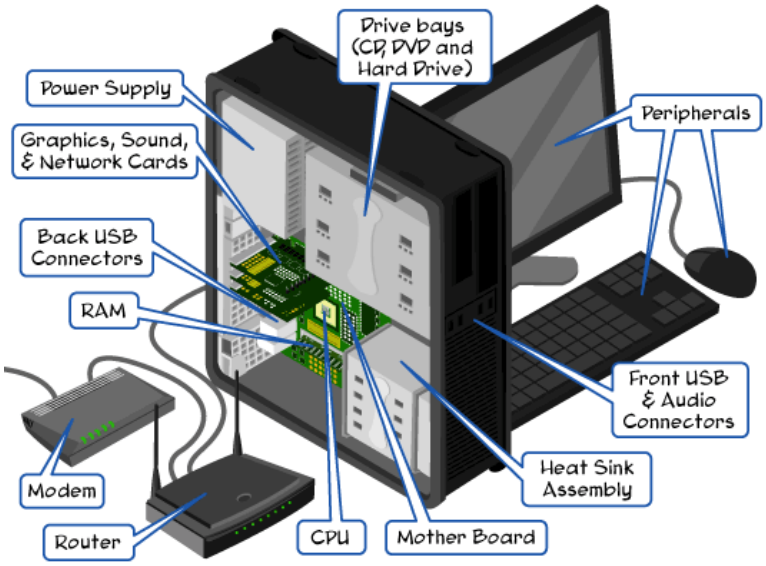
Lecture 1

computer:

is a device that accepts information (in the form of digitalized data) and manipulates it for some result based on a program or sequence of instructions on how the data is to be processed. Complex computers also include the means for storing data (including the program, which is also a form of data) for some necessary duration. A program may be invariable and built into the computer (and called logic circuitry as it is on microprocessors) or different programs may be provided to the computer (loaded into its storage and then started by an administrator or user). Today's computers have both kinds of programming.

The process of developing and implementing arious sets of instructions to enable a computer to do a certain task. These instructions are considered computer programs and help the computer to operate smoothly.



What is software and hardware?

Computer software, or simply software, is that part of a computer system that consists of encoded information or computer instructions, in contrast to the physical hardware from which the system is built.

Computer software includes computer programs, libraries and related non-executable data, such as online documentation or digital media. Computer hardware and software require each other and neither can be realistically used on its own.

software, is a collection of computer programs and related data that provide the instructions telling a computer what to do and how to do it.

Computer hardware (or simply hardware in computing contexts) is the collection of physical elements that constitutes a computer system. Computer hardware is the physical parts or components of a computer, such as the monitor, keyboard, computer data storage, hard disk drive(HDD), graphic cards, sound cards, memory (RAM), motherboard, and so on, all of which are tangible physical objects.[1] By contrast, software is instructions that can be stored and run by hardware..

Procedural programming: is derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, or functions (not to be confused with mathematical functions, but similar to those used in functional programming), simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

The history of C++:

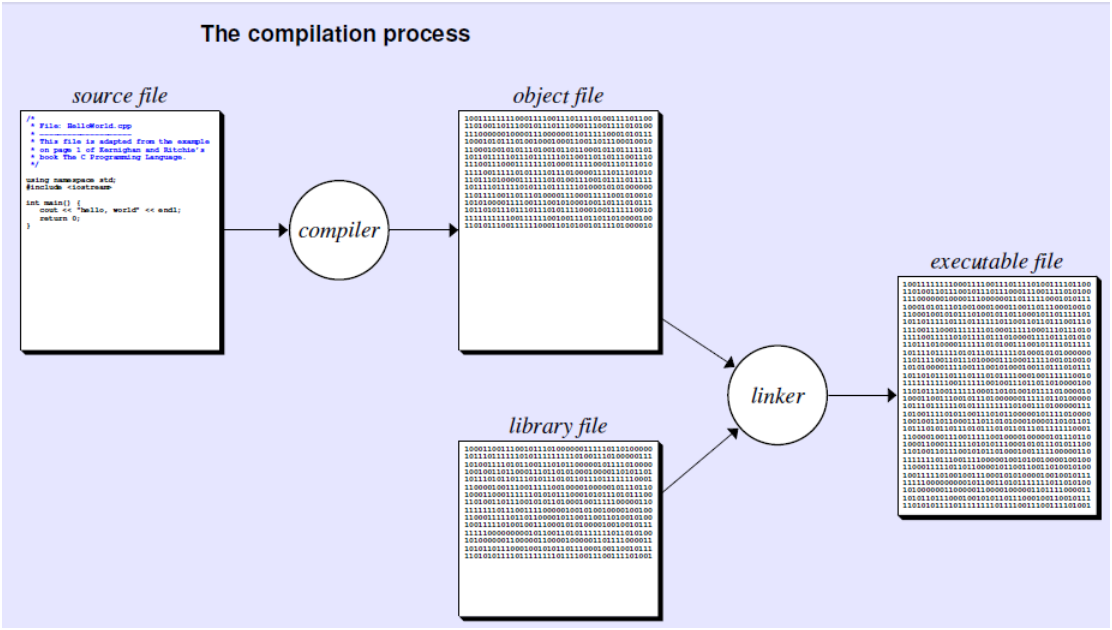
In the early days of computing, programs were written in machine language, which consists of the primitive instructions that can be executed directly by the machine. Programs written in machine language are difficult to understand, mostly because the structure of machine language reflects the design of the hardware rather than the needs of programmers. Worse still, each type of computing hardware has its own machine language, which means that a program written for one machine will not run on other types of hardware. In the mid-1950s, a group of programmers under the direction of John Backus at IBM had an idea that profoundly changed the nature of computing. Would it be possible, Backus and his colleagues wondered, to write programs that resembled

the mathematical formulas they were trying to compute and have the computer itself translate those formulas into machine language? In 1955, this team produced the initial version of FORTRAN (whose name is a contraction of formula translation),
which was the first example of a higher-level programming language. Since that time, many new programming languages have been invented, most of which build on previous languages in an evolutionary way. C++ represents the joining of two branches in that evolution. One of its ancestors is a language called C, which was designed at Bell Laboratories by Dennis Ritchie in 1972 and then later revised and standardized by the American National Standards Institute (ANSI) in 1989. But C++ also descends from a family of languages designed to support a different style of programming that has dramatically changed the nature of software development in recent years.

The most important thing while learning C++ is to focus on concepts. The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

The compilation process:

When you write a program in C++, your first step is to create a file that contains the text of the program, which is called a source file. Before you can run your program, you need to translate the source file into an executable form. The first step in that process is to invoke a program called a compiler, which translates the source file into an object file containing the corresponding machine-language instructions. This object file is then combined with other object files to produce an executable file that can be run on the system. The other object files typically include predefined object files called libraries that contain the machine-language instructions for various operations commonly required by programs. The process of 6 Overview of C++ combining all the individual object files into an executable file is called linking. The steps in the compilation process are illustrated in the Figure bellow.


The compilation process
source file
/*
* File: HelloWorld.cpp
*
* This file is adapted from the example
* on page 1 of Kernighan and Ritchie's
* book The C Programming Language.
*/
using namespace std;
#include <iostream>
int main() {
cout << "Hello, world" << endl;
return 0;
}
compiler
object file
library file
linker
executable file

Lecture 2

1. Algorithm:

As stated earlier an algorithm can be defined as a finite sequence of effect statements to solve a problem. An effective statement is a clear, unambiguous instruction that can be carried out .Thus an algorithm should special the action to be executed and the order in which these actions are to be executed.

Studying algorithms helps you develop widely applicable analytical skills—you learn not just how to solve individual problems but to develop recipies for getting answers to many problems.

Algorithm properties:

 Finiteness: the algorithm must terminate a finite number of steps.

 Non-ambiguity: each step must be precisely defined. At the completion of each step, the nest step should be uniquely determined.

 Effectiveness: the algorithm should solve the problem in a reasonable amount of time.

Write an algorithm to add two numbers entered by user.

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

Write an algorithm to find the largest among three different numbers entered by user.

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a>b

    If a>c

      Display a is the largest number.

    Else

      Display c is the largest number.

  Else

    If b>c

      Display b is the largest number.

    Else

      Display c is the greatest number.

Step 5: Stop

Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D←b2-4ac

Step 4: If D≥0

r1←(-b+√D)/2a

r2←(-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp←b/2a

ip←√(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

Write an algorithm to find the factorial of a number entered by user.

Step 1: Start

Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables

factorial←1

i←1

Step 4: Read value of n

Step 5: Repeat the steps until i=n

5.1: factorial←factorial*i

5.2: i←i+1

Step 6: Display factorial

Step 7: Stop

Lecture 3

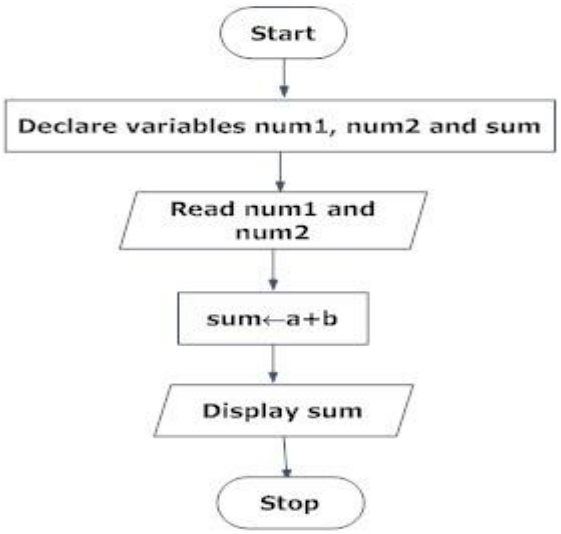
Flowchart:

A flowchart is a type of diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields .The main symbols used to draw a flowchart are shown in following figure.

Symbol	Shape	Terminology	Purpose of the Symbol
	Oval	Terminator	To begin/end (start and stop)
	Arrow	Connector line	To connect any two symbols
	Rectangle	Process	Initialization and Computation
	Parallelogram	Data	Read and Print Data
	Rhombus	Decision	Conditionals – branching
c	Circle	On page connector	To have a flowchart connectivity on the same page

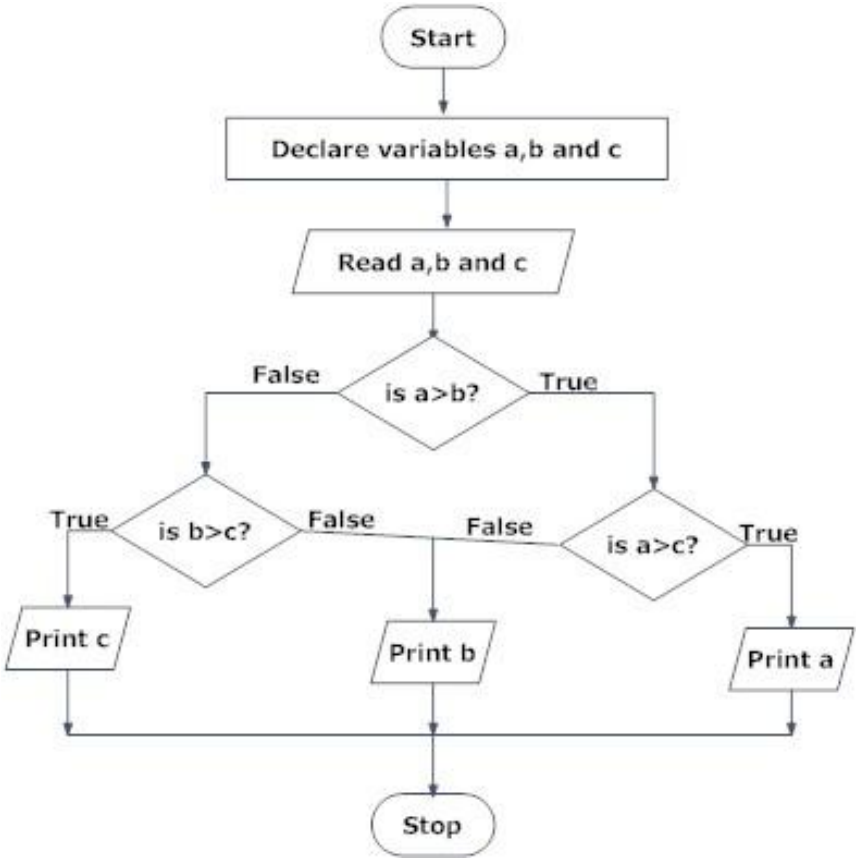
Example 1:

Draw a flowchart to add two numbers entered by user.



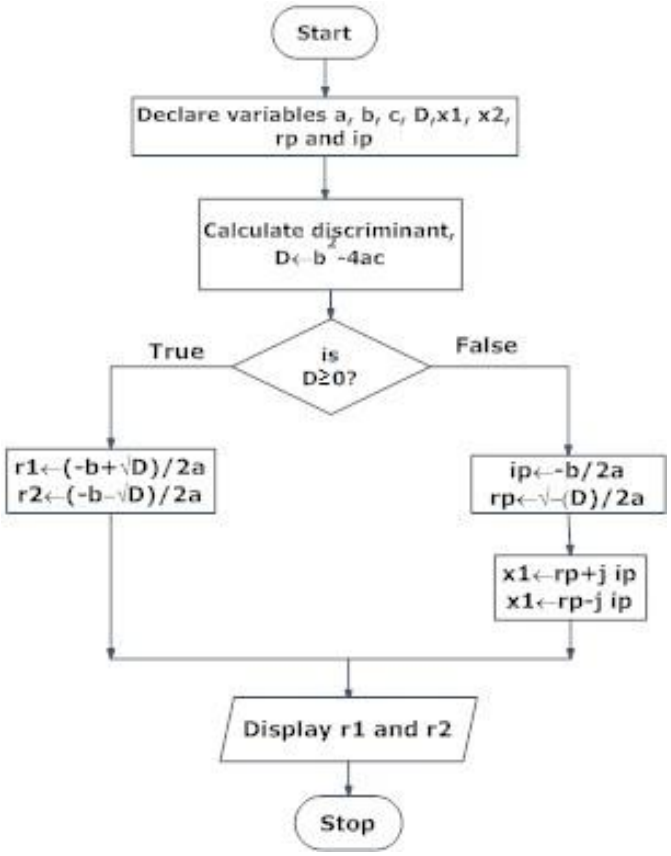
Example 2:

Draw a flowchart to find the largest among three different numbers entered by user.

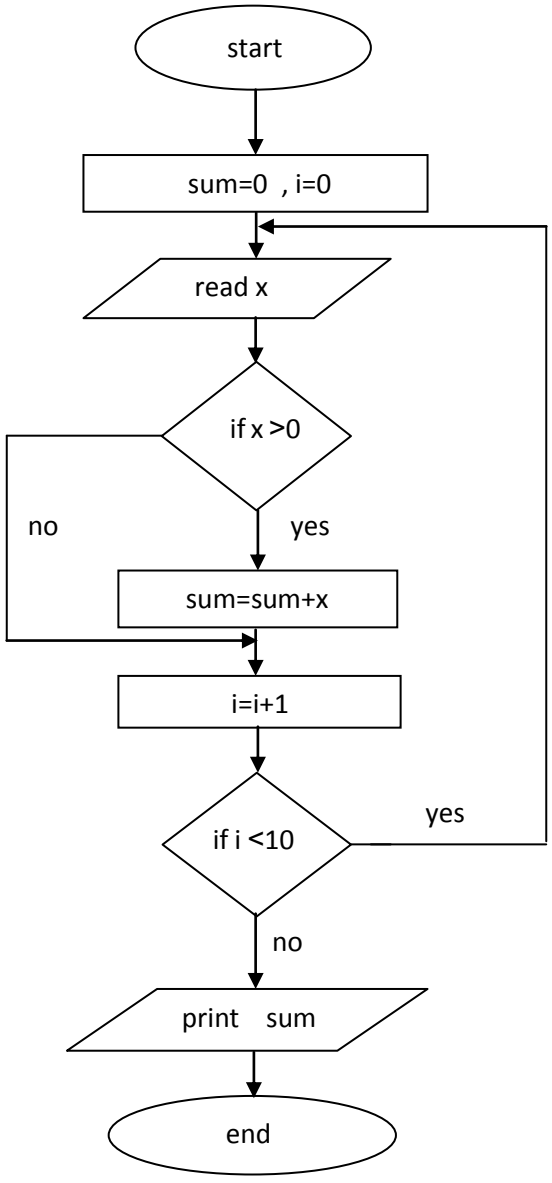


Example 3:

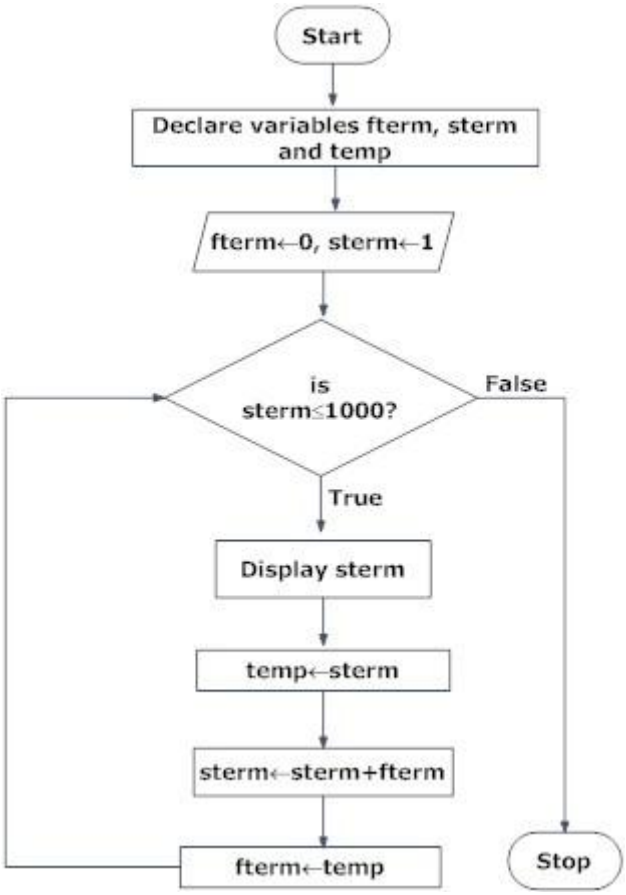
Draw a flowchart to find all the roots of a quadratic equation $ax^2+bx+c=0$



Example 4:
Draw a flowchart to find the summation of positive numbers between ten numbers



Example 5:
Draw a flowchart to find the Fibonacci series till term≤1000.



Lecture 4

C++ Language Basics:

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

- Object - Objects have states and behaviors. Example: A dog has states -
color, name, breed as well as behaviors - wagging, barking, and eating.
An object is an instance of a class.
- Class - A class can be defined as a template/blueprint that describes the
behaviors/states that object of its type support.
- Methods - A method is basically a behavior. A class can contain many
methods. It is in methods where the logics are written, data is
manipulated and all the actions are executed.
- Instant Variables - Each object has its unique set of instant variables.
An object's state is created by the values assigned to these instant
variables.

C++ Program Structure:
Let us look at a simple code that would print the words *Hello World*.

```
#include <iostream>
using namespace std;
// main() is where program execution begins.
int main()
{
cout << "Hello World"; // prints Hello World
return 0;
}
```

Let us look at the various parts of the above program:

1. The C++ language defines several headers, which contain information
that is either necessary or useful to your program. For this program, the
header <iostream> is needed.

2. The line using namespace std; tells the compiler to use the std
namespace. Namespaces are a relatively recent addition to C++.

3. The next line '// main() is where program execution begins.' is a
single-line comment available in C++. Single-line comments begin with //
and stop at the end of the line.

4. The line int main() is the main function where program execution begins.

5. The next line cout << "This is my first C++ program."; causes the
message "This is my first C++ program" to be displayed on the screen.

6. The next line return 0; terminates main() function and causes it to return the value 0 to the calling process.

Compile & Execute C++ Program:

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open a text editor and add the code as above.
2. Save the file as: hello.cpp
3. Open a command prompt and go to the directory where you saved the
file.
4. Type 'g++ hello.cpp' and press enter to compile your code. If there are no
errors in your code the command prompt will take you to the next line
and would generate a.out executable file.

5. Now, type 'a.out' to run your program.
6. You will be able to see ' Hello World ' printed on the window.

```
$ g++ hello.cpp
$ ./a.out
Hello World
```

Make sure that g++ is in your path and that you are running it in the directory
containing file hello.cpp.
You can compile C/C++ programs using makefile. For more details, you can
check our ‘Makefile Tutorial’.
Semicolons & Blocks in C++
In C++, the semicolon is a statement terminator. That is, each individual
statement must be ended with a semicolon. It indicates the end of one logical
entity.
For example, following are three different statements:

```
x = y;
y = y+1;
```

C++
8

```
add(x, y);
```

A block is a set of logically connected statements that are surrounded by
opening and closing braces. For example:

```
{
cout << "Hello World"; // prints Hello World
return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it
does not matter where you put a statement in a line. For example:

```
x = y;
y = y+1;
add(x, y);
```

is the same as

```
x = y; y = y+1; add(x, y);
```

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or
any other user-defined item. An identifier starts with a letter A to Z or a to z or
an underscore (_) followed by zero or more letters, underscores, and digits (0 to

9).
C++ does not allow punctuation characters such as @, $, and % within
identifiers. C++ is a case-sensitive programming language.
Thus, **Manpower** and **manpower** are two different identifiers in C++.
Here are some examples of acceptable identifiers:

```
mohd zara abc move_name a_123
myname50 _temp j a23b9 retVal
```

C++ Keywords
The following list shows the reserved words in C++. These reserved words may not be used as constant or variable

Lecture 5:-

Variables

Data values in a program are usually stored in ***variables***, which are named locations in memory capable of holding a particular data type. You have already seen examples of variables in the **PowersOfTwo** program and are almost certainly familiar with the basic concept from your earlier programming experience. The purpose of this section is to outline the rules for using variables in C++.

Variable declarations

In C++, you must ***declare*** each variable before you use it. The primary function of declaring a variable is to make an association between the ***name*** of the variable and the ***type*** of value that variable contains. The placement of the declaration in a program also determines the ***scope*** of the variable, which is the region in which that variable is accessible.

The usual syntax for declaring a variable is

*type namelist***;**

where *type* indicates the data type and *namelist* is a list of variable names separated by commas. In most cases, each declaration introduces a single variable name. For example, the function **main** in **PowersOfTwo** begins with the line

int limit;

which declares the variable **limit** to be of type **int**. You can, however, declare several variable names at once, as in the following declaration, which declares three variables named **n1**, **n2**, and **n3**:

double n1, n2, n3;

In this case, the variables are each declared to be of type **double**, which is the type C++ uses to represent numbers that can have fractional parts. The name *double* is

short for *double-precision floating-point,* but there is no reason to worry about what all those terms mean.

Data types

Each variable in a C++ program contains a value constrained to be of a particular type. You set the type of the variable as part of the declaration. So far, you have seen variables of type **int** and **double**, but these types merely scratch the surface of the types available in C++. Programs today work with many different data types, some of which are built into the language and some of which are defined as part of a particular application. Learning how to manipulate data of various types is an essential part of mastering the basics of any language, including C++.

The concept of a data type

In C++ , every data value has an associated data type. From a formal perspective, a
data type is defined by two properties: a ***domain,*** which is the set of values that belong to that type, and a ***set of operations,*** which defines the behavior of that type. For example, the domain of the type **int** includes all integers
. . . -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 . . .
and so on, up to the limits established by the hardware of the machine. The set of operations applicable to values of type **int** includes, for example, the standard arithmetic operations like addition and multiplication. Other types have a different domain and set of operations.

As you will learn in the later chapters in this course, much of the power of modern programming languages like C++ comes from the fact that you can define new data types from existing ones. To get that process started, C++ includes several fundamental types that are defined as part of the language. These types, which act as the building blocks for the type system as a whole, are called ***atomic*** or ***primitive types***. These predefined types are grouped into five categories—integer, floating-point, Boolean, character, and enumerated types—which are discussed in the sections that follow.

Expressions

Whenever you want a program to perform calculations, you need to write an expression that specifies the necessary operations in a

form similar to that used for expressions in athematics. For example, suppose that you wanted to solve the quadratic equation
a*x*2 + b*x* + c = 0
As you know from high-school mathematics, this equation has two solutions given by the formula
x =
The first solution is obtained by using + in place of the ± symbol; the second is obtained by using – instead. In C++, you could compute the first of these solutions by writing the following expression:

```
(-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

There are a few differences in form: multiplication is represented explicitly by a *****, division is represented by a ***/***, and the square root function (which comes from a library called **<cmath>** described in detail in Chapter 2) is written using the name **sqrt** rather than a mathematical symbol. Even so, the C++ form of the expression captures the intent of its mathematical counterpart in a way that is quite readable, particularly if you've written programs in any modern programming language.
In C++, an expression is composed of terms and operators. A ***term***, such as the
variables **a**, **b**, and **c** or the constants 2 and 4 in the preceding expression, represents
a single data value and must be either a constant, a variable, or a function call. An
operator is a character (or sometimes a short sequence of characters) that indicates a
-*b* + *b*2 - 4*ac*
2*a*

computational operation. A list of the operators available in C++ appears in
Table 1-4. The table includes familiar arithmetic operators like **+** and **-** along with
several others that pertain only to types introduced in later chapters.