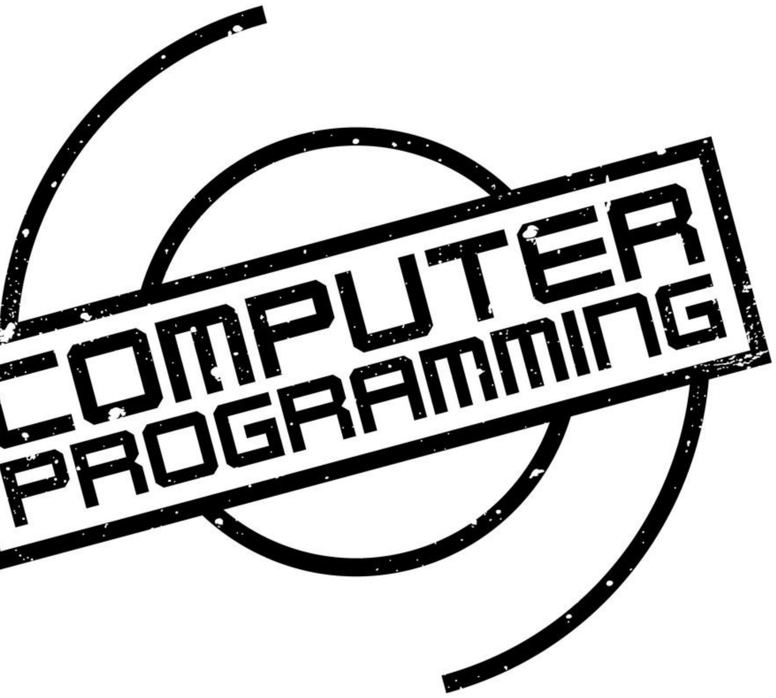
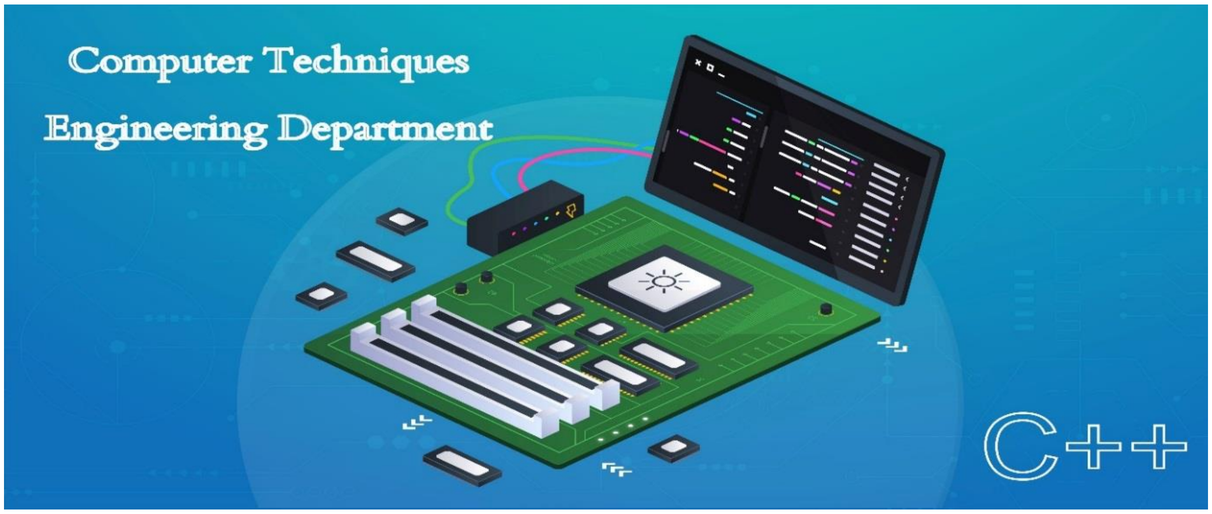
1st Week


COMPUTER
PROGRAMMING


Computer Techniques
Engineering Department
C++

Algorithms

Look around you. Computers and networks are everywhere, enabling an intricate web of complex human activities: education,

commerce, entertainment, research, manufacturing, health management, communication, even war. Of the two main technological underpinnings of this amazing proliferation, one is the breathtaking pace with which advances in microelectronics and chip design have been bringing us faster and faster hardware. The other is efficient algorithms that are fuelling the computer revolution.

In mathematics, computer science, and related subjects, an *algorithm* is a finite sequence of steps expressed for solving a problem. An *algorithm* can be defined as “a process that performs some sequence of operations in order to solve a given problem”. Algorithms are used for calculation, data processing, and many other fields.

In computing, algorithms are essential because they serve as the systematic procedures that computers require. A good algorithm is like using the right tool in the workshop. It does the job with the right amount of effort. Using the wrong algorithm or one that is not clearly defined is like trying to cut a piece of plywood with a pair of scissors: although the job may get done, you have to wonder how effective you were in completing it.

Let us follow an example to help us understand the concept of algorithm in a better way. Let’s say that you have a friend arriving at the railway station, and your friend needs to get from the railway station to your house. Here are three different ways (algorithms) that you might give your friend for getting to your home.

The taxi/auto-Basrah algorithm:

Go to the taxi/auto- Basrah stand.

Get in a taxi/auto- Basrah.

Give the driver my address.

The call-me algorithm:

When your train arrives, call my mobile phone.

Meet me outside the railway station.

The bus algorithm:

Outside the railway station, catch bus number 321.

Transfer to bus 308 near Basrah station.

Get off near Basrah University.

Walk two blocks west to my house.

In computer programming, there are often many different algorithms to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. Sorting is one place where a lot of research has been done, because computers spend a lot of time sorting lists.

Three reasons for using algorithms are *efficiency, abstraction* and *reusability.*

Efficiency: Certain types of problems, like sorting, occur often in computing. Efficient algorithms must be used to solve such problems considering the time and cost factor involved in each algorithm.

Abstraction: Algorithms provide a level of abstraction in solving problems because many seemingly complicated problems can be distilled into simpler ones for which well-known algorithms exist. Once we see a more complicated problem in a simpler light, we can think of the simpler problem as just an abstraction of the more complicated one. For example, imagine trying to find the shortest way to route a packet between two gateways in an internet. Once we realize that this problem is just a variation of the more general shortest path problem, we can solve it using the generalised approach.

Reusability: Algorithms are often reusable in many different situations. Since many well-known algorithms are the generalizations of more complicated ones, and since many complicated problems can be distilled into simpler ones, an efficient means of solving certain simpler problems potentially lets us solve many complicated problems.

1.Expressing Algorithms:

Algorithms can be expressed in many different notations, including *natural languages*, *pseudocode*, *flowcharts* and *programming languages*. *Natural language* expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or

technical algorithms. *Pseudocode* and *flowcharts* are structured ways to express algorithms that avoid many ambiguities common in natural language statements, while remaining independent of a particular implementation language. *Programming languages* are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used to define or document algorithms.

Sometimes it is helpful in the description of an algorithm to supplement small flowcharts with natural language and/or arithmetic expressions written inside block diagrams to summarize what the flowcharts are accomplishing.

Consider an example for finding the largest number in an unsorted list of numbers.The solution for this problem requires looking at every number in the list, but only once at each.

Algorithm using natural language statements: Assume the first item is largest. Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it. The last noted item is the largest item in the list when the process is complete.

```
Algorithm using
pseudocode: largest = L0
for each item in the list (Length(L) ≥
    1), do if the item ≥ largest, then
            largest = the
item return largest
```

2.Benefits of using Algorithms:

The use of algorithms provides a number of benefits. One of these benefits is in the *development of the procedure* itself, which involves identification of the processes, major decision points, and variables necessary to solve the problem. Developing an algorithm allows and even forces examination of the solution process in a rational manner. Identification of the processes and decision points reduces the task into a series of smaller steps of more manageable size. Problems that would be difficult or impossible to solve in entirety can be approached as a series of small, solvable sub-problems.

By using an algorithm, decision making becomes a more *rational process*. In addition to making the process more rational, use of algorithm will make the process more efficient and more consistent. *Efficiency* is an inherent result of the analysis and specification process. *Consistency* comes from both the use of the same specified process and increased skill in applying the process. An algorithm serves as a mnemonic device and helps ensure that variables or parts of the problem are not ignored. Presenting the solution process as an algorithm allows more precise communication. Finally, separation of the procedure steps facilitates division of labour and development of expertise.

A final benefit of the use of an algorithm comes from the *improvement* it makes possible. If the problem solver does not know what was done, he or she will not know what was done wrong. As time goes by and results are compared with goals, the existence of a specified solution process allows identification of weaknesses and errors in the process. Reduction of a task to a specified set of steps or algorithm is an important part of analysis, control and evaluation.

Before writing an algorithm for a problem, one should find out what is/are the inputs to the algorithm and what is/are expected output after running the algorithm. Now let us take some exercises to develop an algorithm for some simple problems.

While writing algorithms we will use following symbol for different operations:

‘+’ for Addition
‘*’ for Multiplication
‘$\longleftarrow$’ for assignment
‘-’ for Subtraction
‘/’ for Division

For example A $\longleftarrow$ X*3 means A will have a value of X*3.

EX.1: Write an algorithm to find the area of a circle of radius r ?

Area of a circle (A) = $\pi.r^2$

Inputs to the algorithm:

Radius (r)of the circle.

Expected output:

Area (A) of the circle.

Algorithm:

```
  Step1: Start
  Step2: Read\input the radius r of the circle
  Step3: A ⟵ PI* r * r
  Step4: Print Area (A)
  Step5: End
```

EX.2: Write an algorithm to read two numbers and find their sum?

Inputs to the algorithm:

First num1.

Second num2.

Expected output:

Sum of the two numbers.

Algorithm:

```
  Step1: Start
  Step2: Read\input the first number (num1)
  Step3: Read\input the second number(num2)
  Step4: Sum ⟵ num1+num2
  Step5: Print Sum
  Step6: End
```

EX3: Write an algorithm to convert Fahrenheit (F) temperature to Celsius (C) where $C = \frac{5}{9} * (F - 32)$

Inputs to the algorithm:

Temperature in Fahrenheit (F)

Expected output:

Temperature in Celsius (C)

Algorithm:

```
Step1: Start
Step 2: Read Temperature in Fahrenheit F
Step 3: C⟵ 5/9*(F - 32)
Step 4: Print Temperature in Celsius(C)
Step5: End
```

Control Structures of Algorithms

The control structures of algorithms can be summarized as follows:

1. Sequence

The sequence is exemplified by sequence of statements place one after the other - the one above or before another gets executed first.

2. Branching or Selection

The branch refers to a binary decision based on some condition. If the condition is true, one of the two branches is explored; if the condition is false, the other alternative is taken.

EX1: write an algorithm to find the greater number between two input numbers?

Inputs to the algorithm:
Tow numbers (A) and (B)
Expected output:
The greater number (C)
Algorithm:

```
Step1: Start
Step2: Read/input A and B
Step3: If A greater than or equal B then C⟵A
Step4: If B greater than or equal A then C⟵B
Step5: Print C
Step6: End
```

EX2: Write an algorithm to find the result of equation:

$$F = \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$$

Inputs to the algorithm:
The value of (x)
Expected output:
The result of (F)
Algorithm:

```
Step1: Start
Step2: Read/input x
Step3: If X greater than or equal zero then F⟵X
Step4: If X less than zero then F⟵X
Step5: Print F
Step6: End
```

***Homework**

EX3: Write an algorithm to find the largest value of any three input numbers?

3. Loop or Repetition

The loop allows a statement or a sequence of statements to be repeatedly executed based on some loop condition. It is represented by the ‘while’ and ‘for’ constructs in most programming languages.

You must ensure that the condition for the termination of the looping must be satisfied after some finite number of iterations, otherwise it ends up as an infinite loop, a common mistake made by inexperienced programmers. The loop is also known as the repetition structure.

EX1: write an algorithm to print even numbers between 0 and 99?

Expected output:

The even numbers between 0 and 99 (I)

Algorithm:

```
Step1: Start
Step2: I ← 0
Step3: Print I
Step4: I←I+2
Step5: If (I <=98) then go to line 3
Step6: End
```

***Homework**

EX2: Design an algorithm which inputs a natural value, n, and prints odd numbers equal or less than n?

EX3: Design an algorithm which generates even numbers between 1000 and 2000 and then prints them in the standard output. It should also print total sum of these numbers?

Expected output:
The even numbers (I)
The summation of the even numbers (S)
Algorithm:

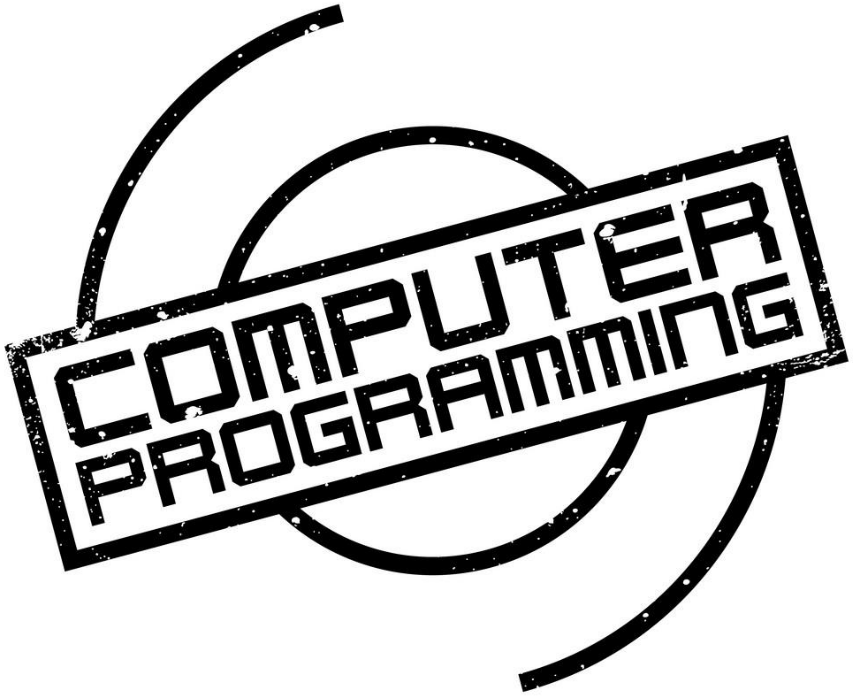
```
Step1: Start
Step2: I ← 1000 and S ← 0
Step3: Print I
Step4: S←S+I
Step5: I←I+2
Step6: If (I <= 2000) then go to line 3 else go to line 7
Step7: Print S
Step8: End
```

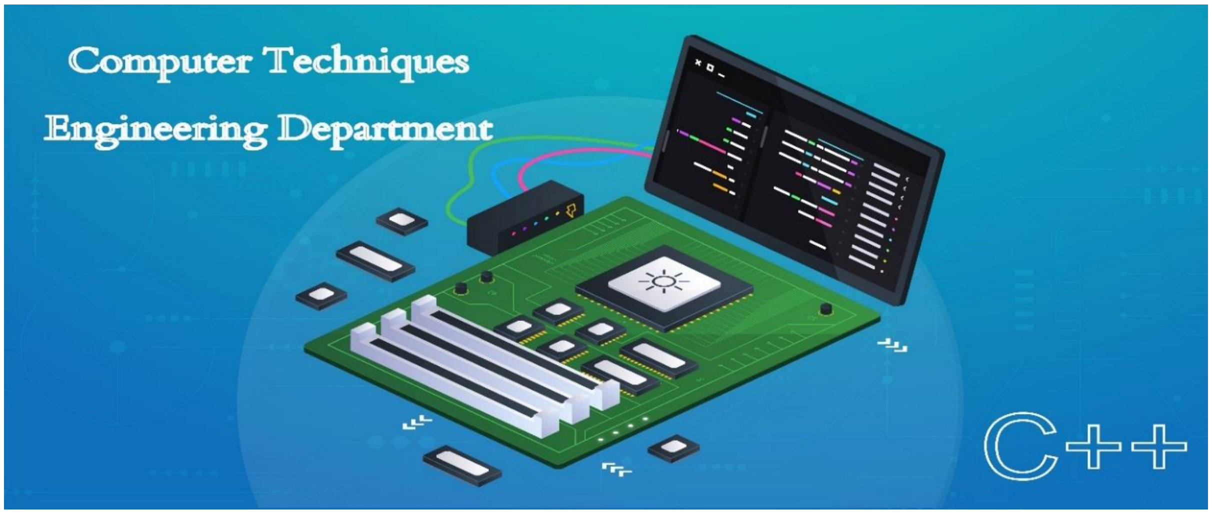
EX4: Design an algorithm with a natural number, n, as its input which calculates the following formula and writes the result in the standard output:

$$S = \frac{1}{2} + \frac{1}{4} + \ldots + 1/n$$

Inputs to the algorithm:
The value of (n)
Expected output:
The result of (S)
Algorithm:

```
Step1: Start
Step2: Read n
Step3: I ← 2 and S ← 0
Step4: S←S+1/I
Step5: I←I+2
Step6: If (I <= n) then go to line 4 else Print S
Step7: End
```


COMPUTER
PROGRAMMING


Computer Techniques
Engineering Department
C++

Flowcharts

A *Flowchart* is a type of diagram (graphical or symbolic) that represents an algorithm or process. Each step in the process is represented by a different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction. A flowchart typically shows the flow of data in a process, detailing the operations/steps in a pictorial format which is easier to understand than reading it in a textual format.

A flowchart describes what operations (and in what sequence) are required to solve a given problem. A flowchart can be likened to the blueprint of a building. As we know a designer draws a blueprint before starting construction on a building. Similarly, a programmer prefers to draw a flowchart prior to writing a computer program. Flowcharts are a pictorial or graphical representation of a process. The purpose of all flow charts is to communicate how a process works or should work without any technical or group specific jargon.

Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts often facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

1. Advantages of Using Flowcharts:

The benefits of *flowcharts* are as follows:

Communication: Flowcharts are better way of communicating the logic of a system to all concerned.

Effective analysis: With the help of flowchart, problem can be analysed in more effective way.

Proper documentation: Program flowcharts serve as a good program documentation, which is needed for various purposes.

Efficient Coding: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.

Proper Debugging: The flowchart helps in debugging process.

Efficient Program Maintenance: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

2. When to Use a Flowchart:

To communicate to others how a process is done.

A flowchart is generally used when a new project begins in order to plan for the project.

A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process, while drawing clear lines between where one process ends and the next one starts.

Developing a flowchart stimulates communication among participants and establishes a common understanding about the process. Flowcharts also uncover steps that are redundant or misplaced.

Flowcharts are used to help team members, to identify who provides inputs or resources to whom, to establish important areas for monitoring or data collection, to identify areas for improvement or increased efficiency, and to generate hypotheses about causes.

It is recommended that flowcharts be created through group discussion, as individuals rarely know the entire process and the communication contributes to improvement.

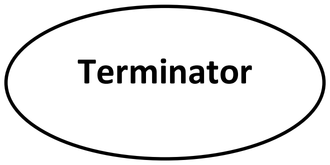
Flowcharts are very useful for documenting a process (simple or complex) as it eases the understanding of the process.

Flowcharts are also very useful to communicate to others how a process is performed and enables understanding of the logic of a process.

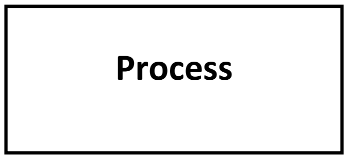
3. Flowchart Symbols & Guidelines:

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for flowcharting many computer programs are shown.

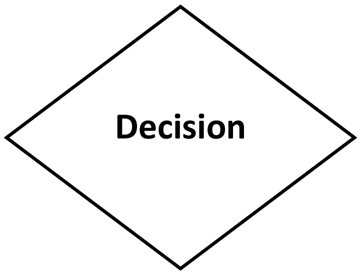
Terminator: An oval flow chart shape indicates the start or end of the process, usually containing the word “Start” or “End”.



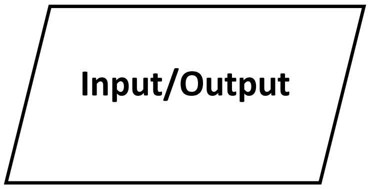
Process: A rectangular flow chart shape indicates a normal/generic process flow step. For example, “Add 1 to X”, “M = M*F” or similar.



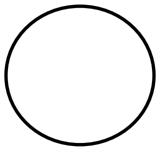
Decision: A diamond flow chart shape indicates a branch in the process flow. This symbol is used when a decision needs to be made (e.g. IF/THEN/ELSE), commonly a Yes/No question or True/False test.



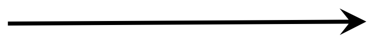
Data: A parallelogram that indicates data input or output (I/O) for a process. Examples: Get X from the user, Display X.



Connector: A small, labelled, circular flow chart shape used to indicate a jump in the process flow. Connectors are generally used in complex or multi-sheet diagrams.



Arrow: used to show the flow of control in a process. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.



These are the basic symbols used generally. Now, the ***basic guidelines*** for drawing a flowchart with the above symbols are that:

- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
-
- The flowchart should be neat, clear and easy to follow. There should not be any room for ambiguity in understanding the flowchart.

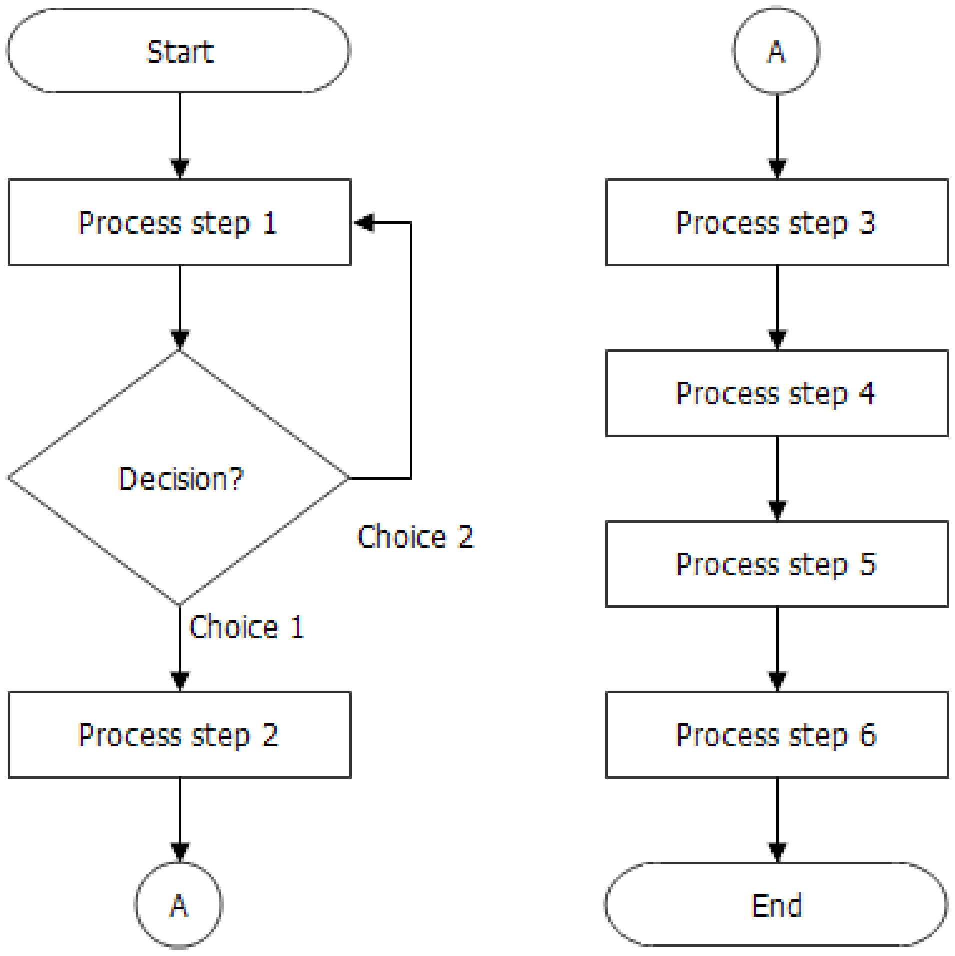
- The flowchart is to be read left to right or top to bottom.

- A process symbol can have only one flow line coming out of it.

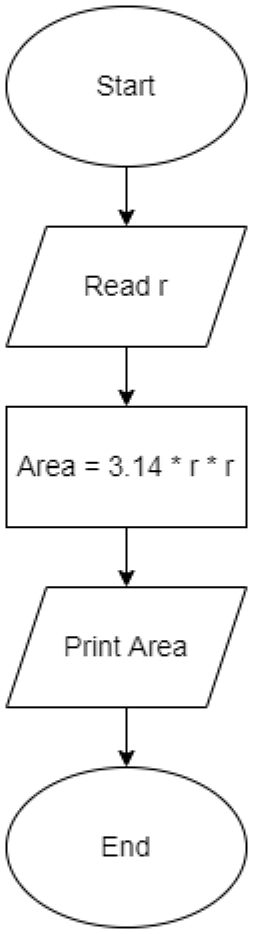
- For a decision symbol, only one flow line can enter it, but multiple lines can leave it to denote possible answers.

- The terminal symbols can only have one flow line in conjunction with them.

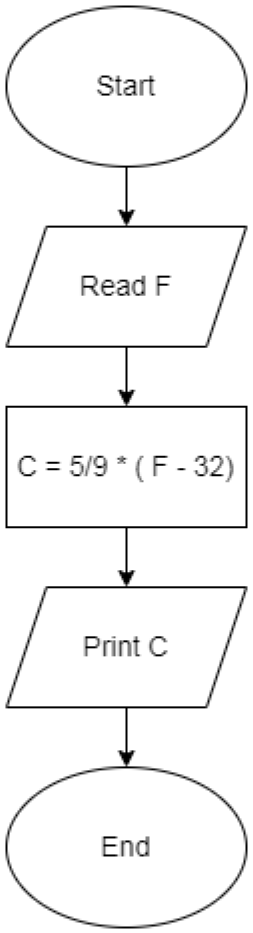
Basic Flowchart



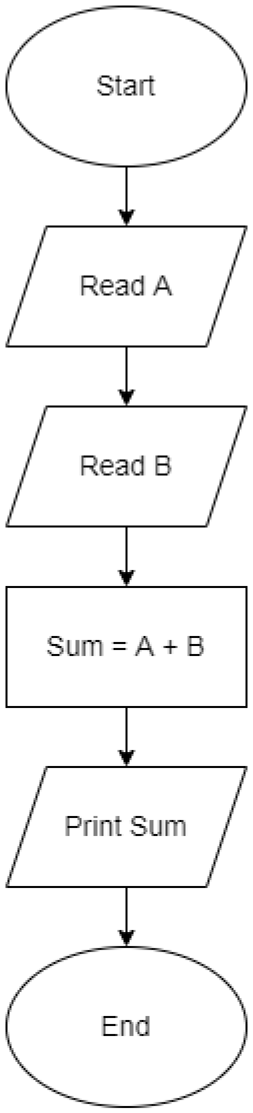
EX1: Draw a flow chart to find the area of a circle of radius r?



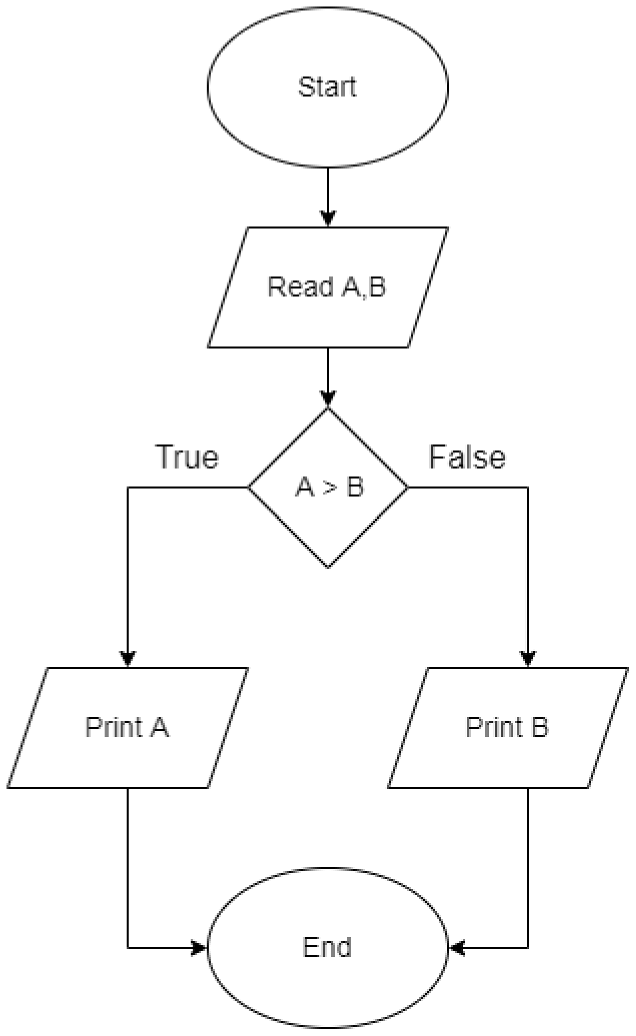
EX2: Draw a flow chart to convert temperature Fahrenheit to Celsius?



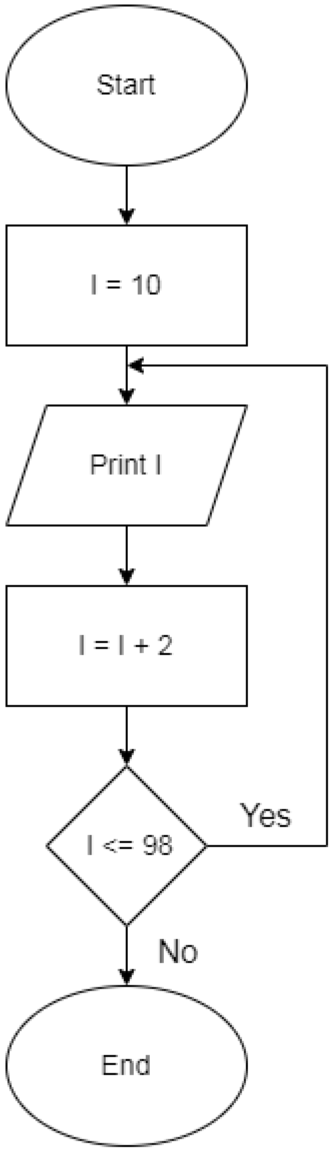
EX3: Draw a flow chart for inputs two numbers and prints sum of their value?



EX4: Draw a flow chart to find the greater number between two numbers?



EX5: Draw a flow chart for printing even numbers between 9 and 98?



*Homework

EX6: Draw a flow chart for the problem of printing odd numbers less than or equal an input number? It should also print their sum and count?

EX7: Draw a flow chart for the print the average from 25 input exam scores?

Work Sheets

For all following question write an algorithm and then draw the flow chart?

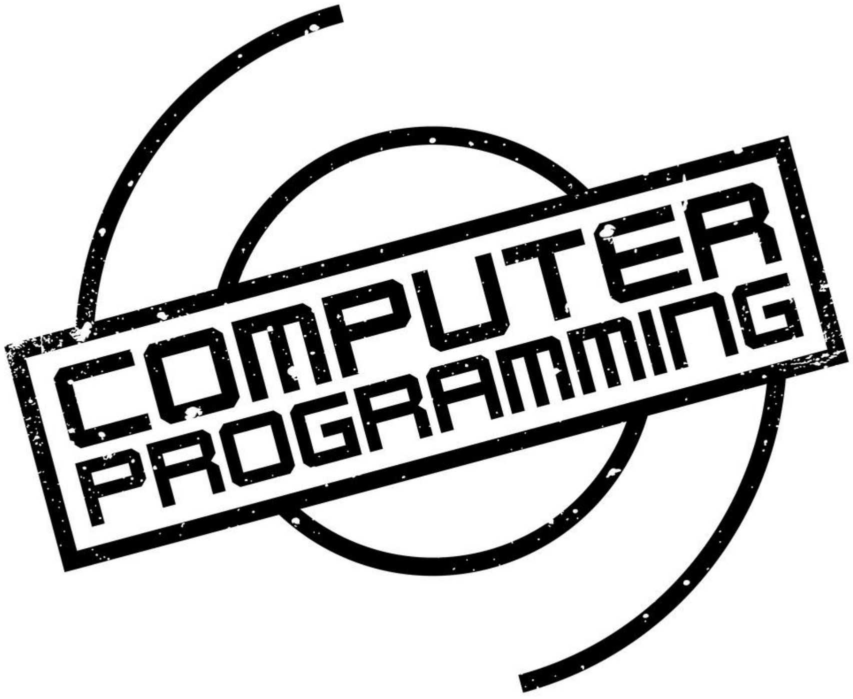
1. Find the multiplication of 10 numbers entered by the user?

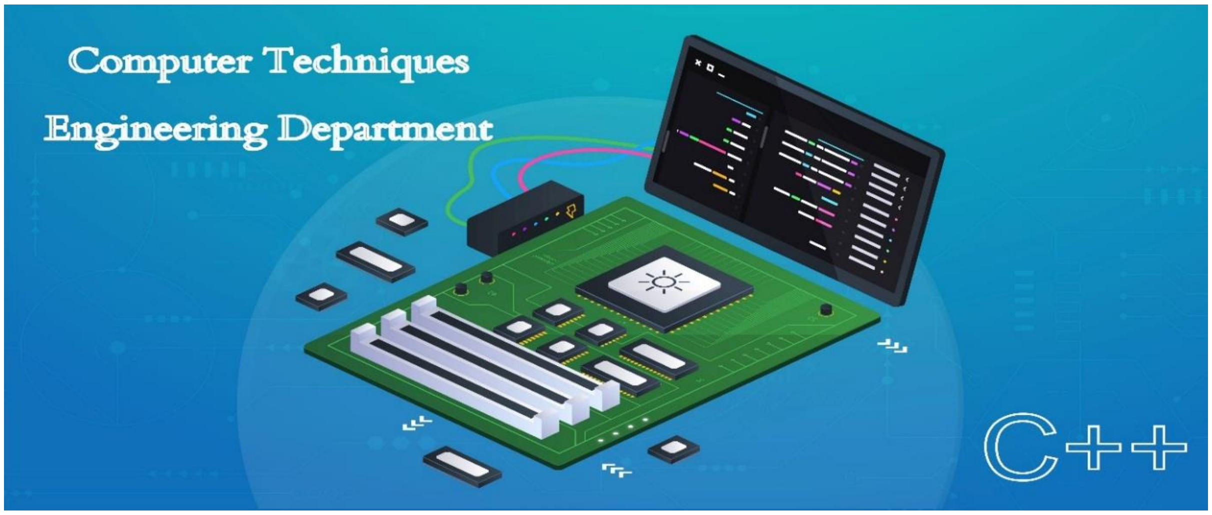
2. Find the sum of only the negative numbered among 25 numbers entered by the user?

3. Read 3 numbers: x , y and z and print the largest number of them.

4. Lengths of three sides of a triangle a, b, c are given as input.
   Find if the triangle is isosceles, equilateral, or scalene.

Hint: In an equilateral triangle three sides are equal.
In an isosceles triangle two sides are equal.
In a scalene triangle three sides are not equal.


COMPUTER
PROGRAMMING


Computer Techniques
Engineering Department
C++

Flowcharts

A *Flowchart* is a type of diagram (graphical or symbolic) that represents an algorithm or process. Each step in the process is represented by a different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction. A flowchart typically shows the flow of data in a process, detailing the operations/steps in a pictorial format which is easier to understand than reading it in a textual format.

A flowchart describes what operations (and in what sequence) are required to solve a given problem. A flowchart can be likened to the blueprint of a building. As we know a designer draws a blueprint before starting construction on a building. Similarly, a programmer prefers to draw a flowchart prior to writing a computer program. Flowcharts are a pictorial or graphical representation of a process. The purpose of all flow charts is to communicate how a process works or should work without any technical or group specific jargon.

Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts often facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

1. Advantages of Using Flowcharts:

The benefits of *flowcharts* are as follows:

Communication: Flowcharts are better way of communicating the logic of a system to all concerned.

Effective analysis: With the help of flowchart, problem can be analysed in more effective way.

Proper documentation: Program flowcharts serve as a good program documentation, which is needed for various purposes.

Efficient Coding: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.

Proper Debugging: The flowchart helps in debugging process.

Efficient Program Maintenance: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part.

2. When to Use a Flowchart:

To communicate to others how a process is done.

A flowchart is generally used when a new project begins in order to plan for the project.

A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process, while drawing clear lines between where one process ends and the next one starts.

Developing a flowchart stimulates communication among participants and establishes a common understanding about the process. Flowcharts also uncover steps that are redundant or misplaced.

Flowcharts are used to help team members, to identify who provides inputs or resources to whom, to establish important areas for monitoring or data collection, to identify areas for improvement or increased efficiency, and to generate hypotheses about causes.

It is recommended that flowcharts be created through group discussion, as individuals rarely know the entire process and the communication contributes to improvement.

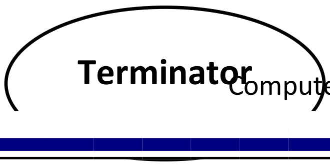
Flowcharts are very useful for documenting a process (simple or complex) as it eases the understanding of the process.

Flowcharts are also very useful to communicate to others how a process is performed and enables understanding of the logic of a process.

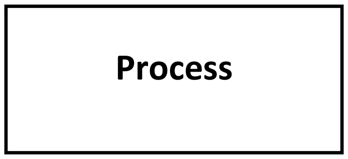
3. Flowchart Symbols & Guidelines:

Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for flowcharting many computer programs are shown.

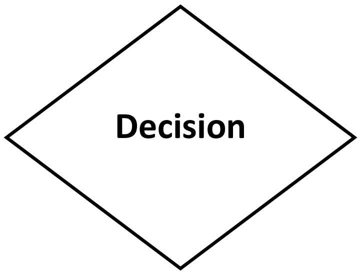
Terminator: An oval flow chart shape indicates the start or end of the process, usually containing the word “Start” or “End”.



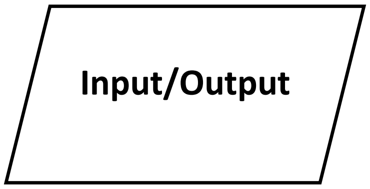
Process: A rectangular flow chart shape indicates a normal/generic process flow step. For example, "Add 1 to X", "M = M*F" or similar.



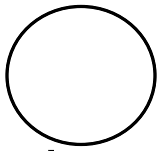
Decision: A diamond flow chart shape indicates a branch in the process flow. This symbol is used when a decision needs to be made (e.g. IF/THEN/ELSE), commonly a Yes/No question or True/False test.



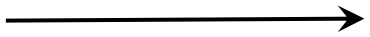
Data: A parallelogram that indicates data input or output (I/O) for a process. Examples: Get X from the user, Display X.



Connector: A small, labelled, circular flow chart shape used to indicate a jump in the process flow. Connectors are generally used in complex or multi-sheet diagrams.



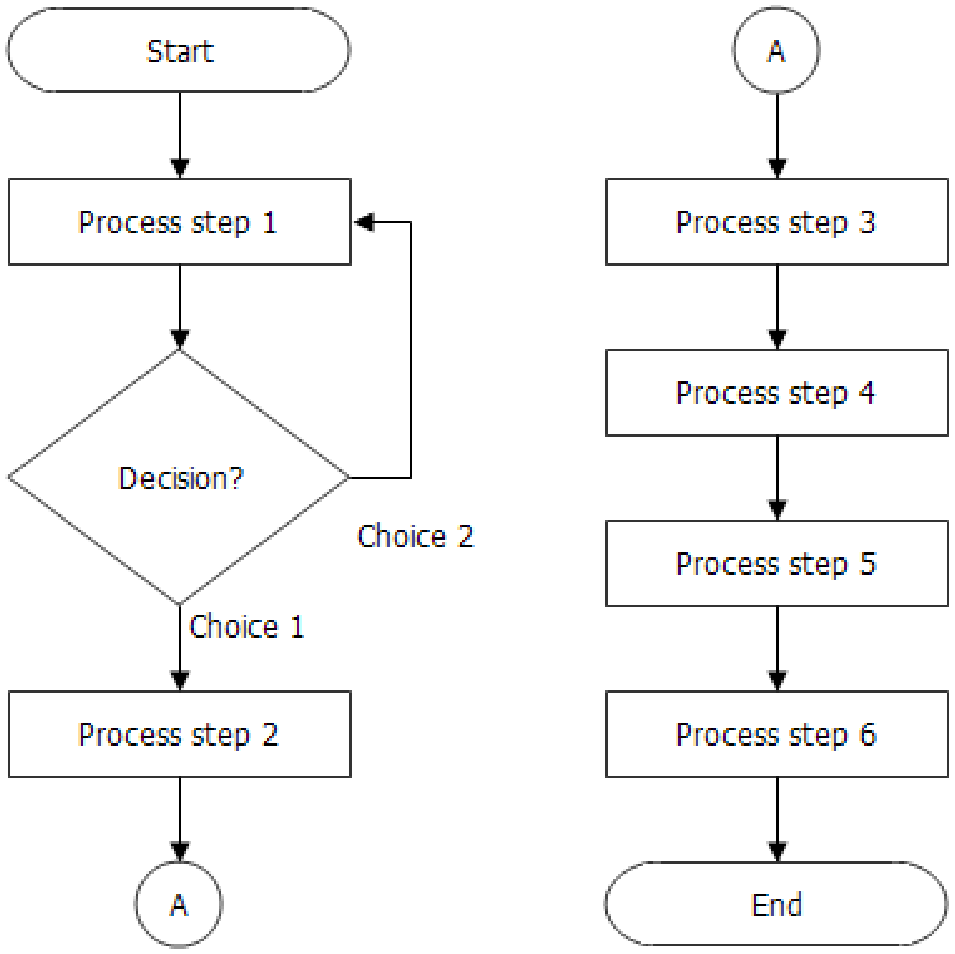
Arrow: used to show the flow of control in a process. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.



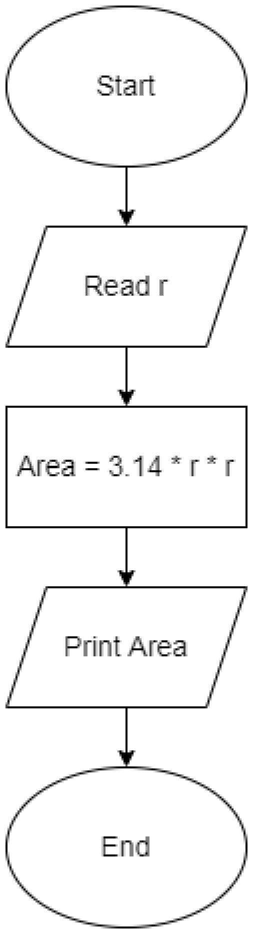
These are the basic symbols used generally. Now, the ***basic guidelines*** for drawing a flowchart with the above symbols are that:

- In drawing a proper flowchart, all necessary requirements should be listed out in logical order.
-
- The flowchart should be neat, clear and easy to follow. There should not be any room for ambiguity in understanding the flowchart.
- The flowchart is to be read left to right or top to bottom.
- A process symbol can have only one flow line coming out of it.
- For a decision symbol, only one flow line can enter it, but multiple lines can leave it to denote possible answers.
- The terminal symbols can only have one flow line in conjunction with them.

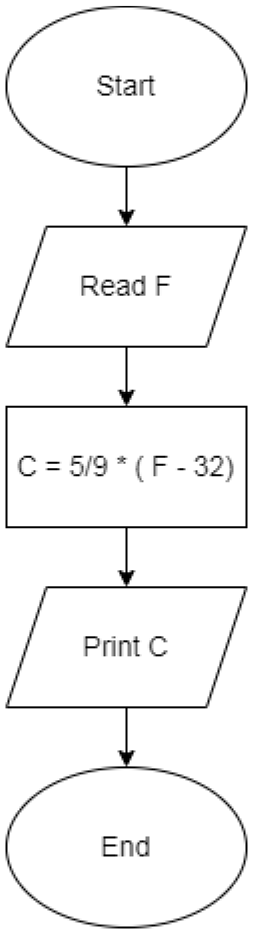
Basic Flowchart



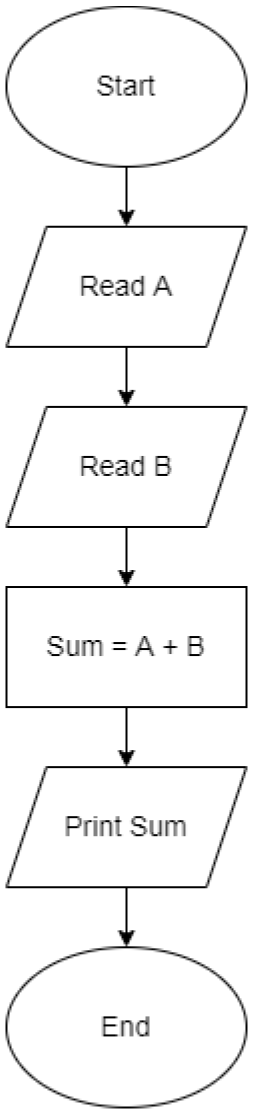
EX1: Draw a flow chart to find the area of a circle of radius r?



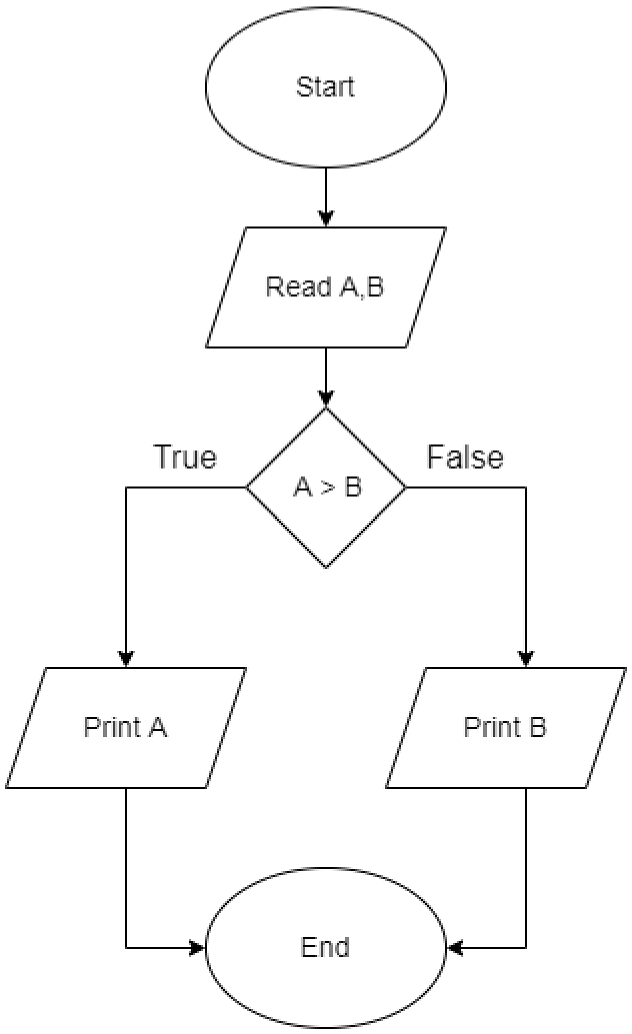
EX2: Draw a flow chart to convert temperature Fahrenheit to Celsius?



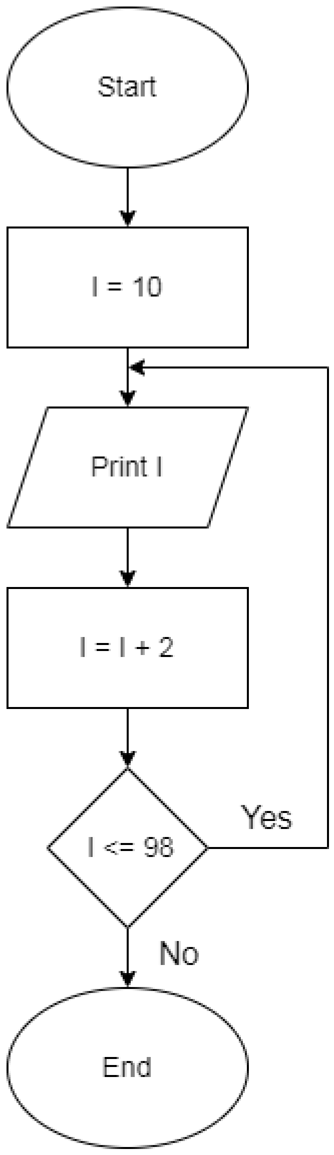
EX3: Draw a flow chart for inputs two numbers and prints sum of their value?



EX4: Draw a flow chart to find the greater number between two numbers?



EX5: Draw a flow chart for printing even numbers between 9 and 98?



*Homework

EX6: Draw a flow chart for the problem of printing odd numbers less than or equal an input number? It should also print their sum and count?

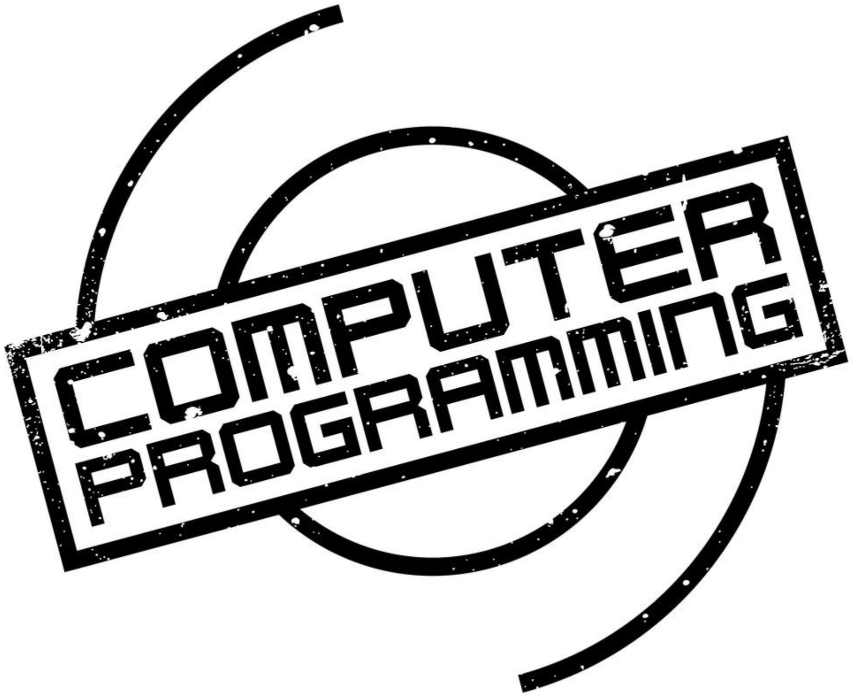
EX7: Draw a flow chart for the print the average from 25 input exam scores?

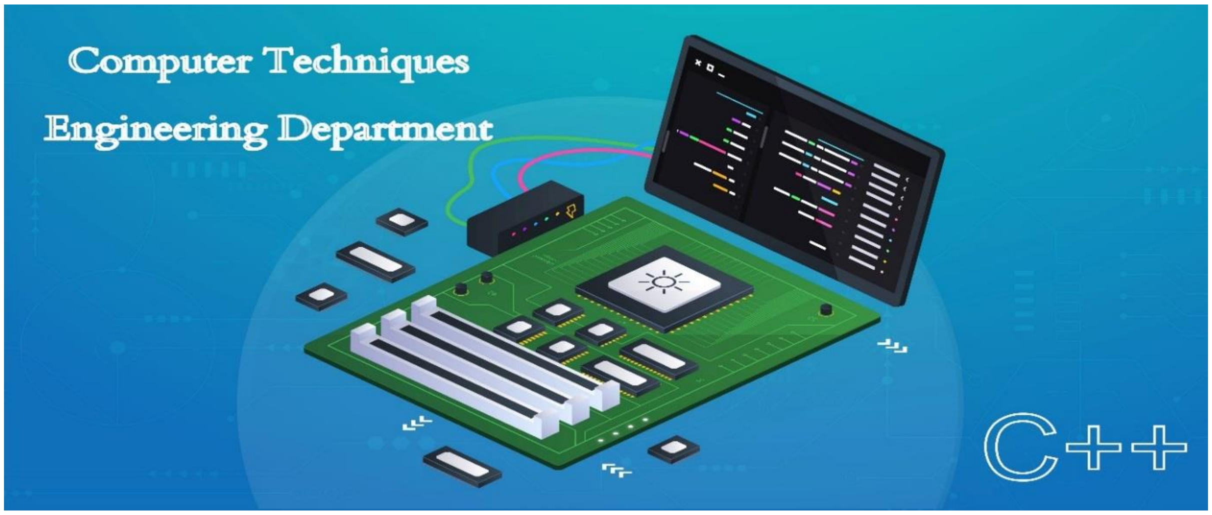
Work Sheets

For all following question write an algorithm and then draw the flow chart?

1. Find the multiplication of 10 numbers entered by the user?
2. Find the sum of only the negative numbered among 25 numbers entered by the user?
3. Read 3 numbers: x , y and z and print the largest number of them.
4. Lengths of three sides of a triangle a, b, c are given as input.
   Find if the triangle is isosceles, equilateral, or scalene.

Hint: In an equilateral triangle three sides are equal.
In an isosceles triangle two sides are equal.
In a scalene triangle three sides are not equal.


COMPUTER
PROGRAMMING


Computer Techniques
Engineering Department
C++

Introduction to C++ (Structure of a program)

The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details. 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. For this, an appreciation of programming and design techniques is far more important than an understanding of details; that understanding comes with time and practice.

C++ is an object oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++.

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is "close to the machine" so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The second purpose ideally requires a language that is "close to the problem to be solved" so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed keeping these purposes in mind.

C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. C++ was developed as an enhancement to the C language and originally named C with classes.

The main source of inspiration for C++ was C and hence it was initially called C with classes. C is retained as a subset and also C's emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks.

The other main source of inspiration was Simula67; the class concept (with derived classes and virtual functions) was borrowed from it. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68. Templates were partly inspired by Ada's generics (both their strengths and weaknesses) and partly by Clu's parameterized modules. Similarly, the C++ exception handling mechanism was inspired partly by Ada, Clu, and ML.

The name C++ (pronounced ‘‘see plus plus’’) was coined by Rick Mascitti in 1983. The name signifies the evolutionary nature of the changes from C; ‘‘++’’ is the C increment operator. The slightly shorter name ‘‘C+’’ is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is also not called D, because it is an extension of C, and it does not attempt to remedy problems by removing features.

After years of development, the C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. That standard is still current, but is amended by the 2003 technical corrigendum, ISO/IEC 14882:2003.

As one of the most popular programming languages ever created, C++ is used by hundreds of thousands of programmers in essentially every application domain. Some of its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games. C++ has greatly influenced many other popular programming languages, most notable C# and Java. C++ is also used for hardware design, where design is initially described in C++. Below a few of its applications are mentioned elaborately:

Early applications tended to have a strong systems programming flavor. For example, several *major operating systems* have been written in C++ and many more have key parts done in C++.

C++ provides uncompromising low-level efficiency essential for device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed.

Most applications have sections of code that are critical for acceptable performance. For most code, maintainability, ease of extension, and ease of testing is key. C++'s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over

time. Examples are banking, trading, insurance, telecommunications, and military applications.

Graphics and user interfaces are areas in which C++ is heavily used. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are C++ programs. In addition, some of the most popular libraries supporting X for UNIX are written in C++. Thus, C++ is a common choice for the vast number of applications in which the user interface is a major part.

A SIMPLE C++ PROGRAM

Before looking at how to write C++ programs consider the following simple example program

```
// my first program in C++

/* my hello world program in C++ with

   More comments */

#include <iostream>
using namespace std;

int main ()
 {
    cout << "Hello World!";
   return 0;
 }
```

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!"

sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

```
// my first program in C++
```

Or

```
/* my hello world program in C++ with
   More comments*/
```

This is a comment line and comment block. All lines beginning with two slash signs (//) will represent comment line. Also, all lines beginning with slash followed by star and ended up with start followed by slash will represent comment block (/* */). Both comment line and comment block considered comments and do not have any effect on the behaviour of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

```
#include <iostream>
```

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive *#include<iostream>* tells the preprocessor to include the *iostream* standard file. This specific file (*iostream*) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

```
using namespace std;
```

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library.

```
int main ()
```

This line corresponds to the beginning of the definition of the *main* function. The *main* function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function. The word main is followed in the code by a pair of parentheses (()). That is because it is a function declaration. In C++, what differentiate a function declaration from other expressions are the parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

```
cout << "Hello World!";
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout**:** represents the **standard output** stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen). *cout* is declared in the *iostream* standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code. Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

```
return 0;
```

The *return* statement causes the main function to finish. *return* may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's pre-processor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into *cout*), which were all included within the block delimited by the braces ({}) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
 {
    cout << " Hello World!";
   return 0;
 }
```

We could have written:

```
int main () { cout << "Hello World!"; return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code. In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

At this stage let us consider the general format of a C++ program:

```
//Introductory comment
//file name, programmer, when written or modified
/* what program does……
……..
……..
………. */

#include <iostream>

using namespace std;

  int main()
  {
      constant declarations
      variable declarations
      executable statements
  }
```

Note that it makes complex programs much easier to interpret if, as above, closing braces (}) are aligned with the corresponding opening brace ({). However other conventions are used for the layout of braces in textbooks and other C++ programmers' programs. Also additional spaces, new lines etc. can also be used to make programs more readable. The important thing is to adopt one of the standard conventions and stick to it consistently.

The format given above is the general format of a C++ program. Although this structure will also include the class declaration and member functions definitions but we will come elaborate on that part later.

EX1: Write a C++ program that prints on screen the following sentences:

{****Shatt Al-arab University College***}

{* Computer Techniques Engineering Department*}

{**First Year**}

```
// Printing sentences
// Ex1, 4^th,5^th Week, created 2/8/2022
/*
   A C++ program that prints on screen the following sentences:
       {****Shatt Al-arab University College***}
         {* Computer Techniques Engineering Department*}
              {**First Year**}
*/
#include <iostream>
using namespace std;
int main()
{
    cout << "{****Shatt Al-arab University College***}" << endl;
    cout << "    {* Computer Techniques Engineering Department*}"
    <<endl;
    cout << "            {**First Year**}" << endl;
   return 0;
}
```

Cin: is the **standard input** device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age
cin >> age;
```

The first statement declared **a variable of type integer (int)** called age, and the second one waits for an input form cin (the keyboard) in order to store it in this integer variable.

You can also use ***cin*** to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to :

```
cin >> a;
cin >> b;
```

EX2: Write a C++ program that enter any integer value and find its double, the screen output should be like:

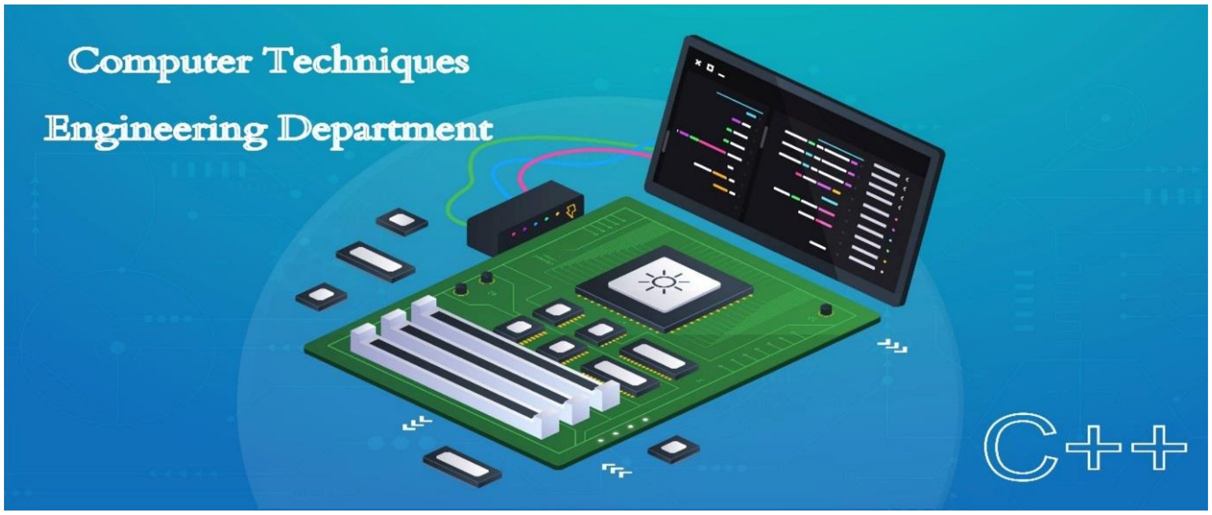
```
Please enter an integer value: 10

The value you entered is 10 and its double is 20
```

```
// Finding a double of any entered value
// Ex2 , 4^th,5^th  Week , created 2/8/2022 m....
/* A C++ program that enter any integer value
     and find its double, the screen output should be like:
           Please enter an integer value: 10
           The value you entered is 10 and its double is 20
*/

#include <iostream>
using namespace std;
int main ( )
  {
     int i;
     cout << "Please enter an integer value: ";
     cin >> i;
     cout << "The value you entered is " << i;
     cout << " and its double is " << i*2;
    return 0;
  }
```

COMPUTER PROGRAMMING



1. Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

- A type which is established when the variable is defined (e.g., integer, real, character). Once defined, the type of a C++ variable cannot be changed.

- A value which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

- A variable is used for the quantities which are manipulated by a computer program. For example a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.

Let us illustrate the uses of some simple variable.

```
int workDays;
float workHours, payRate, weeklyPay;
```

Suppose the above given part of the program aims to calculate the weekly pay using three variables workDays, workHours, and payRate.

In order to distinguish between different variables, they must be given **identifiers**. Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were weeklyPay, workDays, workHours and payRate, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

2. Identifiers

Identifiers are names given to variables which distinguish them from all other variables. The rules of C++ for valid identifiers state that:

An identifier must:

start with a letter

consist only of letters, the digits 0-9, or the underscore symbol
_

Not be a **reserved word**.

Reserved words are otherwise valid identifiers that have special significance to C++. For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, __, is forbidden.

The following are valid identifiers:

`Length,` `days_in_year,` `DataSet1,` `Profit95,` `_Pressure,` `first_one,` `first_1` **although using** `_Pressure` **is not recommended**.

The following are invalid:

```
days-in-year 1data int first.val throw
```

Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting such a program becomes more and more difficult. The minor typing effort of using ***meaningful*** identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

At this stage it is worth noting that C++ is **case-sensitive**. That is lower-case letters are treated as distinct from upper-case letters. Thus the word days in a program is quite different from the word Days or the word DAYS.

3. Reserved Words

The **syntax rules** (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the **reserved words**, must not be used for any other purposes. Reserved words are otherwise valid identifiers that have special significance to C++. All reserved words are in lower-case letters. The table below lists the reserved words of C++.

Asm	auto	bool	break	case
catch	char	class	const_cast	continue
default	delete	do	double	else
enum	dynamic_cast	extern	false	float
for	union	unsigned	using	friend
goto	if	inline	int	long
mutable	virtual	namespace	new	operator
private	protected	public	register	void
reinterpret_cast	return	short	signed	sizeof
static	static_cast	volatile	struct	switch
template	this	throw	true	try
typedef	typeid	unsigned	wchar_t	while

Some of these reserved words may not be treated as reserved by older compilers. However you would do well to avoid their use. Other compilers may add their own reserved words. Typical are those used by Borland compilers for the computer, which add `near`, `far`, `huge`, `cdecl`, and `pascal`.

Notice that `main` is *not* a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat `main`, `cin`, and `cout` as if they were reserved as well.

4. Data Types

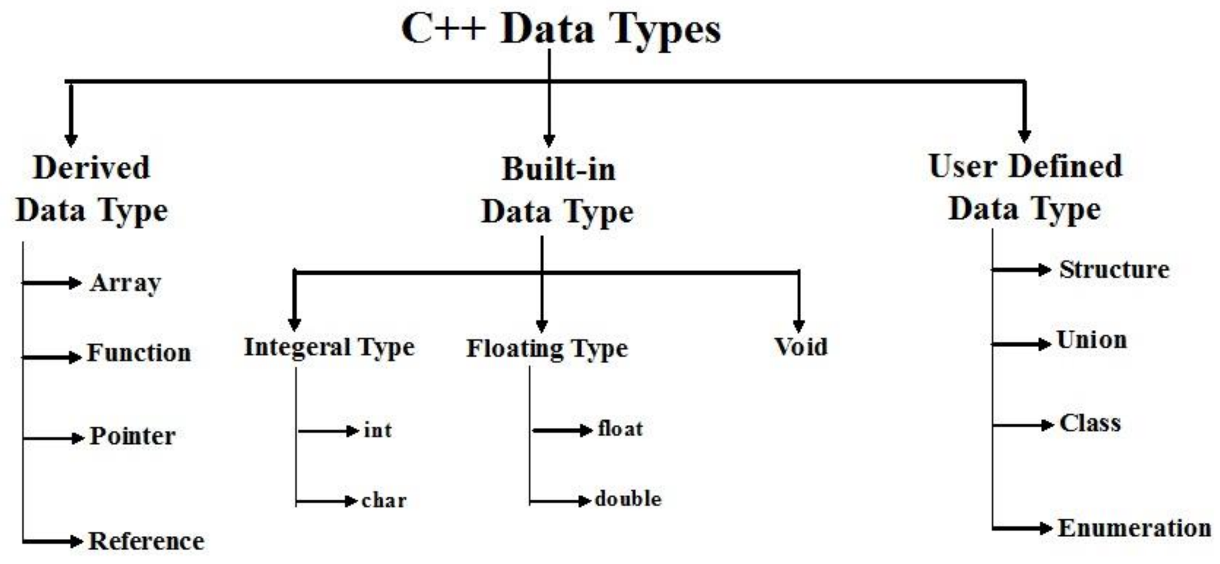
Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example, the declarations

```
float x;          //x is a floating-point variable
int y = 7;        /*y  is  an  integer variable  with
the initial value 7*/
float f(int);     /*f  is  a  function taking   an

argument  of  type  int  and  returning  a  floating-
point number */
```

would make the example meaningful. Because y is declared to be an int, it can be assigned to, used in arithmetic expressions, etc. On the other hand, f is declared to be a function that takes an int as its argument, so it can be called given a suitable argument.

Data types in Standard C++ are classified as shown in the diagram below.



The ***Boolean***, ***character***, and ***integer*** types are collectively called integral types. The ***integral*** and ***floating-point*** types are collectively called ***arithmetic*** types. ***Enumeration*** and structured types are called ***user-defined*** types because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, other types are called ***built-in*** types.

The ***integral*** and ***floating-point*** types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations. The assumption is that a computer provides bytes for holding characters and words, for holding and computing - integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with ***pointers*** and ***arrays*** present these machine-level notions to the programmer in a reasonably implementation independent manner.

For most applications, one could simply use ***bool*** for logical values, ***char*** for characters, ***int*** for integer values, and ***double*** for floating-point values. The remaining fundamental types are variations for optimizations and special needs that are best ignored until such needs arise. They must be known, however, to read old C and C++ code.

4.1. Booleans

A Boolean, bool, can have one of the two values ***true*** or ***false***. A Boolean is used to express the results of logical operations. For example:

```
bool b1 = a == b; // = is assignment, == is Equality
   // ...
```

If a and b have the same value, b1 becomes true; otherwise, b1 becomes false. A common use of bool is as the type of the result of a function that tests some condition (a predicate). For example:

```
bool  b =  7;       //  bool(7) is  true,  so  b becomes true

int i = true;      // int(true) is 1, so i becomes 1
```

In arithmetic and logical expressions, bools are converted to ints; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to bool, a 0 is converted to false and a nonzero value is converted to true.

EX1: What is the screen output after executing the following C++ program?

```
#include <iostream>
using namespace std;
int main()
 {
   bool a = true;
   bool b = true;
   bool  x;
   bool  y;
   x=a+b;
   y=a-b;
   cout<<"The value of x is "<< x << endl;
   cout<<"The value of y is "<< y << endl;
 }
```

4.2 Character Types:

A variable of type char can hold a character of the implementation's character set. For example:

```
char ch = 'a';
```

Almost universally, a char has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Each character constant has an integer value. For example, the value of ‘b’ is 98 in the ASCII character set. Here is a small program that will tell you the integer value of any character you care to input:

EX2: What is the screen output after executing the following C++ program?

```
#include <iostream>
using namespace std;
int main()
{
      char c;
      cin>> c;
      cout<< “The  value  of” << c << “is” << int(c) << endl;
}
```

The notation ***int***(c) gives the integer value for a character c. The possibility of converting a char to an integer raises the question: is a char signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values 0 to 255 or as the values -127 to 127. Unfortunately, which choice is made for a plain char is implementation-defined. C++ provides two types for which the answer is definite; signed char, which can hold at least the values -127 to 127, and unsigned char, which can hold at least the values 0 to 255. Fortunately, the difference matters only for values outside the 0 to 127 range, and the most common characters are within that range.

4.3 Integer Types:

Like char, each integer type comes in three forms: “plain” int, signed int, and unsigned int. In addition, integers come in three sizes: short int, “plain” int, and long int. A long int can be referred to as plain long. Similarly, short is a synonym for short int, unsigned for unsigned int, and signed for signed int. The unsigned integer types are ideal for uses that treat storage as a bit array. Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by the implicit conversion rules. Unlike plain chars, plain ints are always signed. The signed int types are simply more explicit synonyms for their plain int counterparts.

```
int x;
long int y;
short int z;
```

4.4 Floating-Point Types:

The floating-point types represent floating-point numbers. Like integers, floating-point types come in three sizes: *float* (single-precision), *double* (double-precision), and *long double* (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don’t have that understanding, get advice, take the time to learn, or use *double* and hope for the best.

```
 float a;
float b;
double x;
long double y;
short double z;
```

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Size	Range
Char	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Bool	1 byte	true or false
Float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
Double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8 bytes	+/- 1.7e +/- 308 (~15 digits

5. Declaration of variables

In C++ (as in many other programming languages) all the variables that a program is going to use must be ***declared*** prior to use. Declaration of a variable serves two purposes:

It associates a ***type*** and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.

It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

A typical set of variable declarations that might appear at the beginning of a program could be as follows:

```
int i, j, count;
float sum, product;
char ch;
bool passed_exam;
```

which declares integer variables `i`, `j` and count, real variables `sum` and `product`, a character variable `ch`, and a boolean variable `pass_exam`.

A variable declaration has the form:

type identifier-list;

type specifies the type of the variables being declared. The ***identifier-list*** is a list of the identifiers of the variables being declared, separated by commas.
Variables may be initialised at the time of declaration by assigning a value to them as in the following example:

```
int i, j, count = 0;
float sum = 0.0, product;
char ch = '7';
bool passed_exam = false;
```

which assigns the value 0 to the integer variable `count` and the value 0.0 to the real variable `sum`. The character variable `ch` is initialised with the character `7`. `i`, `j`, and product have no initial value specified, so the program should make ***no*** assumption about their contents.

To see what variable declarations look like in action within a program, we are going to see the following C++ code.

EX3: What is the screen output after executing the following C++ program?

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
        declaring variables:
        int a, b;
        int result;
        a = 5;
        b = 2;
        a = a + 1;
        result = a - b;
        cout << result;
      return 0;
}
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

6. Scope of variables

A declaration introduces a name into a ***scope***; that is, a name can be used only in a specific part of the program text. For a name declared in a function (often called a ***local*** name), that scope extends from its point of declaration to the end of the block in which its declaration occurs. A block is a section of code delimited by a { } pair.

A name is called ***global*** if it is defined outside any function, class, or namespace. The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs. A declaration of a name in a block can hide a declaration in an enclosing block or a global name. That is, a name can be redefined to refer to a different entity within a block. After exit from the block, the name resumes its previous meaning. For example:

```
int x; // global x

void f()
 {
   int x; // local x hides global
   x = 1; // assign to local x
    {
      int x; // hides first local x
      x = 2; // assign to second local x
    }
   x = 3;    // assign to first local x

 }
int p = 6*x; // take address of global x
```

Hiding names is unavoidable when writing large programs. However, a human reader can easily fail to notice that a name has been hidden. Because such errors are relatively rare, they can be very difficult to find. Consequently, name hiding should be minimized. Using names such as i and x for global variables or for local variables in a large function is asking for trouble.

There is no way to use a hidden local name.

The scope of a name starts at its point of declaration; that is, after the complete declarator and before the initializer.

7. Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

EX4: What is the screen output after executing the following C++ program?

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
 {
   int a=5; // initial value = 5
   int b(2); // initial value = 2
   int result; // initial value undetermined
    a = a + 3;
    result = a - b;
    cout << result;
   return 0;
 }
```

8. Constants

Constants are expressions with a fixed value. C++ has two kinds of constants: literal, and symbolic. C++ has two kinds of constants: literal, and symbolic.

8.1 Literal constants:

Literal constants are literal numbers used to express particular values within the source code of a program. They are constants because you can't change their values. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
double a=5.8; // initial value = 5.8
int b(2); // initial value = 2
char ch = 'H'; // initial char value= H
 bool a = true;    // initial a value= 1
```

8.2 Symbolic constants:

Symbolic constants can be declared in two different ways: using the ***#define*** preprocessor directive, and through use of the ***const*** keyword.

You can define your own names for constants that you use very often without having to resort to memory consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define identifier value
```

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

EX5: What is the screen output after executing the following C++ program?

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n'

int main ()
  {
    double r=5.0;
    double circumf; // radius
    circumf = 2 * PI * r;
    cout << circumf;
    cout << NEWLINE;
   return 0;
  }
```

OUTPUT: 31.4159

In fact the only thing that the compiler preprocessor does when it encounters ***#define*** directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

There are two major problems with symbolic constants declared using #define. First, because they are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger. Second, #defined values always have global scope. This means value #defined in one piece of code may have a naming conflict with a value #defined with the same name in another piece of code.

A better way to do symbolic constants is through use of the ***const*** keyword. Const variables must be assigned a value when declared, and then that value cannot be changed.

```
const double pi = 3.14159
const char newline = ‘\n’
```

Here, *pi* and *newline* are two typed constants. Although a constant variable might seem like an oxymoron, they can be very useful in helping to document your code. Const variables act exactly like normal variables in every case except that they cannot be assigned to.

EX6: Write a C++ program to calculate the weekly pay?

weeklyPay = workDays * workHours * payRate;

```
#include <iostream>
using namespace std;
int main (void)
 {
      int workDays;
      float workHours, payRate, weeklyPay;
      workDays = 5;
      workHours = 7.5;
      payRate = 38.55;
      weeklyPay = workDays * workHours * payRate;
      cout << "Weekly Pay = ";
      cout << weeklyPay;
      cout << '\n';
  }
```

EX7: Write a C++ program to apply four different types of data and outputs it?

```
#include<iostream>
using namespace std;
int main( )

{
    bool pass=true;
    cout << "bool= "<<pass<<'\n';
    int num=3;
    cout << "Number= "<<num<<'\n';
    char ch='a';
    cout << "Character= "<<ch<<'\n';
    float fa=34.45;
    cout<<"Float number= "<<fa<<'\n';
}
```

EX8: Write a C++ program to input three different types of data and outputs it?

```
#include<iostream>
using namespace std;
int main( )

{
  int n; float f; char c;
  cout << "input integer number: ";
  cin>>n;
  cout<<n<<endl;
  cout << "input float number: ";
  cin>>f;
  cout<<f<<endl;
  cout << "input character: ";
  cin>>c ;
  cout<< c ;
  return 0;
}
```

EX9: Write a C++ program that reads the radius of a circle, then outputs its area?

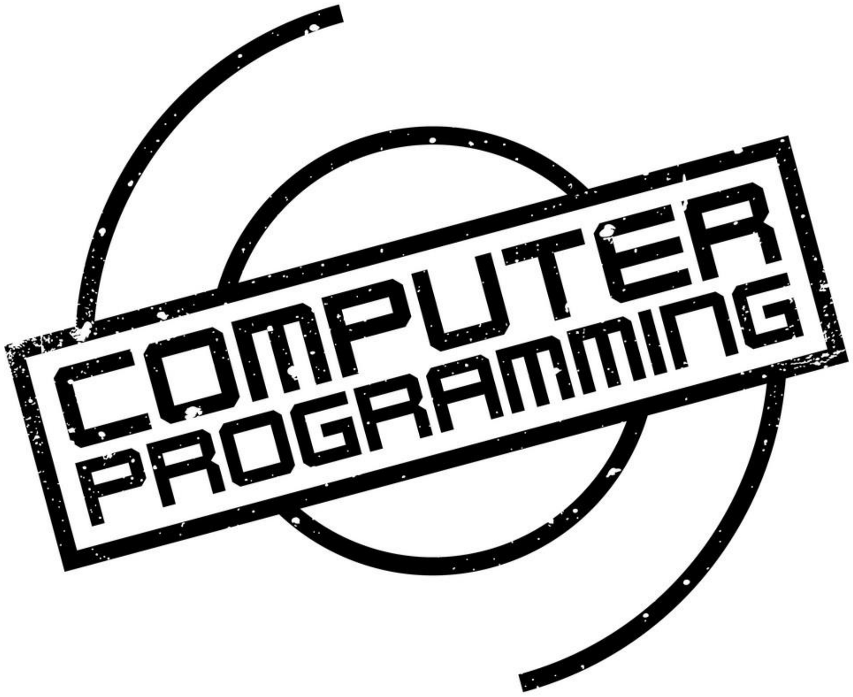
```
#include<iostream>
using namespace std;
int main( )
{
  float radu, Area_cir;
  const float pi = 3.14;
  cout << "enter the radius of circle: ";
  cin>>radu;
  cout<<endl;
  Area_cir = radu * radu * pi;
  cout << "the area of circle: " << Area_cir;
  cout <<endl;
  return 0;
}
```

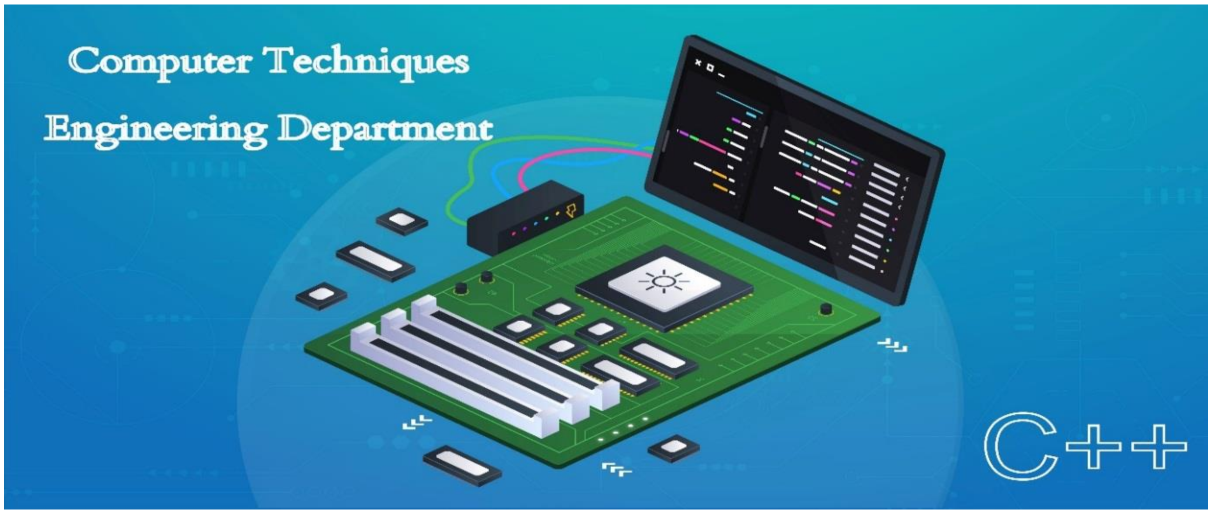
EX10: Write a C++ program that reads two integer values and computes all arithmetic operators?

```
#include<iostream>
using namespace std;
int main( )
{
  int a,b,sum,sub,mul;
  float div;
  cout << "enter any two numbers"<<endl;
  cin>>a>>b;
  sum=a+b;
  sub=a-b;
  mul=a*b;
  div=a/b;
  cout<<"a="<<a<<" b="<<b<<endl;
  cout<<"sum="<<sum<<endl;
  cout<<"sub="<<sub<<endl;
  cout<<"mul="<<mul<<endl;
  cout<<"div="<<div<<endl;
  return 0;
}
```

EX11: What is the screen output after executing the following C++ program?

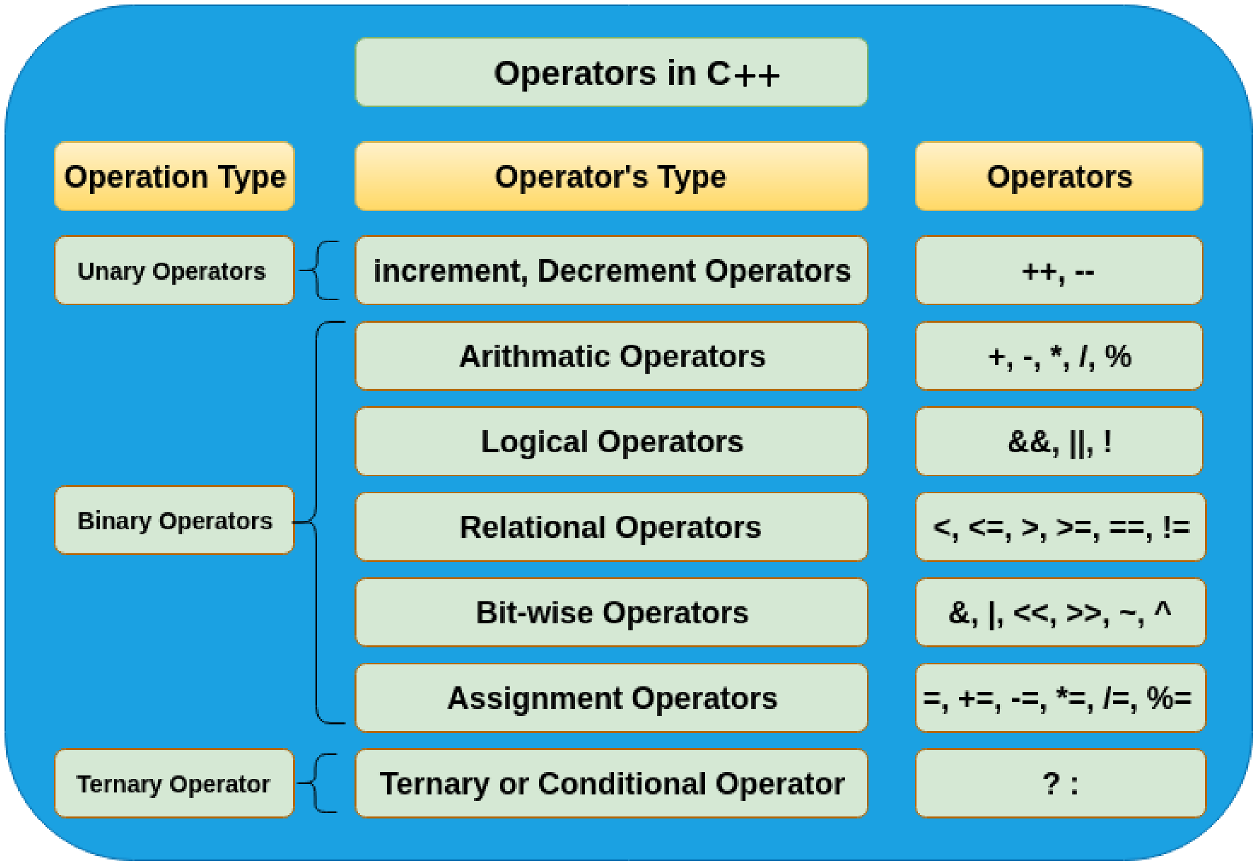
```
 #include<iostream>
 using namespace std;
  int main( )
   {
     int x,y;
     float z,r;
     x= 7 / 2;
     cout << "x=" << x <<endl; y=17.0/(-3);
     cout << "y="<< y <<endl; z=-17/3.0;
     cout << "z="<< z <<endl; r=int(-17.0/-3);
     cout << "r="<< r <<endl;
     //In arithmetic, the remainder is the integer "left over"
after dividing one integer by another to produce an integer
quotient
     int y1, y2;
     y1 = 8 % 5;
     y2 = -17 % 3;
     cout << "y1="<< y1 <<endl;
     cout << "y2="<< y2 <<endl;
   }
```


COMPUTER
PROGRAMMING


Computer Techniques
Engineering Department
C++

9. Operators

This section introduces the built-in C++ operators for composing expressions. C++ provides operators for composing arithmetic, relational, logical, bitwise, and conditional expressions. It also provides operators which produce useful side-effects, such as assignment, increment, and decrement. We will look at each category of operators in turn and also discuss the precedence rules which govern the order of operator evaluation in a multi-operator expression.

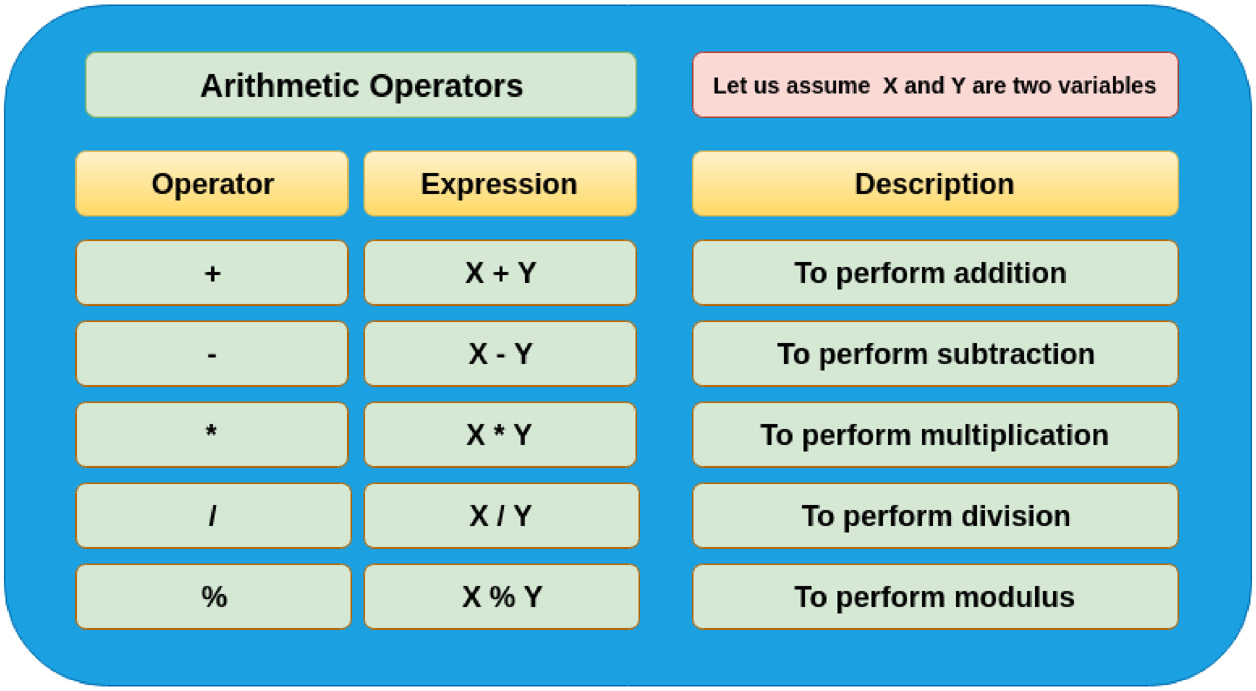


Operators in C++

Operation Type	Operator's Type	Operators
Unary Operators	increment, Decrement Operators	++, --
Binary Operators	Arithmatic Operators	+, -, *, /, %
	Logical Operators	&&, ||, !
	Relational Operators	<, <=, >, >=, ==, !=
	Bit-wise Operators	&, |, <<, >>, ~, ^
	Assignment Operators	=, +=, -=, *=, /=, %=
Ternary Operator	Ternary or Conditional Operator	? :

9.1 Arithmetic Operators

C++ provides five basic *arithmetic operators*. These are summarized in the table below.



Arithmetic Operators

Let us assume X and Y are two variables

Operator	Expression	Description
+	X + Y	To perform addition
-	X - Y	To perform subtraction
*	X * Y	To perform multiplication
/	X / Y	To perform division
%	X % Y	To perform modulus

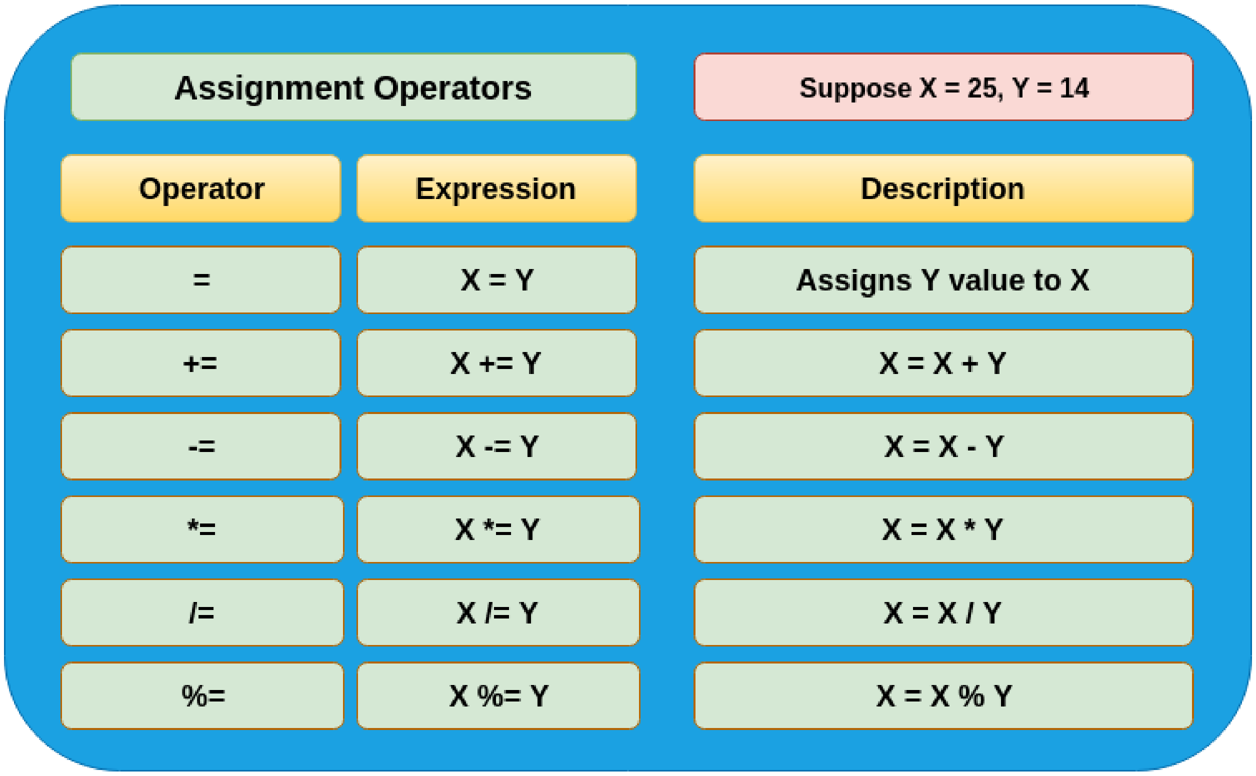
Except for modulo (%) all other arithmetic operators can accept a mix of integer and real operands. Generally, if both operands are integers then the result will be an integer. However, if one or both of the operands are reals then the result will be a real (or double to be exact).

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. 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.

9.2 Assignment Operator

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 left value) is anything that denotes a memory location in which a value may be stored. The kind of lvalue we have seen so far is a variable, pointers and references.

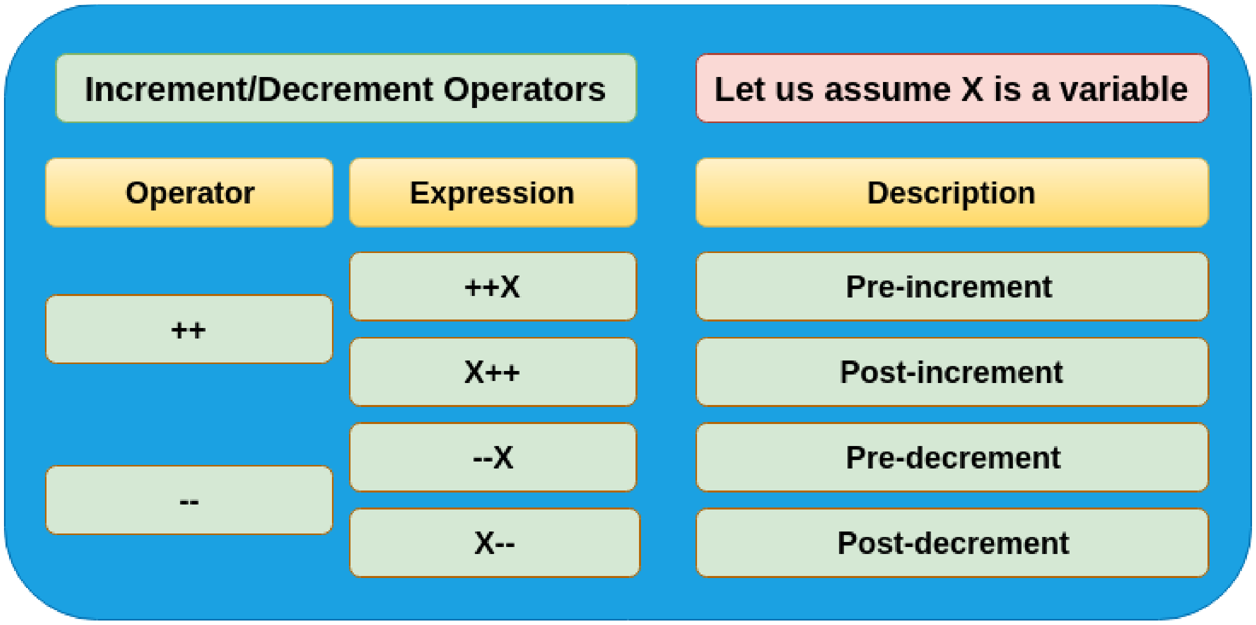
The assignment operator has a number of variants, obtained by combining it with the arithmetic and bitwise operators. These are summarized in the table below. The examples assume that ***n*** is an integer variable



Assignment Operators		Suppose X = 25, Y = 14
Operator	**Expression**	**Description**
=	X = Y	Assigns Y value to X
+=	X += Y	X = X + Y
-=	X -= Y	X = X - Y
*=	X *= Y	X = X * Y
/=	X /= Y	X = X / Y
%=	X %= Y	X = X % Y

9.3 Increment/Decrement Operators

The *auto increment (++)* and *auto decrement (--)* operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. These are summarized in the table below.

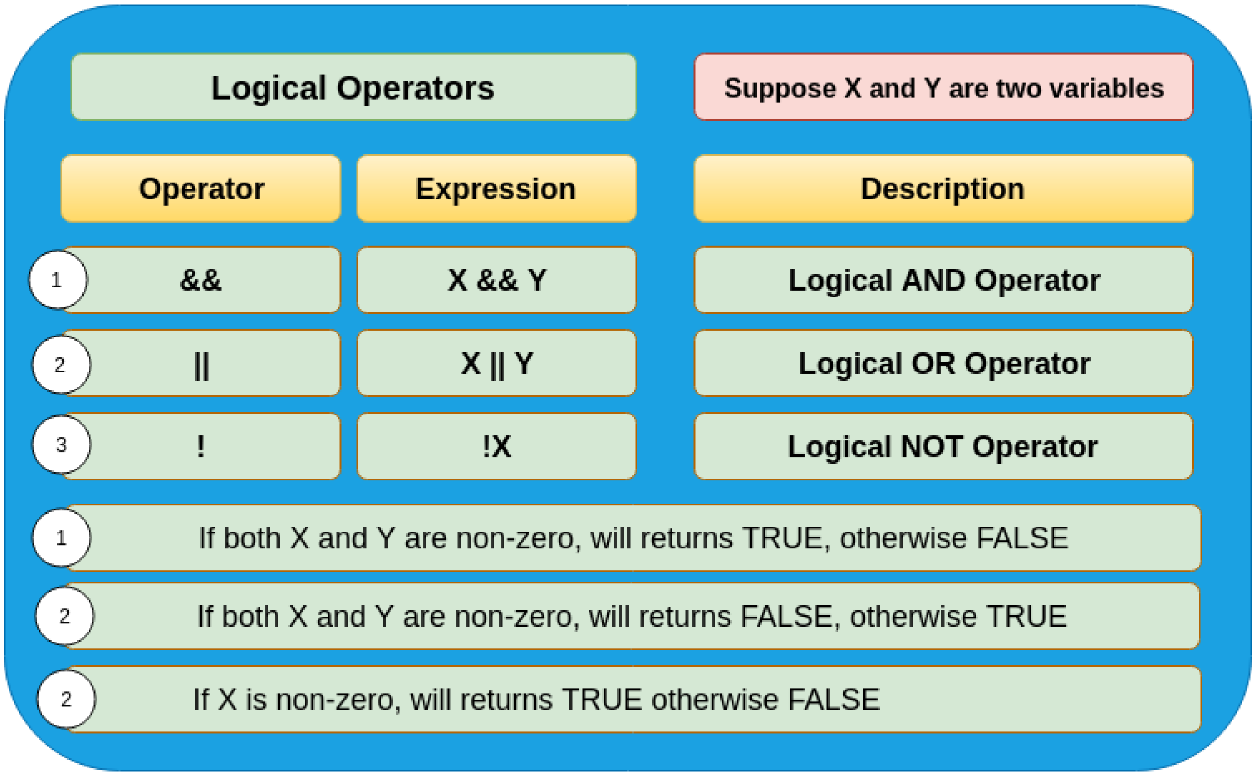


Increment/Decrement Operators		Let us assume X is a variable
Operator	**Expression**	**Description**
++	++X	Pre-increment
	X++	Post-increment
--	--X	Pre-decrement
	X--	Post-decrement

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator applied.

9.4 Logical Operators

C++ provides three *logical operators* for combining logical expression. These are summarized in the table below. Like the relational operators, logical operators evaluate to 1 or 0.



Logical Operators

Suppose X and Y are two variables

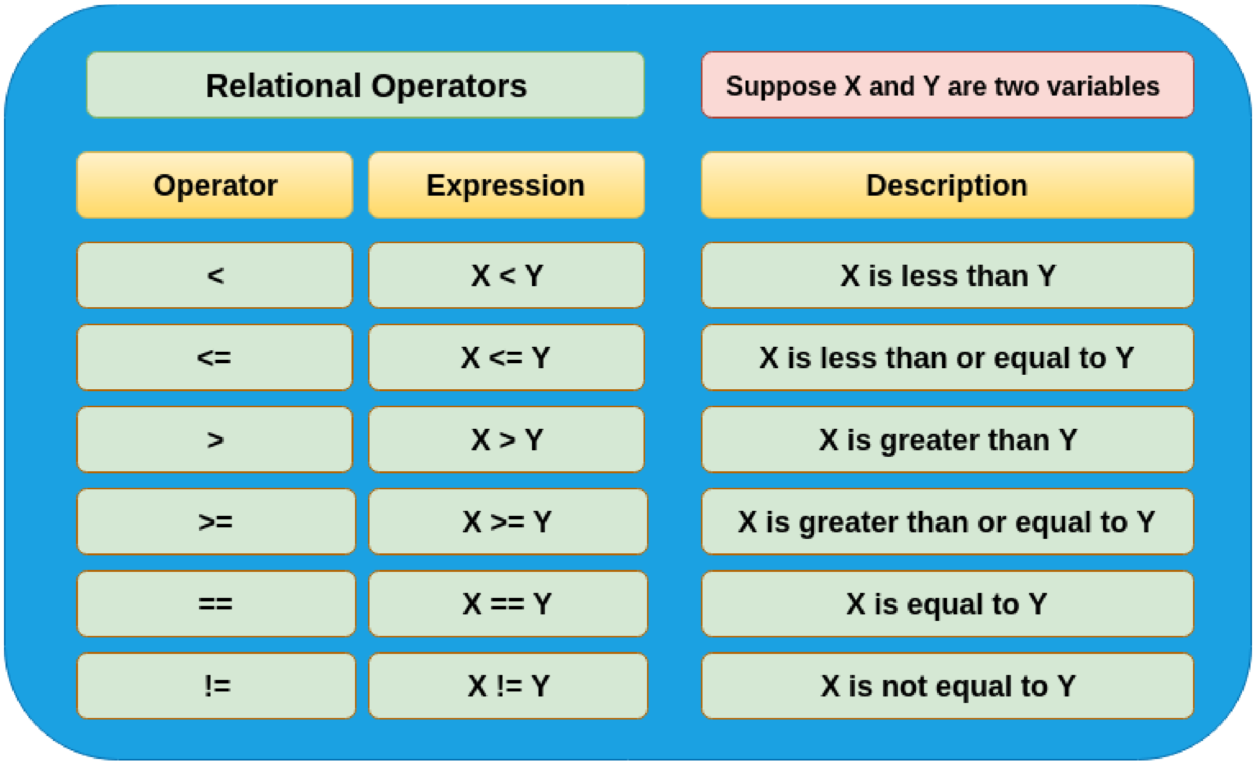
	Operator	Expression	Description
1	&&	X && Y	Logical AND Operator
2	||	X || Y	Logical OR Operator
3	!	!X	Logical NOT Operator

1	If both X and Y are non-zero, will returns TRUE, otherwise FALSE
2	If both X and Y are non-zero, will returns FALSE, otherwise TRUE
2	If X is non-zero, will returns TRUE otherwise FALSE

Logical ***negation*** is a unary operator, which negates the logical value of its single operand. If its operand is nonzero it produces 0, and if it is 0 it produces 1. Logical ***and*** produces 0 if one or both of its operands evaluate to 0. Otherwise, it produces 1. Logical ***or*** produces 0 if both of its operands evaluate to 0. Otherwise, it produces 1.

9.5 Relational Operators

C++ provides six *relational operators* for comparing numeric quantities. These are summarized in the table below. Relational operators evaluate to 1 (representing the true outcome) or 0 (representing the false outcome).



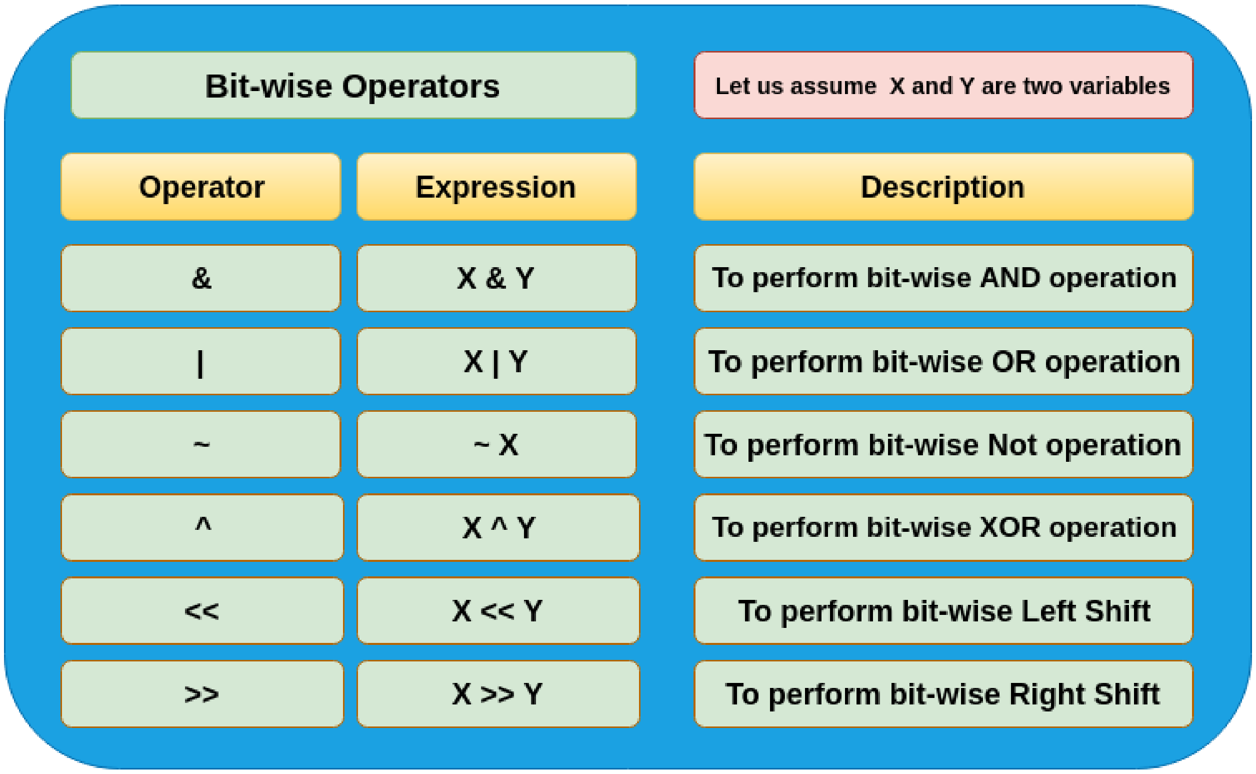
Relational Operators		Suppose X and Y are two variables
Operator	**Expression**	**Description**
<	X < Y	X is less than Y
<=	X <= Y	X is less than or equal to Y
>	X > Y	X is greater than Y
>=	X >= Y	X is greater than or equal to Y
==	X == Y	X is equal to Y
!=	X != Y	X is not equal to Y

Note that the <= and >= operators are only supported in the form shown. In particular, =< and => are both invalid and do not mean anything.

The relational operators should not be used for comparing strings, because this will result in the string addresses being compared, not the string contents. For example, the expression "HELLO" < "BYE" causes the address of "HELLO" to be compared to the address of "BYE". As these addresses are determined by the compiler (in a machine-dependent manner), the outcome may be 0 or may be 1, and is therefore undefined. C++ provides library functions (e.g., ***strcmp***) for the lexicographic comparison of string. These will be described later in the book.

9.6 Bitwise Operators

C++ provides six *bitwise operators* for manipulating the individual bits in an integer quantity. These are summarized in the table below.



Bit-wise Operators

Let us assume X and Y are two variables

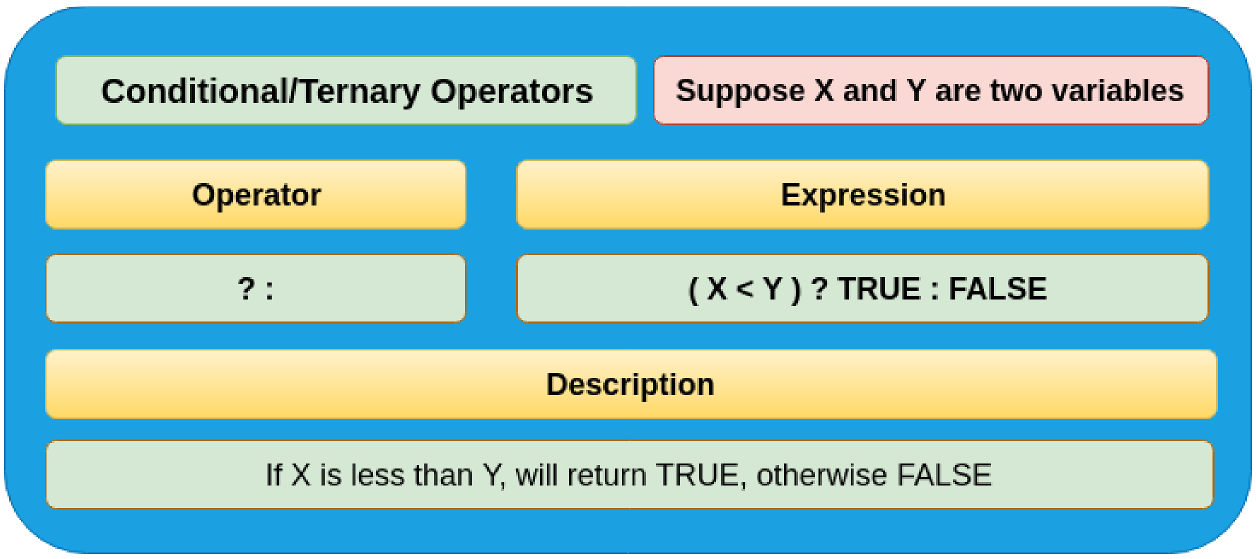
Operator	Expression	Description
&	X & Y	To perform bit-wise AND operation
|	X | Y	To perform bit-wise OR operation
~	~ X	To perform bit-wise Not operation
^	X ^ Y	To perform bit-wise XOR operation
<<	X << Y	To perform bit-wise Left Shift
>>	X >> Y	To perform bit-wise Right Shift

Bitwise operators expect their operands to be integer quantities and treat them as bit sequences. Bitwise **negation** is a unary operator which reverses the bits in its operands. Bitwise **and** compares the corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise. Bitwise **or** compares the corresponding bits of its operands and produces a 0 when both bits are 0, and 1otherwise. Bitwise **exclusive or** compares the corresponding bits of its operands and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise.

Bitwise **left shift** operator and bitwise **right shift** operator both take a bit sequence as their left operand and a positive integer quantity ‘n’ as their right operand. The former produces a bit sequence equal to the left operand but which has been shifted ‘n’ bit positions to the left. The latter produces a bit sequence equal to the left operand but which has been shifted ‘n’ bit positions to the right. Vacated bits at either end are set to 0.

9.7 Conditional/ Ternary Operator

The *conditional operator* takes three operands. It has the general form



Conditional/Ternary Operators	Suppose X and Y are two variables
Operator	**Expression**
? :	(X < Y) ? TRUE : FALSE
Description	
If X is less than Y, will return TRUE, otherwise FALSE	

e.g. If the condition is true, it returns first value i.e. TRUE case value, otherwise returns second value i.e. FALSE case value.

```
operand1 ? operand2 : operand3
```

First operand1 is evaluated, which is treated as a logical condition. If the result is nonzero then operand2 is evaluated and its value is the final result. Otherwise, operand3 is evaluated and its value is the final result. For example:

```
int m = 2, n = 1;
int min = (m < n ? m : n); // min receives 1
```

Note that of the second and the third operands of the conditional operator only one is evaluated.

EX1: What is the screen output after execute the following C++ program?

```
#include <iostream>
using namespace std;
int main( )
{
   int a, b=3;
   a = b;
   a+=2;
   cout << a<<"  "<<++b;

}
```

EX2: Write a C++ program to read four marks of the students, then find & print the average?

```
#include<iostream>
using namespace std;
int main( )
{
  double a,b,c,d,sum=0,av;
  cout<<"enter the 1st mark: ";
  cin>>a;
  cout<<"enter the 2nd mark: ";
  cin>>b;
  cout<<"enter the 3rd mark: ";
  cin>>c;
```

```
    cout<<"enter the 4th mark: ";
    cin>>d;
    sum = a+b+c+d;
    av  = sum/4;
    cout<<"the average is: "<<av<<endl;
    return 0;
  }
```

EX3: Write C++ program to compute the area of circle if the radius r=2.5?