Definition, accessing Strings

Strings

A string is an array of characters. The only strings you've seen until now have been unnamed string constants used in cout statements, such as

```
cout << "hello world.\n";
```

In C++ a string is an array of chars ending with a null character. You can declare and initialize a string just as you would any other array. For example,

```
char Greeting[] = { `H', `e', `l', `l', `o', ` `, `W','o','r','l','d', `\0' };
```

The last character, `\0', is the null character, which many C++ functions recognize as the terminator for a string. Although this character-by-character approach works, it is difficult to type and admits too many opportunities for error. C++ enables you to use a shorthand form of the previous line of code. It is

```
char Greeting[] = "Hello World";
```

You should note two things about this syntax:

- Instead of single quoted characters separated by commas and surrounded by braces, you have a double-quoted string, no commas, and no braces.
- You don't need to add the null character because the compiler adds it for you.

The string Hello World is 12 bytes. Hello is 5 bytes, the space 1, World 5, and the null character 1. You can also create uninitialized character arrays.

String Manipulation

C++ inherits from C a library of functions for dealing with strings. Among the many functions provided are: strcpy(), strcat(), strlen(), strcmp().

strcpy() copies the entire contents of one string into a designated buffer.

Using strcpy().

```
#include <iostream.h>
#include <string.h>
  int main()
  {
     char String1[] = "I am Ethiopian";
     char String2[80];
     strcpy(String2,String1);
    cout << "String1: " << String1 << endl;
    cout << "String2: " << String2 << endl;
```

```
    return 0;
 }
```

Output:

String1: I am Ethiopian

String2: I am Ethiopian

The header file string.h is included in line 2. This file contains the prototype of the strcpy() function. strcpy() takes two character arrays--a destination followed by a source. If the source were larger than the destination, strcpy() would overwrite past the end of the buffer.

strcmp(s1,s2) – this function compares two strings s1,s2 and return values based on the comparison. It returns:

<0 if s1<s2

=0 if s1=s2

>0 if s1>s2

You use it to sort strings of characters.

strlen(s1)– this returns the length of a string s1

strcat(s1,s2)–this concatenates string s2 onto the end of s1. It combines the two strings together into one unit.

strcpy(s1,s2)–it copies the content of string s2 onto string s1.

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
       char firstname[20];
       char lastname[20];
       char fullname[60];
       cout<<"Enter your first name: ";
       cin>>firstname;
       cout<<"Enter your last name: ";
       cin>>lastname;
       strcpy(fullname, firstname); //copies firstname and store on full name
       strcat(fullname, lastname); //copies lastname and add to the end of fullname
       cout<<" Your full name is "<<fullname;
       return 0;
```

```
}
```

Example: a program that sorts names alphabetically

```
#include<iostream.h>
#include<conio.h>
void main()
{
              char name[20][40];
              char temp[40];
              for(int i=0;i<20;i++)
 {
                 cin>>name[i];
             }
             for(i=0;i<20;i++)
               for(int j=0;j<20;j++)
               {
                        if (strcmp(name[i],name[j])>0)
               {
                             strcpy(temp,name[i]);
                         strcpy(name[i],name[j]);
                         strcpy(name[j],temp);
                        }
               }
}
```

Exercise

1) Declare an array that accepts names of ten students.

2) Sort the names of students accepted in question 1 alphabetically. (Hint: use strcmp and swapping)

Chapter Seven

Functions

What is function?

A function is defined as a group of instruction used to achieve a specific task. These functions can be called from other function. Hence if any groups of statements are to be frequently used then the statements are grouped together and named which is accessed by that new function when needed.

Functions decrease the complexity of a program. A program having multiple functions can be easily debugged than a program that is large and runs without a single function. So, a function reduces the size of a program and increases the program modularity.

C++ adopts functions as its fundamental programming approach. Basics of C++ deals more with functions and even classes also use the functions as their member functions.

The need for function

Functions are more important for the following reasons:

1. When solving large problems, it is usually necessary to split the problem down into a series of sub-problems, which in turn may be split into further sub-problems etc. This is usually called a **top-down** approach. This process continues until problems become of such a size that they can be solved by a single programmer.
2. In a particular organization or industry, it may be found that in carrying out the top-down approach in some tasks occur very frequently. For example, the operation of sorting a file of data into some order occurs frequently in data-processing applications. Thus, a **library** of such commonly used functions can be built up and re-used in many different programs. This obviously saves much work and cuts down errors if such functions have already been well tested.
3. There are many very specialized problem areas, not every programmer can know every area. For example, many programmers working in scientific applications will frequently use

mathematical function routines like sine and cosine, but would have no idea how to write such routines. Similarly, a programmer working in commercial applications might know very little about how an efficient sorting routine can be implemented. However, a specialist can write such routines, place them in a public library of functions and all programmers can benefit from this expertise by being able to use these efficient and well tested functions.

Functions are of two types:

- Library function
- User defined functions

Library/Built-in function

Library functions are pre-written functions or built in functions. Many activities in C++ use library functions. These functions include graphical functions, text formatting functions, memory management and data conversion. Some of the mathematical functions available in the C++ mathematics library are listed below.

acos(x)	inverse cosine, -1 <= x <= +1, returns value in radians in range 0 to PI
asin(x)	inverse sine, -1 <= x <= +1, returns value in radians in range 0 to PI
atan(x)	inverse tangent, returns value in radians in range -PI/2 to PI/2
cos(x)	returns cosine of x, x in radians
sin(x)	returns sine of x, x in radians
tan(x)	returns tangent of x, x in radians
exp(x)	exponential function, e to power x
log(x)	natural log of x (base e), x > 0
sqrt(x)	square root of x, x >= 0
fabs(x)	absolute value of x
floor(x)	largest integer not greater than x
ceil(x)	smallest integer not less than x

Consider the following example which uses ceil and floor library functions from math.h. The floor function rounds down the float value to the nearest lower integer whereas ceil rounds up the float value to the nearest upper integer.

Example:

```
#include<math.h>
#include<iostream.h>
int main(){
float number=255.98;
float down,up;
down=floor(number);
up=ceil(number);
cout<<"The number is "<<number<<endl;
cout<<"The floor of the number is"<<down<<endl;
cout<<"The ceil part of the number is"<<up<<endl;
}
```

Output

```
The number is 255.98
The floor of the number is255
The ceil part of the number is256
```

User defined functions

Besides the library functions, we can also define our own functions which are called as user defined functions. C++ allows declaring user-defined functions, which reduces program size and increase modularity.

The following program shows how to use user-defined functions.

Example:

```
#include<iostream.h>
void func();//Function declaratio
void main()
{
func();//Function calling
cout<<"Statement 1 in main function"<<endl;
func();//Function calling
```

Output

```
Function called from main
Statement 1 in main function
Function called from main
Statement 2 in main function
Function called from main
```

```
cout<<"Statement 2 in main function"<<endl;
func();//function can be called from main any number of times
}
void func()
{
cout<"Function called from main"<<endl;//function definition
}
```

Just like a variable has to be declared before using, a function also has to be declared.

The statement just above the main is the **function declaration**. A **function declaration** tells the compiler that at some later point we are going to use a function by that name. It is called as prototype declaration. It should be terminated because it acts like a single statement.

Syntax for function declaration:

```
return_type  function_name();
```

The statement inside **main ()** tells the compiler to execute the functions. At this point the statements that are included in the function are executed. This is called **function calling.** It acts as a single statement; hence it should be terminated.

Syntax for function calling:

```
function_name();
```

The statement outside the **main ()** that is along with is called **function definition**. It specifies the task that the function is going to perform. The statements inside the braces are the body of the function. It includes variable declaration, loops and decisions.

A function definition should not end with semicolon. The starting brace and closing brace specify the function and the statements within it. At every point of function, i.e. at function declaration, definition and calling you find () (parenthesis). () or parenthesis specifies that the string preceding it is a function.

The statement void specifies that the function need not return any value. It will be explained in detail when dealing with call by value.

Consider the following statement

```
int i=10;
```

This statement declares a variable and defines it at the time of declaration. Just similarly a function can also be defined at the time of declaration. In this case the definition of a function cannot be after **main ()** because when you call the function in the **main ()** then the compiler raises an error.

So, you need to declare the function before **main().** Whenever it is necessary to define a function along with declaration before the **main().**

Example

```
#include<iostream.h>
void add(){
int i,j,k;
cout<<"Enter i"<<endl;
cin>>i;
cout<<"Enter j"<<endl;
cin>>j;
k=i+j;
cout <<"The result is "<<k<<endl;
}
void main(){
add();//calling the add function
cout<<"The main function calling the add function"<<endl;
add();//calling the add function
}
```

Output

```
Enter i
2
Enter j
6
The result is 8
The main function calling the add function
Enter i
6
Enter j
15
The result is 21
```

Function call by value

A function can also have arguments. An argument is a piece of data passed from a program to a function. The arguments are the variables that are passed in the parenthesis of a function. These arguments are used to process the data sent by the program.

Syntax: `int myfunc(int,int);`

This specifies the function called **myfunc** that takes two integer variables as arguments. A compiler identifies a function with the number of arguments and type of arguments other-wise which raises a compiler error. When we are declaring functions, the variables in function declaration are known as **parameters**. During function execution they are known as **function arguments.**

Example:

```
#include<iostream.h>
int disp(int,int) ;//declares prototype of the function disp
void main(){
int x,y;//declares two integer variables
cout<<"Enter number 1"<<endl;
cin>>x;
cout<<"Enter number 2"<<endl;
cin>>y;
disp(x,y);//function calling by passing
value for a and b
}
int disp(int a,int b){
cout<<"x+y="<<a+b<<endl;
cout<<"x-y="<<a-b<<endl;
cout<<"x*y="<<a*b<<endl;
cout<<"x/y="<<a/b<<endl;
}
```

Output

```
Enter number 1
18
Enter number 2
6
x+y=24
x-y=12
x*y=108
x/y=3
```

The program accepts two integer variables and passes them as **arguments** to the function **disp.** The disp function performs all the arithmetic operations on them and displays the result.

Sometimes it becomes necessary that a program passing arguments expecting a return from the function. This is known as **arguments with return value.** Actually, when you declare a function as **void** it implies that the function need not return anything.

This means that **if you don't specify void before the function name, the calling function is expecting a return from the called function.**

- Default return type is integer.

The return may be an int variable or a char variable or any other variable of any type. In this case a function that returns a value should be specified before the function at the time of declaration and definition.

Example: Consider the following program

The program defines a function called power, which takes two integer variables and prints the power of the first number raised to second number. Finally, after calculating power the value is returned to the calling function. You will observe that in place of void before the function name it is given as int so that it will return a value of int.

```
#include<iostream.h>
int power(int,int);//function declaration with return value
void main()
{
int a,b,c;
cout<<"Enter the first number"<<endl;
cin>>a;
cout<<"Enter the second number"<<endl;
cin>>b;
c=power(a,b);//assigning the returned value of the  function power
cout<<"The result of  "<<a<<"  raised  "<<b<<"  is: "<<c;
}//definition of function power
int power(int i,int j)
{
int p=1;
while(j>0)
{
```

Output

```
Enter the first number
3
Enter the second number
2
The result of 3 raised 2 is: 9
```

```
p=p*i;
j--;
}
return p;//p is returned to initialize to c
}
```

From the above program it is understood that, to make a function to return a value, the type has to be specified before the function name.

Call by reference

Call by value can return the result of a calculation. But it cannot affect the original variables. That is, in call by value the variables that are passed with as arguments from the caller function gets copied into the variables that are defined with the function definition. So, the copies of the original variables are created at another memory location. Thus, the changes to these variables are not affected to original variables.

Sometimes it is necessary to change the values of original variables that have been passed in the caller function. We can achieve this using **call by reference.**

In **call by reference**, the address of the variables passes as arguments to the function. Thus, the variables which are defined along with the function definition can actually access the values of the variables that are being passed to the function to the caller function. And hence the alteration for the original variables is possible.

Example:

The program follows call by reference method to swap two variables.

```
#include<iostream.h>
int swap(int &,int &);//& indicates you are passing the addresses
of the values not the values
void main(){
int a=10;
int b=90;
```

```
swap(a,b);//values of a and b are passed to the function
cout<<"The values of a is:"<<a<<endl;
cout<<"The value of b is: <<b<<endl;
}
int swap(int &p,int &q)
{
int temp;
temp=p;
p=q;
q=temp;
return 0;
}
```

```
The value of a is: 90
The value of b is: 10
```

In the example shown above you passed the address of the variables as argument. You will find that the declaration of swap included an & specified in place of arguments. It means that the function is expecting addresses of the variables as arguments.

& is called "address of" operator. It is used to pass the address of a variable to the called function or variable. So, the variable declared in definition of function also gets created at the same address. It implies that the variables can access the value of original variables. Thus, they can be altered. So, call by reference works fine when it is needed to access the original variables not created a copy of it.