
Programming Fundamentals

C++

Contents

<i>Chapter 1: Programming Logic and Techniques</i> -----	1
<i>Chapter 2: Introduction to C++</i> -----	15
<i>Chapter 3: C++ Concepts</i> -----	27
<i>Chapter 4: Manipulators & Control Structures</i> -----	64
<i>Chapter 5: Functions</i> -----	96
<i>Chapter 6: Derived Data Types (Arrays, Pointers, Functions)</i> -----	127
<i>Chapter 7: Strings in C++</i> -----	156

Chapter 1

CHAPTER 1.

PROGRAMMING LOGIC AND TECHNIQUES

CONTENTS

- 1.1 What is an Algorithm?
- 1.2. Characteristics of an Algorithm:
- 1.3. Algorithm Designing Tools
- 1.4. Designing a program
- 1.5 Program Testing:

1.1 What is an Algorithm?

Algorithm is an effective method to obtain step by step solution of problems. Knowledge of algorithms forms the foundation to study programming languages. Before starting with programming let us understand algorithms i.e. how to write algorithms, characteristics of algorithms, algorithm designing tools and conversion of algorithms to programs.

An algorithm is defined as the finite set of steps followed in order to solve the given problem.

For Example:

Algorithm To find the average score of a student, for the three test marks .

Step 1: Read the three test marks; say S1, S2, S3

Step 2: $SUM = S1 + S2 + S3$

Step 3: $Average = SUM / 3$

Step 4: Display Average

Step 5: Stop

Algorithm to find the largest of three numbers

- Step 1:** Read the three numbers say A, B, C.
- Step 2:** If $(A > B)$ and $(B > C)$ then A is the largest number, go to step 4.
- Step 3:** If $(B > A)$ and $(A > C)$ then B is the largest else C is the largest number.
- Step 4:** Stop.

One can convert these algorithms to programming statements by learning any programming language.

1.2. Characteristics of an Algorithm:**(1) Input**

An algorithm must be provided with any number of input/data values. (in some cases no external input is needed)

(2) Output

An Algorithm must produce at least one output.

(3) Definiteness

Each step of the algorithm must be clear and distinct which ensures that the statement must be unambiguous.

For Example let us go through the following algorithm to divide two numbers

Algorithm:

- Step 1:** Read the two numbers say a, b
- Step 2:** $c = a/b$
- Step 3:** Print c
- Step 4:** Stop

Step 2 of this algorithm is not clear, as there will no output(infinite output) if $b=0$. Since the system doesn't

provide such an answer, care must be taken while writing an algorithm.

The above algorithm can be rectified as follows

Step 1: Read the two numbers say a, b

Step 2: If $(b=0)$ then

Print "denominator value is 0"

And go to step 5

Step 3: $c = a/b$

Step 4: Print c

Step 5: Stop

(4) Finiteness

The algorithm must terminate after a finite number of steps. Let us illustrate this point with the help of an example.

Algorithm:

Step 1: Let $a = 9$

Step 2: If $(a > 10)$ then go to step 5

Step 3: $X = Y * Z$

Step 4: Print X and go to step 2

Step 5: Stop

Here we notice that in the algorithm, nowhere the value of "a" is changed, which controls the flow of the algorithm and hence algorithm never terminates. Such statements must be avoided. The finiteness property assures the unambiguity in the flow.

(5) Effectiveness

It must be possible in practice, to carry out each step manually (using paper and pencil). The statements must be feasible. Thus algorithm even though is definite, should also be practical.

If one takes care of above mentioned characteristics while writing algorithm, then we can be sure of the results, for any kind of inputs.

1.3. Algorithm Designing Tools




Algorithm must be designed in such a way that it follows the pure top-down approach. This will ensure the straight line execution of the algorithm. An algorithm can be expressed or designed in many ways. One can make use of any language to specify steps involved in solving a particular problem but simple and precise language could be adopted. Two famous ways of writing algorithm are making use of flowcharts and pseudocode (Structured English).

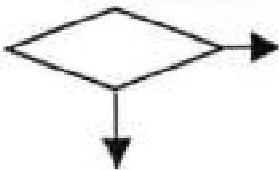
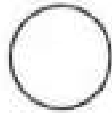

Let us understand designing of algorithms with the help flowchart and pseudocode language.

1.3.1. Flowchart :

Flowchart is a diagrammatic way of representing, the steps to be followed for solving the given problem.

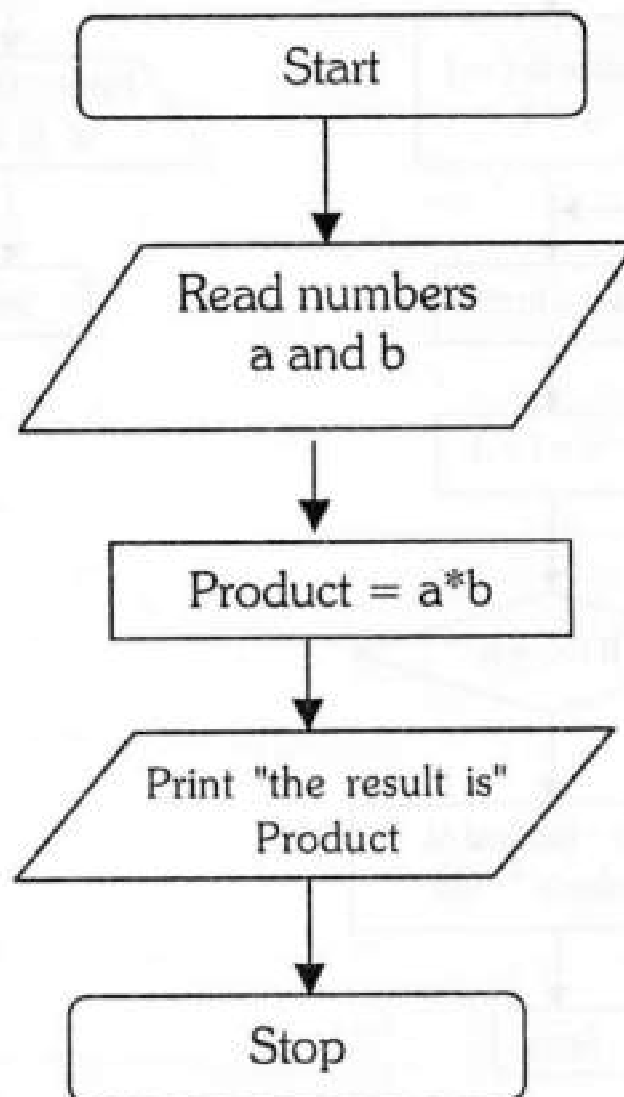
Flowchart provides visualization of the steps involved. Since it is in the form of a diagram one can understand the flow very easily. Here are some diagrammatic symbols to design a flowchart.

Symbol	Purpose
	START/STOP
	ASSIGNMENT STATEMENTS, EXPRESSIONS etc.
	INPUT/OUTPUT

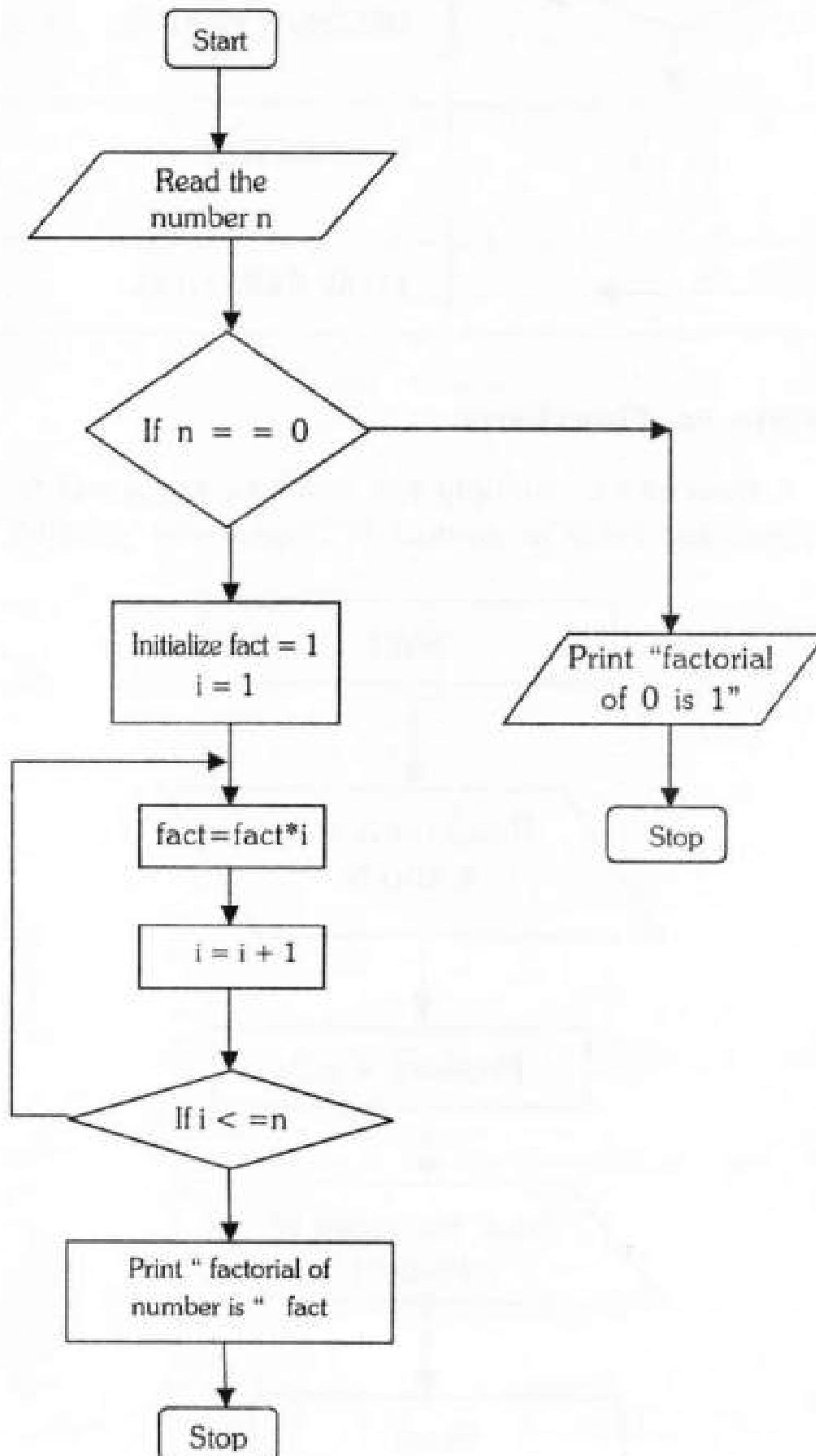
	DECISION MAKING
	CONNECTOR
	FLOW INDICATOR

Example on Flowcharts:

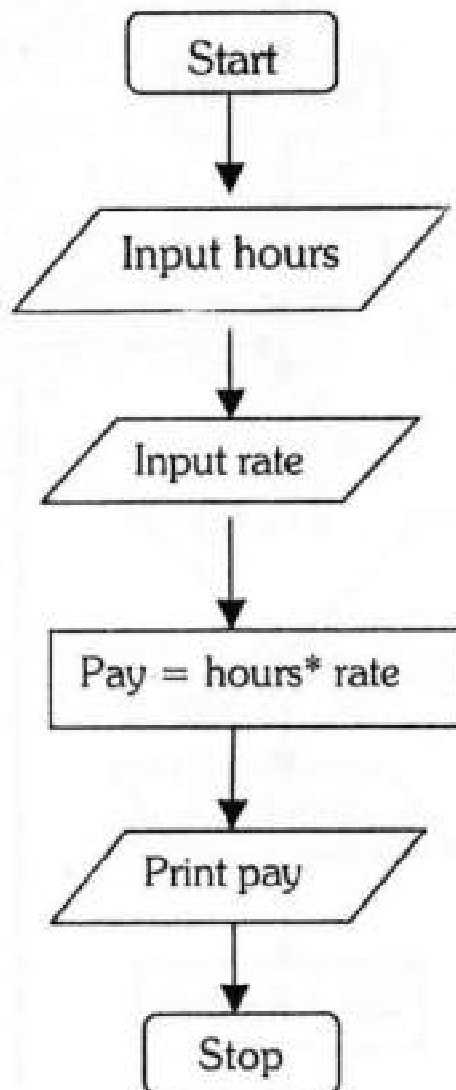
- A flowchart to multiply two numbers say a and b and store the result in product & Display the “product”



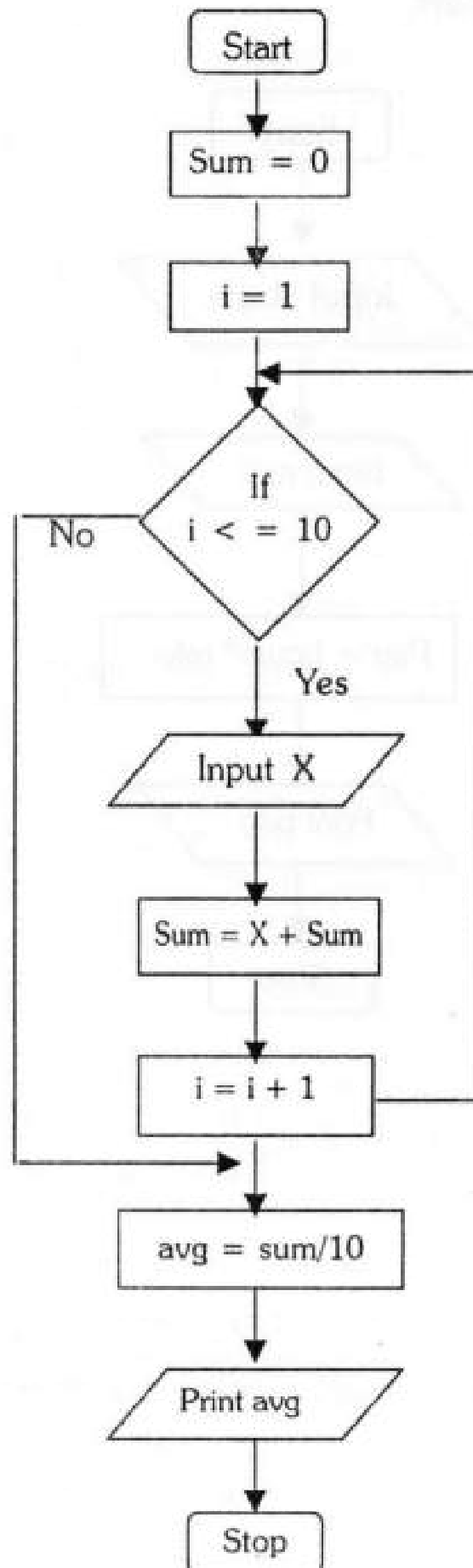
- Flow chart to find factorial of a number 'n'.



- Flowchart to calculate the pay by using the formula $\text{pay} = \text{rate} * \text{hours}$.



- Flowchart to find the average of 10 numbers .



Another way of designing algorithms is by using pseudocode.

1.3.2. Pseudocode

Pseudocode is an artificial and informal language that helps the programmers to develop algorithms. Pseudocode designs the algorithm in text based format. It allows programmers to focus on the logic of the algorithm without being distracted by details of the language syntax. It narrates steps of the algorithm more precisely.

Following are some keywords used to indicate input, output, and other operations

Input - READ, GET

Output - PRINT, DISPLAY

Compute - CALCULATE, DETERMINE

Initialize - SET, INT

Add one - INCREMENT

Conditions & Loops

IF-THEN ELSE

Repetitive execution

WHILE

CASE

REPEAT UNTIL

FOR

Let us go through some pseudocodes

Pseudocode to obtain sum of two numbers

BEGIN

INPUT X,Y

DETERMINE $SUM=X+Y$

```
PRINT SUM
END
```

Pseudocode to obtain average of three numbers

```
BEGIN
DISPLAY "input 3 nos"
INPUT X,Y,Z
DETERMINE SUM=X+Y+Z
DETERMINE AVG=SUM/3
PRINT "average is " AVG
END
```

1.4. Designing a program

A program is a set of instructions that are grouped together to accomplish a task or tasks. The instructions consist of task like reading and writing memory, arithmetic operations, and comparisons.

Aim of a particular Program is to obtain solution to a given problem.

We can design a program by going through the following first four major steps:

- a. ANALYSE THE PROBLEM**
 - b. DESIGN A SOLUTION/PROGRAM**
 - c. CODE/ENTER THE PROGRAM**
 - d. TEST THE PROGRAM**
 - e. EVALUATE THE SOLUTION**
- a. ANALYSE THE PROBLEM**

When we analyze a problem, we think about the requirements of the program and how the program can be solved.

b. DESIGN A SOLUTION/PROGRAM

This is the stage where we decide how our program will work to meet the decisions made during analysis.

Program design does not require the use of a computer. We can design program using pencil and paper.

This is the stage where algorithms are designed.

c. CODE/ENTER THE PROGRAM

Here we enter the program into the machine by making use of suitable programming Language.

d. TEST THE PROGRAM

This part deals with testing of programs for various inputs and making necessary changes if required.

Testing cannot show that a program produces the correct output for all possible inputs, because there are typically an infinite number of possible inputs. But testing can reveal syntax errors, run-time problems and logical mistakes.

e. EVALUATE THE SOLUTION

Thus, finally program can be implemented to obtain desired results.

Here are some more points to be considered while designing a program.

- Use of procedures.
- Choice of variable names
- Documentation of programs
- Debugging programs
- Testing.

Use of procedures:

Procedure is a part of code used to carry out independent task. Separate procedures could be written to carry out different task and then can be combined and linked with the main procedure. This will help in making the algorithm and eventually programs readable and modular (divided into modules).

Choice of Variables:

We can make program more meaningful and easier to understand by choosing appropriate variable and constant names. For example, if wish to store age of two different people we can define variables age1, age2 to store their ages. The main advantage of choosing correct variables is that the program becomes self explanatory.

Documentation of Program:

Brief and accurate comments can be included at the beginning of each procedure/ function. Program (Algorithms) should be documented, so that it can be used easily by other people unfamiliar with the working and input requirements of the program. Thus documentation will specify what response it requires from the user during execution.

Debugging the Program:

It is expected that one should carry out number of tests during implementation of algorithm, to ensure that program is behaving correctly according to its specifications. The program may have some logical errors, which may not be detected during compilation. To detect such type of errors we may print computed values at various steps involved in algorithm. We should always manually execute the program by hand before ever attempting to execute it on machine.

1.5 Program Testing:

The program should be tested against many possible inputs. Some of the things we might check are whether the program solves the smallest possible problem, whether it may not be possible or necessary to write programs that handle all input conditions, all the time. Whenever possible, program should be accompanied by input and output sections.

Here are some desirable characteristics of a program

- **Integrity:** Refers to the accuracy of program.
- **Clarity:** Refers to the overall readability of a program, with emphasis on its underlying logic.
- **Simplicity:** The clarity and accuracy of a program are usually enhanced by keeping the things as simple as possible, consistent with the overall program objectives.
- **Efficiency:** It is concerned with execution speed and efficient memory utilization.
- **Modularity:** Many program can be decomposed into a independent procedures or modules.
- **Generality:** Program must be as general as possible. (viz. rather than keeping fixed values for variables, it is better to take it as an input during execution).

REVIEW QUESTIONS

1. Define an algorithm and state its characteristics.
2. Write a note on algorithm designing tools.
3. Explain the use of Flowcharts in designing of algorithms and symbols used in flowcharts.
4. What do you mean by a Pseudocode?
5. Write a note on designing of a program.
6. Explain the use of following while designing a program

- a. Use of procedures.
 - b. Choice of variables
 - c. Documentation of programs
 - d. Debugging programs
 - e. Testing.
7. What are the desirable characteristics of a program?
8. Write algorithms & Draw flowchart for the following.
- To obtain hypotenuse of the right angle triangle if remaining two sides are provided.
 - To obtain simple interest if principal (P), rate of interest (R) and no. of years (N) is provided. (Simple interest = $(P \cdot R \cdot N) / 100$)
 - To obtain compound interest if principal (P), rate of interest (R) and no. of years (N) is provided. (Compound interest Amount after n years = $P(1 + r)^n$ whre $r = R/100$)
 - To convert temperature in centigrade to Fahrenheit and vice versa.
 - To obtain greatest of three numbers.
 - To obtain roots of a quadratic equation.
 - To swap values of two variables.

□ □ □

Chapter 2

CHAPTER 2

INTRODUCTION TO C++

CONTENTS

- 2.1. Introduction to programming
- 2.2. THE ORIGIN OF C++
- 2.3. Structure of a simple C++ program
- 2.4. Compiling and running C++ programs
- 2.5. Pitfall and Tips to Programming
- 2.1. Introduction to programming

“Programming” is the process of writing instructions in any computer programming language, to get the task done by a computer.

C++ is a very common programming language, and a good language to start learning basic concepts with, but you need to figure out what you find most useful. To do that, you will have to try different programming languages over the time and see which ones fit best for you; which language structure makes sense, which one seems to be able to accomplish the goals you want etc.

Writing the program is the process of breaking out your instructions step by step and instructing the compiler or interpreter to do those things in the proper programming language.

Your first step is to figure out exactly what you want your program to do, step by step. It is helpful to write this out on paper (Writing an algorithm). Once you gain more experience you will start to see the value in doing this.

Once you have your steps figured out, you will want to write your program in the language you have chosen. Whatever programming language you choose, it will have specific word and styles to do different things. Much like we use words and punctuation every day, so do programs.

2.2. THE ORIGIN OF C++

C++ an object oriented programming language & was developed by Bjarne Stroustrup starting in 1979 at AT & T Bell Laboratories as an enhancement to the C programming language and originally named C with Classes. It was renamed C++ in 1983, where ++ is an increment operator in C & C++.

The C programming language was developed at AT&T for the purpose of writing the UNIX operating system. C was developed with the primary goal of operating efficiency. Bjarne Stroustrup developed C++ in order to add object oriented constructs to the C language. C++ can also be considered as a combination of C along with object oriented features of Simula67(General object oriented language).

C++ is also a traditional procedural language with some additional constructs. A well written C++ program will reflect elements of both object oriented programming style and classic procedural programming. C++ is actually an extensible language since we can define new types in such a way that they act just like the predefined types which are part of the standard language. C++ is designed for large scale software development.

C++ is regarded as a "middle-level" language, as it comprises combination of both high-level and low-level language features.

Note:- Programming languages can be classified into the following types

1. **Machine language** - Machine languages are the only languages understood by computers. While easily understood by computers, machine languages are almost impossible for humans to use because they consist entirely of numbers.
2. **Assembly language** - Assembly languages have the same structure and set of commands as machine

languages, but they enable a programmer to use names instead of numbers.

3. **High level Language** - A programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of a computer. Such languages are considered high-level because they are closer to human languages.

Machine language or an assembly language both are Low-level languages since they are closer to the hardware.

Some of the applications of C++ includes systems software, application software, device drivers, embedded software, high-performance server and client applications and entertainment software.

C++ is also used for hardware design.

The C++ language began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates, and exception handling among other features.

2.3. Structure of a simple C++ program

```
// simple program in C++
#include <iostream.h>
int main ()
{
cout << "Welcome to NIRMALA COLLEGE";
return 0;
}
```

Output: Welcome to NIRMALA COLLEGE

// simple program in C++

This is a comment line. Lines beginning with two slash signs (//) are used to comment on the program or a part of the program. In the above program the line is a brief description of our program .

#include <iostream.h>

Lines beginning with a hash sign (#) are directives for the preprocessor. In this case the directive `#include<iostream.h>` tells the preprocessor to include the header file `iostream` which is a standard file. This specific file (`iostream`) includes the declarations of the basic standard input-output library in C++, and is included because functions(`cout`) from this file is used in the program.

int main ()

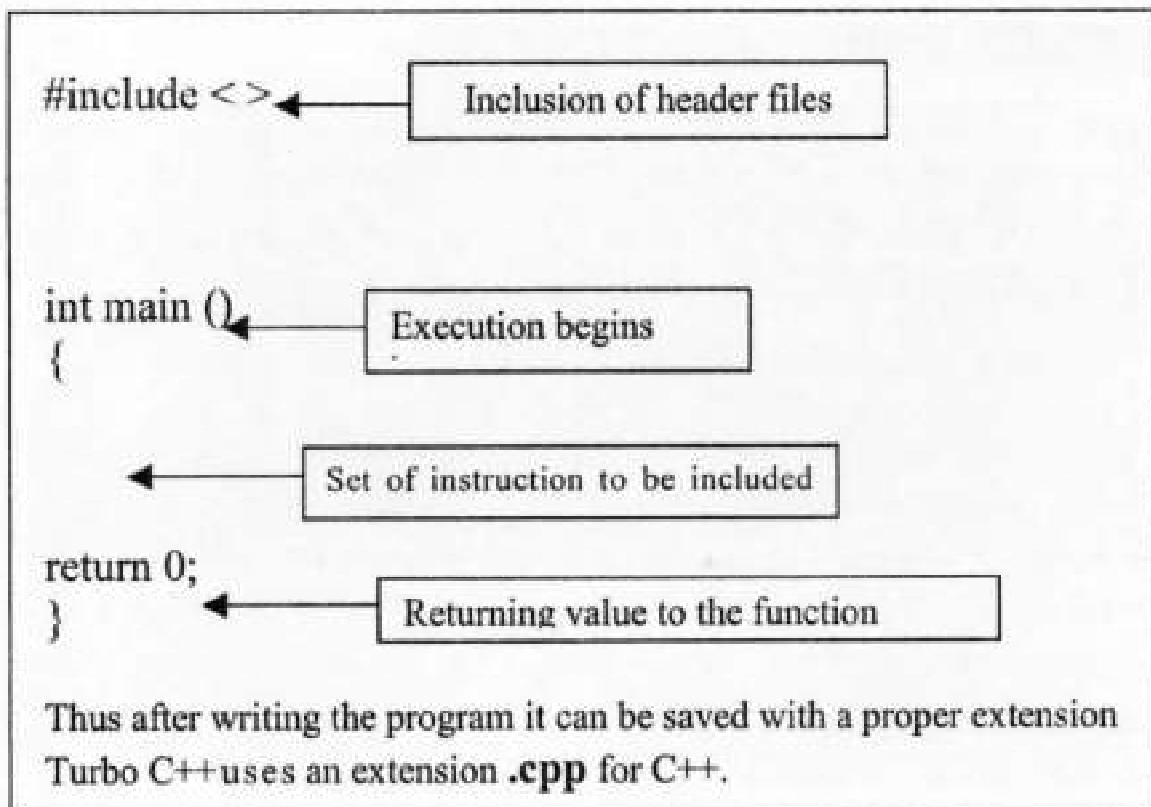
This line corresponds to the beginning of the definition of the main function. The `main()` function is the point where programs execution begins. Every C++ program must have a `main()`.

cout << "Welcome to NIRMALA COLLEGE";

`cout` represents the standard output stream in C++ and the meaning of the entire statement is to print (Welcome to NIRMALA COLLEGE) into the standard output stream (which usually is the screen).

return 0;

The return statement causes the main function to finish. This is the most usual way to end a C++ console program. Every function in C++ must return a value, thus `return 0;` returns 0 to the integer function `main()`.



2.4. Compiling and running C++ programs

Let us understand compiling and debugging with the help of Turbo C++ & Borland C++.

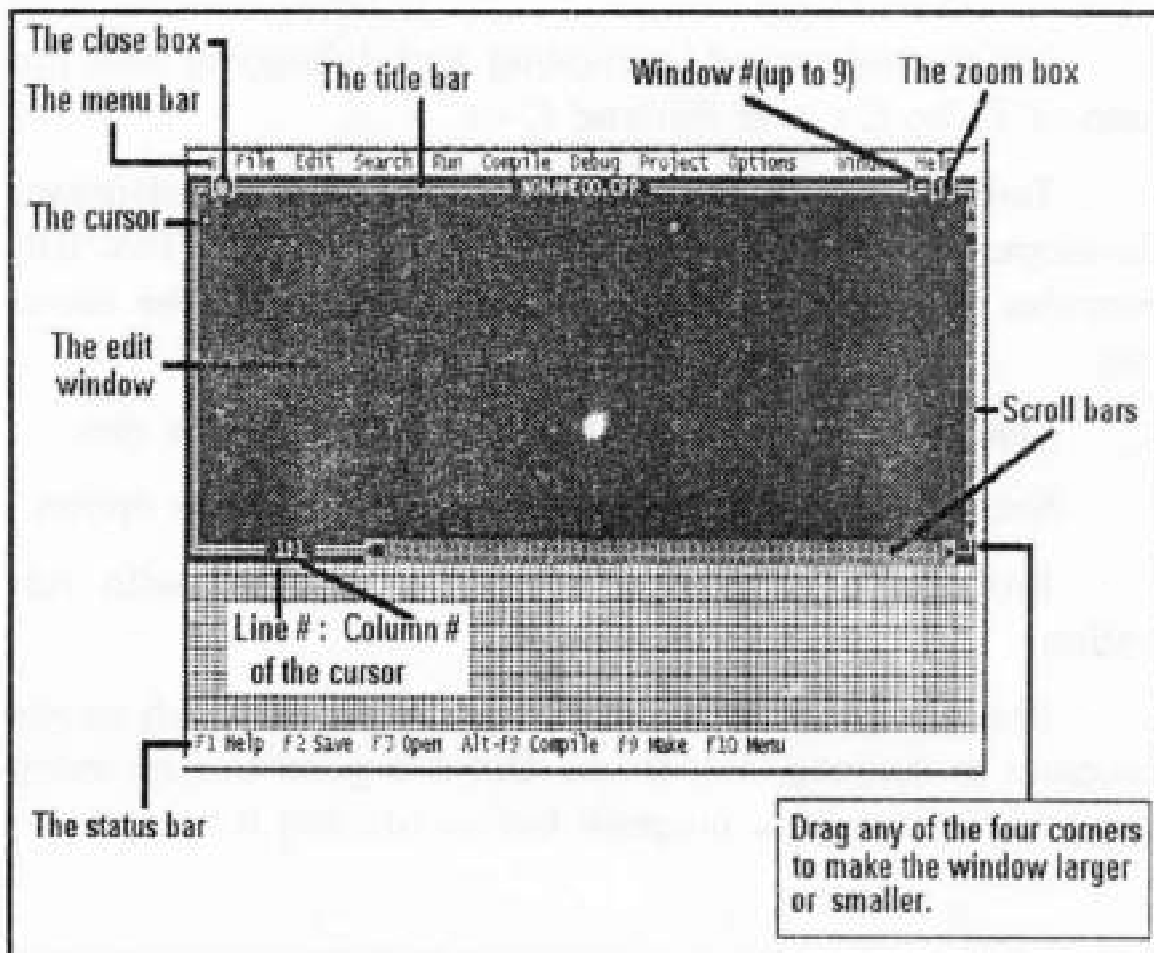
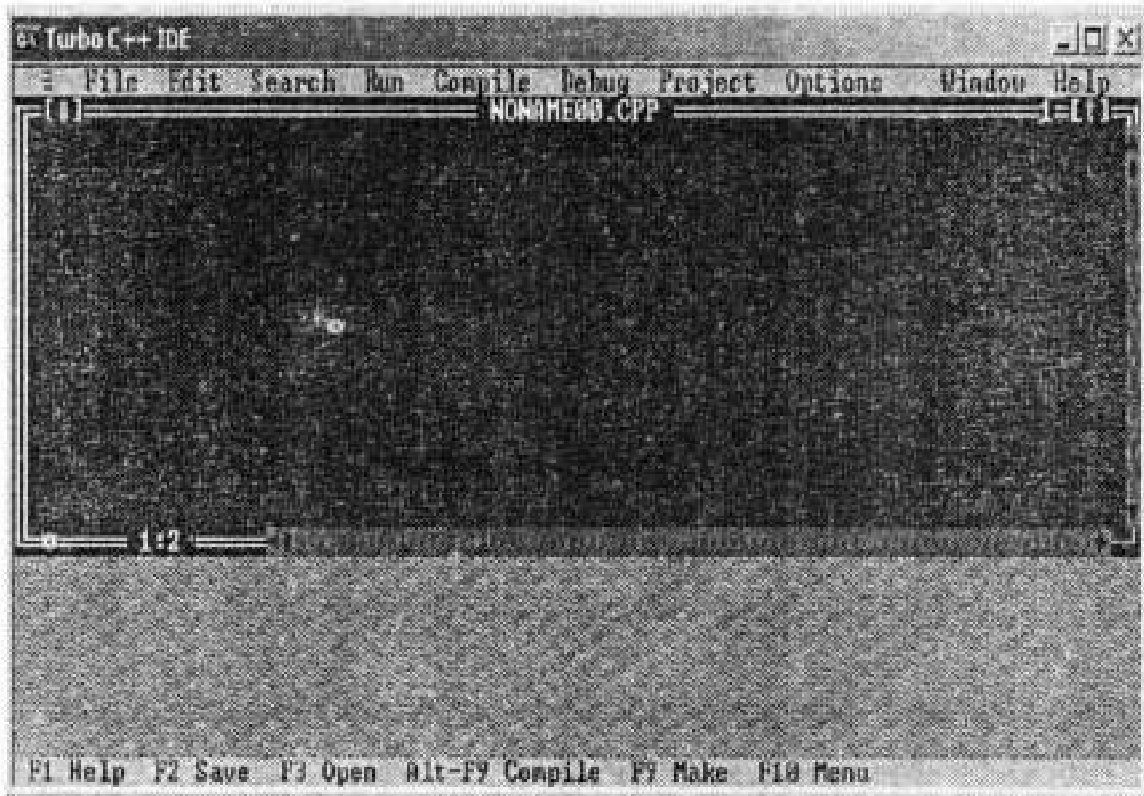
Turbo C++ & Borland C++ provides an integrated development environment (IDE) under MS DOS. This IDE provides us with an editor and several menus on the menu bar.

File menu helps in creating and saving source files.

Source file can be compiled by using compile option.

Program once compiled can be executed with run option.

(It is also possible to directly use run option which causes program to compile, link & run. But it is good programming practice to compile a program before running it.)



Unix AT&T C++ & Visual C++ are the other systems which can be used to compile and run C++ programs.

2.5. Pitfall and Tips to Programming

Here are some pitfalls and basic things to be taken care of in any programming language. Some of the tips are specifically with respect to concepts covered in remaining chapters and could be understood well after going through those concepts .

1. Plan and be organised :

When planning a program it is always better to plan ahead rather than jump straight into it. It is a good idea to write down the functions of the program in the order you need to code them, in little 'blocks'. Even "draw" it if it helps. When actually coding it, use comments. Doing these little things could save your valuable programming time and helps make the code look a little more explanatory and professional.

2. Write it out on paper

Think before you code and write the entire idea or the logic of solving a particular problem on paper; doing so would prevent lots of logical errors. This technique also works when you are not able to figure out why the logic is not working.

3. Indent your Program

Always indent your program for readability. It is a good practice and useful to debug your program. Good indentation makes it easy to see that your closing braces are correctly aligned and help you scan through your code quickly by giving visual clues about where the program flow changes.

4. Always use Comments

Make comments in your code often. Not only does it help people who are trying to help you understand it more, when you come back later, you'll be able to pick up where you left off and edit much faster.

For example:

```
int age1; /*holds age of first employee*/
```

Use either // or /* depending on your compiler and language

5. Always write if statements with braces.

By putting braces around every block of code you write, you ensure that future edits won't introduce bizarre bugs.

If you have a one-line if statement:

```
if(<condition>)
```

```
    execute();
```

you should still surround

```
execute(); with braces:
```

```
if(<condition>)
```

```
{
```

```
    execute();
```

```
}
```

Now, if you go back and add a second instruction

```
if(<condition>)
```

```
{
```

```
    execute();
```

```
    execute2();
```

```
}
```

You don't have to worry about putting in the braces, and you know that you won't forget to put them in.

6. Restrict goto

Some very simple programming languages allow you to control the flow of your program with the goto keyword. You simply place a label somewhere in your program, and then

can jump to that point in your code anytime you like by following goto with that keyword. Like so:

```
goto myLabel  
  
...  
  
myLabel:  
  
/* code */
```

In these simple and limited programming languages, goto may be used quite commonly, and this feature is included in C/C++ too. **It is however, considered as extremely bad programming practice.**

The reason for this is simply that there are better alternatives. C and C++ provide more advanced control structures like various types of functions and loops, which not only make it easy to code a certain behavior, but safer. goto is still sometimes used for customized control, like breaking out of heavily nested loops.

If your program does indeed require the use of goto, you should probably be redesigning the whole program to fix it properly instead of using a quick fix like goto.

If in doubt, don't use goto.

7. Use Appropriate Variables

While programming, we have to take care about the variables. We need to give appropriate name for the variables to avoid confusion!

8. Take advantage of array indices

When operating on two arrays with the same index and operation, the switch statement can usually be avoided.

Consider the general switch statement that assigns and tallies all the occurrences of array[i] to count[i]:

```
switch (array[i]) {  
    case 1: count[1]++;  
    break;  
    case 2: count[2]++;  
    break;  
    case 3: count[3]++;  
    break;  
    /*...  
    case n: count[n]++;  
    break;  
    */  
}
```

This can be shortened to the single statement:

```
count[array[i]]++;
```

9. Beware of loop invariants

Be aware of loop invariants and how they can affect code efficiency. An example:

If you have code within a loop that uses non-varying variables:

```
for(i = 0; i < 100; i++)  
{  
    total = i + x + y;  
    printf("%d\n", total);  
}
```

As you can see, *x* and *y* are not variables within the loop and so it is much more efficient to code the loop as follows:

```
total = x + y;
for (i = 0; i < 100; i++)
{
total += i;
}
```

This is because the loop does one less addition on each loop cycle.

10. Clean up after cin

It's a good idea to follow `cin` with `cin.get()` or `cin.ignore()` because `cin` can leave a terminating character in the stream, which could cause small problems with your code. For example:

```
#include<iostream>

int main()
{
    int age;
    std::cout<<"Enter your age: ";
    std::cin>>age;
    std::cout<<"You entered " <<age<<std::endl;
    std::cin.get();
    return 0;
}
```

That code exits right after printing the age. The program 'skips' right over where it should 'pause' for the user to press Enter. This is bad because the user never gets to see what is being written to the screen after they enter the age. The code, however, works as intended:

11. Code Optimization Tips

- If you've got a lot of if / else statements, try to put the one most likely to be met first.
- Use `a++` and `a--` instead of `a+=1` and `a-=1`

12. While Doing calculations

- Remember when doing calculations in C++, that multiplication and division takes priority above addition and subtraction.
- Adding brackets according to these rules, makes it to understand, and is a neater way to code.

Remember that when you use the `cout` statement, do not use quotes around an arithmetic expressions, it will simply display the whole line.

13. Reuse the code

14. Avoid usage of temporary variables.

15. Avoid memory wastage.

REVIEW QUESTIONS

1. What is the difference between C & C++ ?
2. "C++ is a middle level language", Explain ?
3. Write a note on types of programming languages.
4. Explain the parts of a simple c++ program.
5. What is the use of `#include<iostream.h>`.
6. Write a note on pitfalls and tips to programming.



Chapter 3

CHAPTER 3.

C++ CONCEPTS

CONTENTS

- 3.1. Variables
- 3.2. Identifiers
- 3.3. Keywords
- 3.4. CONSTANTS
- 3.5. Data types in C++
- 3.6. Operators in C++
- 3.7. Operators precedence
- 3.8. Declaration of variables
- 3.9. Initialization of variables
- 3.10. Type casting
- 3.11. Preprocessor directives
- 3.12. Defined constants or symbolic constants (#define)
- 3.13. Declared constants (const)
- 3.14. Namespaces
- 3.15. Reference Variable
- 3.16. Input and output statements in C++
- 3.17. Escape Sequences
- 3.18. Indenting programs
- 3.19. Comments in C++
- 3.20. Local & Global variables

3.1. Variables

Variable can be defined as a portion of memory used to store a determined value. Variable names are names given to locations in the memory. These locations can contain integer, real or character constants. We do a number of calculations

in computer and the computed values are stored in some memory spaces.

Since the value stored in each location may change, the name given to these locations are called as 'variable names'.

Each variable needs an identifier (variable name) that distinguishes it from the others.

3.2. Identifiers

A valid identifier is a sequence of one or more letters, digits or underscore character (_).

Here are some rules to define an identifier,

1. Neither spaces nor punctuation marks or symbols are allowed while defining an identifier.
2. Only letters, digits and single underscore characters are valid.
3. Identifiers should not begin with a digit.
4. Keywords (for eg. If , else, for etc.) are reserved and cannot be used as identifiers.
5. Uppercase and lowercase letters are distinct.
6. C++ does not set a maximum length for an identifier but some compilers treat only first 'n' characters significant.
7. Variable name can be arbitrarily long but most implementations recognizes 31 characters (ANSI standards also recognizes 31 characters).

Here are some valid and invalid variable names.

Valid variable names :

n , sam, sha_e, ma2009 , Mumbai

Invalid variable names :

3sam (should start with a letter)

mu*m (special character * is not allowed)

seat no (blank space is not allowed)

3.3. Keywords

Keywords are predefined reserved identifiers that have special meaning. They cannot be used as identifiers in our program. All the keywords should be in lower case letters.

Some of the keywords in C++ are:

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

Note : C++ is a "case sensitive" programming language. That means the identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the AGE variable is not the same as the age variable or the Age variable. These are three different variable identifiers.

3.4. CONSTANTS

A 'constant' is an entity that does not change. Constant refers to fixed value which remains unaffected during execution of the program.

Constants:

There are mainly three types of constants. Namely: **Numeric constants** (integer & real), **character constants** (single character constants) and **string constants**.

Numeric constants:- Numeric constant are positive or negative numbers .

There are four types of numeric constants: integer constant, floating point constant, hex constant & octal constant.

Integer constants

This are numerical constants that identify integer decimal values. Integer constant do not contain decimal points.

Integer constant	integer
	short integer(short)
	long integer(long)

Here are some valid integer constants

1776

707

-273

Short integer constants:- These are integer constants with maximum size of 16 bits. Short integer constants fall in the range of -32,768 to 32767 (-2^{15} to $2^{15}-1$).

long integer constants:- These are integer constants with size of 32 bits. Short integer constants fall in the range of -2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31}-1$).

Hex constants: - Constants which represent hexadecimal numbers (integer numbers with base 16 & are expressed in digits 0-9 and letters a-f) are called as hex constants. Hexadecimal constants begin with 0x or 0X .

octal constant	Short octal
	long octal

Example: 0x1C // Hexadecimal representation for decimal 28

Octal constants: - Constants which represent octal numbers (integer numbers with base 8 & are expressed in digits 0-7) are called as octal constants. octal constants begin with 0.

Hex constant	Short hexadecimal
	long hexadecimal

Example: 034 // Octal representation for decimal 28

More examples of Hex and Octal constants

```
//Octal Constants
012
0204
// Hexadecimal Constants
0xa or 0xA
0x84
```

Floating Point constants :- These are numbers with decimals and/or exponents. They can include either a decimal point or an e character or both a decimal point and an e character.

In exponential form, the **Floating Point Numbers** are represented as two parts. The part lying before the 'e' is the 'mantissa', and the one following 'e' is the '**exponent**'.

For e.g. 3.2e4 (this denotes the no. 3.2×10^4) in this case 3.2 is the mantissa and 4 is the exponent

Here are some valid floating point constants,

3.14159

$6.02e^{23}$ (this is same as 6.02×10^{23})

$1.6e^{-19}$ (this is same as 1.6×10^{-19})

The default type for floating point constants is double. If we explicitly want to express a float or long double numerical constant, we can use the f or l suffixes respectively:

3.14159L // long double

6.02e23f // float

Floating point constant	Single precision(float)	Size of 4 bytes
	Double precision(double)	Size of 8 bytes
	Long double	Size of 12 or 16 bytes

Note:- Space, comma, and characters other than digits are not allowed in Numeric constants.

Character constants:- A character constant is an alphabet, a single digit or a single special symbol enclosed within single inverted commas.

The maximum length of a character constant can be 1 character.

1 byte of memory is available to hold single character constant.

Ex: 'B', 'l', '#'

String constants :- String constant is a sequence of alphanumeric (letters and numbers) characters enclosed in double quotation marks whose maximum length is 255 characters.

Following are the examples of valid string constants.

- "Nirmala"
- "Rs 2500.00"
- "University of Mumbai"

Following are the examples of invalid string constants.

- Nirmala - Characters are not enclosed in double quotation marks.
- "information technology - Closing double quotation mark is missing
- 'Hello' - Characters are not enclosed in double quotation marks.

3.5. Data types in C++

When we declare variable, we actually create a memory location in computer's memory. Now each variable may not require same amount of memory space. The space occupied by variables in memory depends on the data type of the variable.

Following are fundamental data types in C++, with range of values and size.

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e ^{+/- 38} (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e ^{+/- 308} (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e ^{+/- 308} (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

3.6. Operators in C++

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulation.

C++ has a rich set of operators which can be classified as

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increments and Decrement Operators
6. Conditional Operators
7. Some special operators in C++ (comma, sizeof(), scope, New and delete operators)
8. Bitwise operators

3.6.1. Arithmetic operators

The five arithmetical operations supported by the C++ language are:

Operator	Meaning
+	Addition or Unary Plus
-	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

Here are some examples of arithmetic operators

$x + y$

$x - y$

$-x + y$

$a * b + c$

$-a * b$

here a, b, c, x, y are known as operands.

$\%$ - Modulo is the operation that gives the remainder of division of two values.

For example, $a = 17\% 3;$

the variable a will contain the value 2, since 2 is the remainder from dividing 17 by 3.

3.6.2. Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator come into picture. C supports the following relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form.

$exp1$ relational operator $exp2$

Where $exp1$ and $exp2$ are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

$6.5 <= 25$ TRUE

$-65 > 0$ FALSE

$10 < 7 + 5$ TRUE

Relational expressions are used in decision making statements of C++ such as if, while and for statements to decide the course of action of a running program.

3.6.3. Logical Operators

C++ has the following logical operators, they compare & evaluate logical expressions.

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

Logical AND (&&)

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

Truth table for && OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Example

$a > b \ \&\& \ x == 10$

The expression to the left is $a > b$ and that on the right is $x == 10$ the whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

Logical OR (||)

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Truth table for || OPERATOR

a	b	a b
true	true	true
true	false	true
false	true	true
false	false	false

Example

`a < m || a < n`

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

Logical NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

For example

`!(x >= y)` the NOT expression evaluates to true only if the value of x is neither greater than or equal to y

3.6.4. Assignment Operators (=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

Example

$x = a + b$

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can make use of compound assignment operators.

$a += b$ expression is equivalent to $a = a + b$

$a *= b$ expression is equivalent to $a = a * b$

$a /= b$ expression is equivalent to $a = a / b$ and so

on.

```
// compound assignment operators
#include <iostream.h>
main ()
{
int a, b=3;
a = b;
a+=2; // equivalent to a=a+2
cout << a;
return 0;
}
Output : 5
```

3.6.5. Unary operators (Increment and Decrement Operators)

The increment and decrement operators are the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below

1. ++ variable name
2. variable name++
3. --variable name
4. variable name--

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator -- subtracts the value 1 from the current value of operand. ++variable name and variable name++ mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following

```
m = 5;
```

```
y = ++m; (prefix)
```

In this case the value of **y** and **m** would be **6**

Suppose if we rewrite the above statement as

```
m = 5;
```

```
y = m++; (post fix)
```

Then the value of **y** will be **5** and that of **m** will be **6**. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

For example:

```
int a,b;
a=3;
cout<<a<<'\n';      3
cout<<a++<<'\n';    3
cout<<a<<'\n';      4
cout<<++a<<'\n';    5
cout<<a<<'\n';      5
cout<<a--<<'\n';    5
cout<<a<<'\n';      4
cout<<--a<<'\n';    3
```


consider $a=3, b=4$

let $c=a+b++$, here a will be 3 and b will be 4 . Thus c will be 7 , after this execution b will be incremented by 1 and will be assigned 5.

After execution values of $a=3, b=5$ & $c=7$

let $d=- --a+b++$, here a will be 2(decremented by one before using) and b will be 5. Thus d will be 6, after this execution b will be incremented by 1 and will be assigned 6.

After execution values of $a=2, b=6, c=7$ & $d=6$

let $d=a++-b--$, here a will be 2 and b will be 6. Thus d will be -4 , after this execution a will be incremented by 1 and will be assigned 3 and b will be decremented by one and will be assigned 5.

After execution values of $a=3, b=5, c=7$ & $d=-4$

3.6.6. Conditional or Ternary Operator

The conditional operator consists of 2 symbols the question mark (?) and the colon (:). Conditional operators acts on three expressions and therefore is also called as ternary operator.

The syntax for a ternary operator is as follows

$exp1 ? exp2 : exp3$

The ternary operator works as follows

$exp1$ is evaluated first. If the expression is true then $exp2$ is evaluated & its value becomes the value of the expression. If $exp1$ is false, $exp3$ is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example

Here are some examples of conditional operator

`7==5 ? 4 : 3` // returns 3, since 7 is not equal to 5.

`7==5+2 ? 4 : 3` // returns 4, since 7 is equal to 5+2.

`5>3 ? a : b` // returns the value of a, since 5 is greater than 3.

`a>b ? a : b` // returns whichever is greater, a or b.

If condition is true the expression will return result1, if it is not it will return result2.

```
// conditional operator
#include <iostream.h>
int main ()
{
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;
cout << c;
return 0;
}
Output: 7
```

3.6.7. Special operators**(a) Comma operator (,)**

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example:

```
a = (b=3, b+2);
```

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

(b) sizeof()

This operator accepts one parameter, which can be either a type or a variable and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type.

The value returned by sizeof is a constant, so it is always determined before program execution.

(c) scope operator (::)

:: is used as the scope resolution operator in C++ . With scope resolution operator we can define a class member function. Scope resolution operator is used to differentiate between members of base class with similar name.

(d) New & delete operators

Memory allocations and deallocations in C++ is carried out by using New & delete operators.

3.6.8. Bitwise operators

C++ provides operators to work with the individual bits. Bitwise operators are used in bitwise logical decision making.

Operator	Name	Description
<code>a&b</code>	and	Returns 1 if both bits are 1. 3 & 5 which is same as (011)&(101) After the operation we obtain (001) i.e. 1
<code>a b</code>	or	1 if either bit is 1. 3 5 which is same as (011) (101) After the operation we obtain (111) i.e. 7
<code>a^b</code>	xor	Returns 1 if both bits are different. 3 ^ 5 which is same as (011) ^ (101) After the operation we obtain (110) i.e. 6
<code>~a</code>	not	This unary operator inverts the bits. For Example: <code>x=23 (0001 0111)~x</code> will be <code>132(1110 1000)</code>
<code>x<<p</code>	left shift	Shifts the bits of x value to left by p positions. <code>X<<3</code> this will shift the bits of x to left by 3 positions. For Example: <code>x=23 (0001 0111)x<<3</code> resultant bit pattern after this operation is <code>1011 1000</code>
<code>x>>p</code>	right shift	Shifts the bits of x value to right by p positions. <code>X>>3</code> this will shift the bits of x to right by 3 positions. For Example: <code>x=23 (0001 0111)x>>3</code> resultant bit pattern after this operation is <code>0000 0010</code>

3.7. Operators precedence

Operator precedence deals with order of executions of operators when more than one operators are involved.

For example a mathematical expression involving `+`, `-`, `*`, `/` are executed using BODMAS (brackets of division, multiplication, addition and subtraction) rule.

Operators are assigned a particular level (precedence), and are executed according to these levels.

Operator on higher level is executed first compared to operator on lower level.

When operators of same level are encountered, then operators are executed either from left to right or right to left.

Their associativity indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
() [] ++ --	Parentheses (function call) Brackets (array subscript) Postfix increment/decrement	left-to-right
++ -- + -	Prefix increment/decrement Unary plus/minus	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/ greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
= +=	Assignment	right-to-left
-= *=	Addition/subtraction assignment	
/=	Multiplication/division assignment	

For Example:

Assume $x=3$, $y=7$

If ($a > b + 3$ && $b < a - 4$) in these expression we have operators $>$, $<$, $+$, $-$, &&

Order of execution will be

+

-

>

<

&&

3.8. Declaration of variables

Every variable used in the program should be declared to the compiler. The declaration does two things.

1. Tells the compiler the variables name.
2. Specifies what type of data the variable will hold.

The general syntax of declaration is to write the specifier of the desired data type (like int, float, etc.) followed by a valid variable name (identifier).

```
datatype variable name;
```

We can also declare more than one variable of same data type in a single declaration statement. In such case variables are separated by commas.

```
datatype variable 1, variable 2,.....,variable n;
```

A declaration statement must end with a semicolon.

Example:

```
int sum;
```

```
int number, salary;
```

```
double average, mean;
```

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented.

Note: Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero).

For example:

```
unsigned short int marks;
```

```
signed int temperature;
```

By default, most compiler assume the type to be signed, therefore `signed int temperature;` is same as `int temperature;`

```
#include <iostream.h>
int main ()
{
    // declaring variables:
    int a, b;
    int result;
    // process:
    a = 5;
    b = 2;
    a = a + 1;
    result = a - b;
    // print out the result:
    cout << result;
    // terminate the program:
    return 0;
}
```

Output : 4

3.9. Initialization of variables

Variables can be assigned value during declaration this is called initialization of variables. There are two ways to do this in C++:

Syntax :

```
type identifier = initial_value ;
```

For example, if we want to declare an integer variable initialized with a value 3 during declaration, it can be done as follows:

```
int a = 3;
```

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

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

For example:

```
int a (3 ) ;
```

3.10. Type casting

We always declare a variable for data type before using it in the program. In some situations we may need to convert data type of the variable to obtain a particular result. This can be achieved by type casting.

Type casting is the process of converting one data type into another or converting an expression of a given type into another.

Type casting can be carried out in two ways

- 1) Implicit Conversion
- 2) Explicit Conversion

(i) Implicit Conversion

Implicit Conversion is also called as converting by assignment operator or automatic conversion. In implicit conversions value gets automatically converted to the specific type to which it is assigned.

For example:

```
int x;  
float y;  
x=y;
```

Here the data type float namely variable *y* is converted to int and is assigned to the integer variable *x*. (in this case fractional part of *y* will be truncated and will be assigned to *y*).

(ii) Explicit Conversion

Explicit conversion is done with the help of cast operator. The cast operator is a technique used to forcefully convert one data type to the other. The whole process of conversion is called casting.

SYNTAX :

```
(datatype) expression;
```

For example:

```
1. float x;  
x= ( int ) ( 14.2/2 ) ;
```

Here *x* will be assigned value 7 instead of 7.1, as the expression will be casted by the cast operator.

```
2. char x;  
int y;  
y=(int) x;
```

This forces the character variable `x` to be converted to integer data type.

3.11. Preprocessor directives

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (`#`). The preprocessor is executed before the actual compilation of code begins.

These preprocessor directives extend only across a single line of code. No semicolon (`;`) is expected at the end of a preprocessor directive.

The `#include` is a "preprocessor" directive that tells the compiler to put code from the header into our program before actually creating the executable. By including header files, you can gain access to many different functions.

The concept of preprocessor directive

- Instructions are given to the compiler by making use of preprocessor directive.
- The preprocessor directive always begins with the `#` sign.
- The preprocessor directive can be placed anywhere in the program but most often they are included in the beginning of the program.
- For example :- `#include`, `#define`, `#else` etc.

- 1) `#include`-this preprocessor directive instructs the compiler to include a header file into the source code file.

Example `#include <iostream.h>`

- 2) `#define`-this preprocessor directive instructs the compiler to define a symbolic constant.

Example `#define PI 3.14`

Header files in C++ (with respect to Turbo & Borland C++ compilers)

- `iostream.h` - This standard header files contains a set of general purpose functions for handling input and output of data.
- `iomanip.h` - this header file is included to carry out operations related to manipulator functions. (Manipulator functions will be seen in detail in the next chapter.)

3.12. Defined constants or symbolic constants (#define)

A symbolic constant value can be defined as a preprocessor statement and used in the program as any other constant value.

These values may appear anywhere in the program, but must come before it is referenced in the program.

It is a standard practice to place them at the beginning of the program.

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.

SYNTAX

```
# define symbolicname value of constant
```

When the preprocessor encounters this directive, it replaces any occurrence of symbolic name in the rest of the code by value of constant. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++, it simply replaces any occurrence of symbolic name.

Valid examples of constant definitions are :

```
#define PI 3.14159
```

```
#define NAME "MAHESH"
```

This defines two new constants: PI and NAME. Once they are defined, we can use them in the rest of the program.

```
// defined constants: calculate circumference
#include <iostream.h>
#define PI 3.14159
#define NEWLINE '\n'
int main ()
{
    double r=5.0; // radius
    double circle;
    circle = 2 * PI * r;
    cout << circle;
    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).

The #define directive is not a C++ statement but a directive for the preprocessor. Therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If we append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

3.13. Declared constants (`const`)

With the `const` prefix we can declare constants with a specific type in the same way as we would do with a variable:

```
const int width = 100;
const char tabulator = '\t';
```

Here, `width` and `tabulator` are two types of constants. They are treated just like regular variables except that their values cannot be modified after their definition.

3.14. Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in “sub-scopes”, each one with its own name.

The format of namespaces is:

```
namespace identifier
{
    entities
}
```

Where `identifier` is any valid identifier and `entities` is the set of classes, objects and functions that are included within the namespace.

For example:

```
namespace myNamespace
{
    int a, b;
}
```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`. In order to access these variables from outside the `myNamespace` namespace we have to use the scope operator (`::`).

For example, to access the variables outside myNamespace we can write:

```
myNamespace::a
```

```
myNamespace::b
```

For Example :

```
// namespaces
#include <iostream>
using namespace std;
namespace first
{
    int var = 5;
}
namespace second
{
    double var = 3.1416;
}
int main () {
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}
```

OUTPUT :

```
5
```

```
3.1416
```

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second.

using

The keyword 'using' is used to introduce a name from a namespace into the declarative region.

The keyword 'using' can also be used as a directive to introduce an entire namespace.

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. We can include the using namespace std; statement in all programs that uses any entity defined in iostream.

```
#include <iostream>
using namespace std;
```

3.15. Reference Variable

Reference is a simple reference datatype that is less powerful but safer than the pointer type inherited from C. C++ references allow us to create a second name for a variable that we can use to read or modify the original data stored in that variable.

Syntax

Declaring a variable as a reference could be done by simply appending an ampersand to the type name, as follows

```
int& x= ....; // here x is a reference variable
```

- references do not require dereferencing in the way that pointers do we can just treat them as normal variables.
- when we create a reference to a variable, we need not do anything special to get the memory address. The compiler does it for us.

Example program

```
int x;
int& f= x;
// f is now a reference to x so this sets x to 56
f= 56;
cout << x <<endl;
```

OUTPUT : 56

3.16. Input and output statements in C++

Basic Input / Output

Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard. The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

(a) `cout`

By default, the standard output device of a program is screen, and the C++ stream object defined to access it is `cout`. Thus `cout` is used to display an object onto the standard output device.

`cout` is used alongwith with the insertion operator, which is `<<` (two "less than" signs).

The general syntax is ,

```
cout<<variable 1<<variable 2<<.....<<variable n;
```

Examples:

```
cout << "Hello"; // prints Hello on screen
int x=3;
cout << x; // prints number 3 on screen
```


Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names.

Consider the following statements,

```
cout << "x"; // prints x
```

```
cout << x; // prints the content of x variable
```

The insertion operator (<<) may be used more than once in a single statement as follows

```
cout << "welcome " << "to " << "Nirmala College";
```

This statement will print the message welcome to Nirmala College on the screen.

We can also use different data types in the same statement as follows

```
Age=26;
```

```
cout << "Hello, I am " << Age << " years old";
```

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it,

therefore, the following statements:

```
cout << " sentence1";
```

```
cout << " sentence2";
```

will be shown on the screen one following the other without any line break between them.

OUTPUT: sentence1 sentence2

In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (escape sequence):

```
cout << " sentence1\n ";  
cout << " sentence2\n sentence3";
```

This produces the following output:

```
Sentence1  
  
sentence2  
  
sentence3
```

we can also use the endl manipulator.

For example:

```
cout << " sentence1" << endl;  
cout << " sentence2" << endl;
```

OUTPUT:

```
sentence1  
  
sentence2
```

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an

additional behavior when it is used with buffered streams: the buffer is flushed.

Anyway, cout will be an unbuffered stream in most cases.

(b) cin

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 .

For example:

```
int x;  
cin >> x;
```

The first statement declares a variable `x` of type `int`, and the second accepts input from `cin` (the keyboard) in order to store it in this integer variable.

`cin` can only process the input from the keyboard once the RETURN(Enter) key has been pressed.

```
// I/O example  
#include <iostream.h>  
int main ()  
{  
int i;  
cout << "Please enter your age ";  
cin >> i;  
cout << "The value you entered is " << i;  
cout << " and its double is " << i*2 << "\n";  
return 0;  
}
```

Please enter your age: 25

The value you entered is 25 and its double is 50

The user of a program may insert a value which does not match with the data type of the variable included with `cin` and this may lead to errors in the program execution. So when you use the data input provided by `cin` extractions you will have to trust that the user of your program will be providing input which matches with the data type which is requested.

We can also accept two or more than two values through a single statement

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

In this cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator i.e. a space, a tab character or a newline.

3.17. Escape Sequences

Character combinations consisting of a backslash (\) followed by a character are called “escape sequences”. An escape sequence is regarded as a single character and is therefore valid as a character constant. Escape sequences are typically used to format output.

Escape sequence	meaning
\n	newline
\r	carriage return(Enter)
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (“)
\?	question mark (?)
\\	backslash (\)

3.18. Indenting programs

Always indent your program for readability. It is a good practice and useful while debugging your program. Good indentation makes it easy to see that your closing braces are correctly aligned and help you scan through your code quickly by giving visual clues about where the program flow changes.

Moreover it brings clarity to the program and maintains existing standard.

Example of an indented program

```
#include <iostream.h>
#include <conio.h>
void main()
{
    clrscr( );
    long int num1,num2,rnum=0;
    cout << "Enter an integer : " << endl;
    cin>>num1;
    num2=num1;
    do
    {
        rnum=rnum*10;
        int digit=num1%10;
        rnum+=digit;
        num1/=10;
    }while(num1);
    cout << "The integer you typed is " << num2
    << "." << endl;
    cout << "The reversed integer is " << rnum
    << "." << endl;
    getch();
}
```

3.19. Comments in C++

Make comments in your code often. Not only does it help people who are trying to help you understand it more, when you come back later, you'll be able to pick up where you left off and edit much faster.

A comment is text that the compiler ignores but that is useful for programmers. The compiler treats them as white space. You can use comments in testing to make certain lines of code inactive.

A C++ comment is written in one of the following ways:

- The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This syntax is the same as ANSI C.
- The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a "single-line comment."

Example:

```
c=a+b;    /*variable c will store sum of a and  
b*/    or    // variable c will store sum of a and b
```

3.20. Local & Global variables

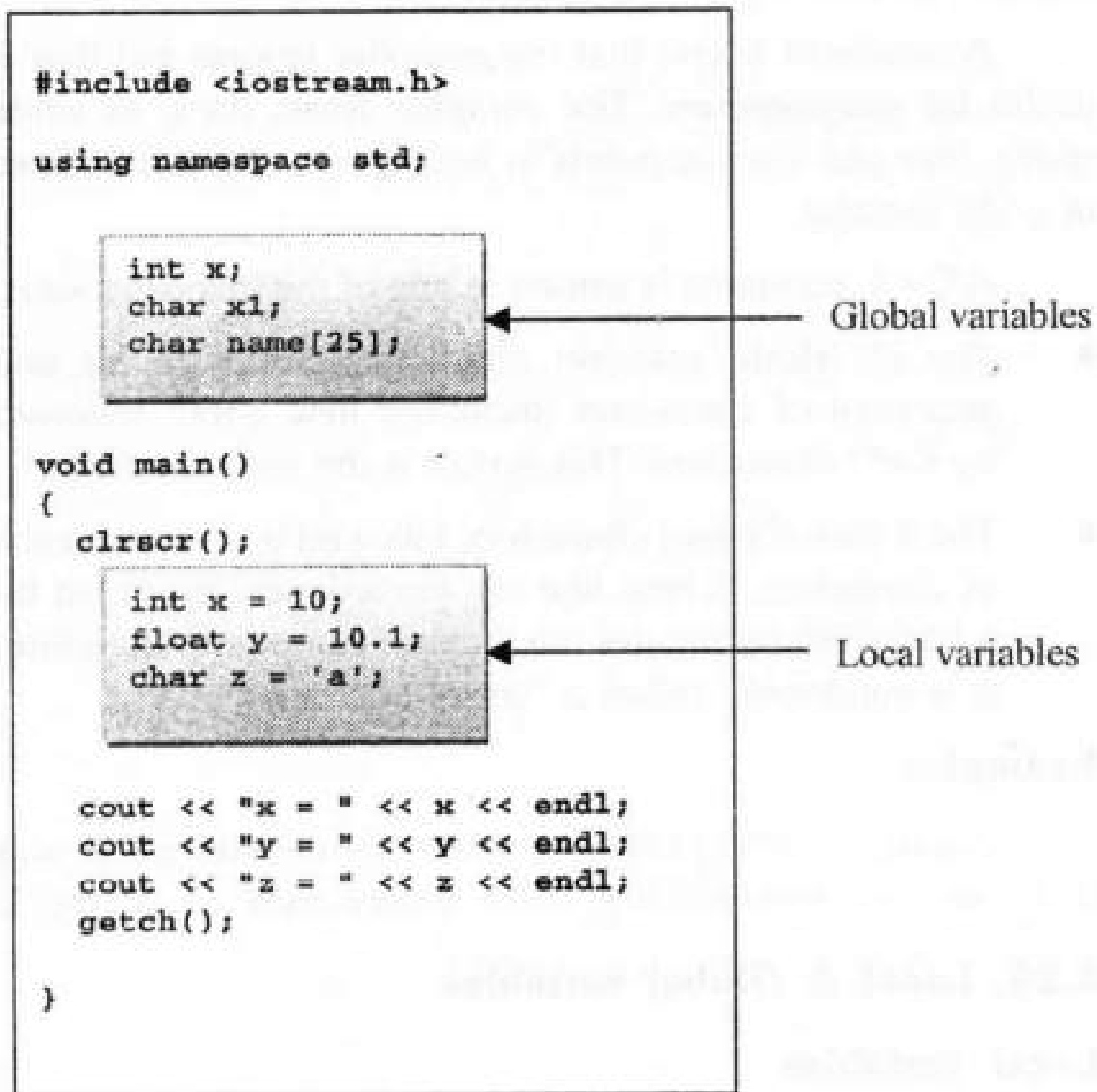
Local variables

Local variables must always be defined at the top of a block. When a local variable is defined - it is not initialised by the system, we must initialise it ourself. A local variable is defined inside a block and is only visible from within the block.

When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

Global variables

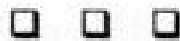
C++ has enhanced the use of global variables. Global variables are initialised by the system when we define them. Global variable is defined out of the block and is available throughout the program.



REVIEW QUESTIONS

- 1) What do you mean by identifiers and keywords?
- 2) Enlist rules of constructing identifiers (variables).
- 3) Explain the concept of preprocessor directive.
- 4) What do you mean by operators?
- 5) Write a note on increment and decrement operator with example.

- 6) Explain the use of comma operator in C++.
- 7) Write a note on constants in C++.
- 8) Explain the use of new and delete operators.
- 9) What are symbolic constants and how are they defined in C++.
- 10) What is the use of cin statement and how it can be used to accept data of different types?
- 11) What is the use of cout statement and how it can be used to print the data of different types?
- 12) Explain namespaces with an example.
- 13) What is reference variable and how does it differ from pointer.



Chapter 4

CHAPTER 4.

MANIPULATORS & CONTROL STRUCTURES

CONTENTS

- 4.1. Manipulators
- 4.2. Control Statements

4.1. Manipulators

Manipulators are operators used in C++ for formatting the output. The data is manipulated according to the desired output. To use manipulator functions in our program we need to include header file `<iomanip.h>`

There are number of manipulators available in C++. Some of the commonly used manipulators are as follows:

1) **endl**

The `endl` is an output manipulator & has the same functionality as the `'\n'` newline character.

For example:

```
cout << "Mumbai" << endl;  
cout << "University";
```

OUTPUT:

```
Mumbai  
University
```

2) **Setbase()**

The `setbase()` manipulator is used to convert the base of one numeric value into another base.

The syntax is:

```
setbase (base)
```

In case of decimal, base is 10 whereas for hexadecimal & octal it is 16 & 8 respectively.

A program to show the base of a numeric value of a variable using set base manipulator functions.

```
# include < iostream. h>
# include <iomanip.h>
void main ()
{
int value ;
cout <<"Enter number"<<endl ;
cin >>value ;
cout << "Decimal base="<<setbase(10)
<<value<<endl;
cout <<"Hexadecimal base="<<setbase(16)
<<value<<endl;
cout <<"Octal base=" <<setbase(8)<<value<<endl;
getch();
}
```

We can also do the the base conversion by using dec, hex and oct as follows,

```
# include < iostream. h>
# include <iomanip.h>
void main (void)
{
int value ;
cout <<"Enter number"<<endl ;
cin >>value ;
cout << " Decimal base="<<dec<<value<<endl;
cout <<"Hexadecimal base="<<hex<<value<<endl;
```

```
cout << "Octal base=" << oct << value << endl ;
getch();
}
```

3) Setw()

This manipulator sets the minimum field width on output.

The syntax is:

```
setw(x)
```

`setw` causes the number or string that follows it to be printed within a field of `x` characters wide and `x` is the argument set in `setw` manipulator.

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
int x=12345;
cout<< setw(5)<< x << endl;
cout << setw(6)<< x << endl;
cout << setw(7)<< x << endl;
}
```

The output of the above example is:

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

4) Setfill()

This is used after `setw` manipulator. If a value does not entirely fill a field, then the character specified in the `setfill` argument of the manipulator is used for filling the fields.

The syntax is:

```
setfill('character')
```

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
  int x,y;
  x=123;
  y=456;
  cout<<setfill('@');
  cout << setw(5) << x<< setw(6) << y<< endl;
}
```

OUTPUT:

@@123@@@456

5) Setprecision()

The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point.

The syntax is:

setprecision(integer value)

Example Program

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
void main( )
{
  float x=5, y=3, z;
  z=x/y;
  cout << setprecision(1) << x << endl;
  cout << setprecision(2) << x << endl;
  cout << setprecision(3) << x << endl; getch();
}
```

OUTPUT :

```
1.7
1.67
1.667
```

6) Ends

The Ends is a manipulator used to attach a null terminating character (' \0 ') at the end of a string. The Ends manipulator takes no argument.

A program to show how a null character is inserted using Ends manipulator while displaying a string onto the screen.

Example to explain Ends

```
# include<iostream.h>
# include<iomanip.h>
void main()
{
    int number = 131 ;
    cout<<' \ ' << "number =" << number << ends;
    cout<<' \ ' <<ends ;
}
```

OUTPUT:

```
"number = 131"
```

7) Ws

The manipulator function ws stands for white space. It is used to ignore the leading white space.

Example to explain Ws

```
# include <iostream.h>
# include <iomanip.h>
# include <conio.h>
void main ( )
{
    char name[100];
    cout<<"enter your name\n " ;
    cin>>ws ;
    cin>>name ;
    cout<<"Your name is= "<<name<< ends ;
    getch();
}
```

OUTPUT :

```
enter your name
Mahesh
Your name is= Mahesh
```

8) Flush()

The flush member function is used to empty the stream associated with the output. This function takes no input parameters. For output on the screen, this is not necessary as all output is flushed automatically.

```
// using Flush()
# include<iostream.h>
# include<iomanip.h>
void main()
{
```

```
    cout << "Welcome to University of Mumbai";  
    cout . flush();  
}
```

9) Precision

The precision member function is used to display the floating point value as defined by the user.

SYNTAX

```
cout.precision ( int x )
```

x represents the number of decimal places to be displayed.

Example1:

```
cout.precision(3);
```

Example2:

```
a = 1.234567;
```

```
b = 44.65 ;
```

```
cout.precision ( 3 ) ;
```

```
cout<<" a = "<<a<<" \n ";
```

```
cout<<" b = "<<b<<" \n ";
```

OUTPUT :

```
a = 1.235
```

```
b = 44.650
```

4.2. Control Statements

Often in programming we come across situations where we need to decide which part of the code should be executed or code should be executed only if a particular condition is satisfied.

In some programs it may be needed to repeat a part of the code, jump from a particular instruction or execute code if a particular condition is fulfilled or take decisions. In all such cases we can make use of control structures provided by C++.

While working with control structures we will be coming across compound statements also called as blocks. Let us understand what compound statements are.

A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

4.2.1. Conditional Statements

- if statement
- if ... else statement
- switch - case Statement

if statement

if statement is a decision making statement.

- This statement controls flow of execution of statements.
- This statement informs compiler of performing certain instructions only if a specified condition is met.
- Statement/s are executed only if the condition is true.

Syntax:-

```
if (condition)
single statement;
```

or

```
if (condition)
{
    multiple statements;
}
```

Example:-

```
int i;
cout <<"Type in an integer\n";
cin >>i;
if (i == 0)
{
    cout<<"The number is zero";
}
if (i > 0)
{
    cout<<"The number is positive";
}
if (i < 0)
{
    cout<<"The number is negative";
}
```

Output:

```
Type in an integer
4      (input provided by user)
The number is positive
```

if ... else

if ... else statement is also a decision making statement.

- This statement controls flow of execution of statements.
- This statement informs compiler of performing certain instructions if a specified condition is true or false.

- Thus we need to provide statements which should be executed when condition is true as well as false.

The if .. else statement has the form:

```
if (condition)
    statement1;
else
    statement2;
```

braces are used to include multiple statements.

```
if (condition)
{
    Multiple statements;
}
else
{
    Multiple statements;
}
```

The if.....else statement is a two way branching statement. On execution the condition is evaluated and if the condition is 'true' then set of statements before else is executed and if condition is 'false' then the set of statements after else are executed.

Example:-

```
int i;
cout<<"Insert an integer\n";
cin>>i ;
if (i > 0)
{
```

```

        cout<<"number is positive!";
    }
    else
    {
        cout<<"number is negative or zero!";
    }
}

```

❖ **Program to find the roots of a quadratic equation.**

```

#include <iostream.h>
#include <conio.h>
#include <math.h>
int main()
{
    clrscr();
    float a,b,c,d,root1,root2;
    cout << "Enter the 3 coefficients a, b, c : " <<
    endl;
    cin>>a>>b>>c;
    if(a==0){
        if(b==0)
            cout << "Both a and b cannot be 0 in ax^2 + bx
            + c = 0" << "\n";
        else
        {
            d=-c/b;
            cout << "The solution of the linear equation is
            : " << d << endl;
        }
    }
}

```

```
)
)
else
{
d=b*b-4*a*c;
if(d>0)
root1=(-b+sqrt(d))/(2*a);
root2=(-b-sqrt(d))/(2*a);
cout << "The first root = " << root1 << endl;
cout << "The second root = " << root2 << endl;
}
getch();
return 0;
}
```

OUTPUT

```
Enter the 3 coefficients a, b, c : 4 4 -3
The first root = 0.5
The second root = -1.5
```

Nesting of if.....else statement

It is possible to insert an if....else statement inside another if.....else statement this is called nesting and is used in programs where we need to check multiple conditions.

Example of nested if.....else

```
#include <iostream.h>
#include <conio.h>
void main ()
{ int result;
```

```

cout<<"Enter your marks";
cin>>result;
if (result < 35)
{
    cout<<"your result is fail";
}
else
{
    if (result <50)
    {
        cout<<"You have passed.";
    }
    else
    {
        cout<<"You got an A grade.";
    }
}
}
}
}

```

Output:-

```

Type in exam result
76 (input from the user)
You got an A grade

```

❖ **Program to enter the sale value and print the agent's commission based on the following criteria.**

Sales Value	Commission
$\leq 10,000$	5%
$> 10,000$ & $\leq 25,000$	10%
$> 25,000$	20%

```
#include <iostream.h>
#include <conio.h>
void main()
{
clrscr();
long int svalue;
float commission;
cout << "Enter the total sale value : " << endl;
cin>>svalue;
if(svalue<=10000)
{
commission=svalue*5/100;
cout << "For a total sale value of Rs." <<
svalue << ", ";
cout << "the agent's commission is Rs." <<
commission;
}
else if(svalue<=25000)
{
commission=svalue*10/100;
cout << "For a total sale value of Rs." <<
svalue << ", ";
cout << "the agent's commission is Rs." <<
commission;
}
else if(svalue>25000)
{
commission=svalue*20/100;
cout << "For a total sale value of Rs." <<
svalue << ", ";
```

```
cout << "the agent's commission is Rs." <<
commission;
}
getch();
}
```

OUTPUT

Enter the total sale value : 26000

For a total sale value of Rs. 26000, the agent's commission is Rs. 5200

Switch Statement

This statement is used to take a multi way decision.

Switch statement allows several conditions to be evaluated within one statement rather than using series of if.....else statements.

Only one variable(or an expression) is tested & all branches depend on the value of that variable.

The variable must be an integral type (int, long, short or char).

Each possible value of the variable can control a single branch.

default case is used to specify instructions to be carried out if variable doesn't match with any branch .

Syntax:-

```
switch (expression)
(
  case label1:
    block of statements;
    break;
  case label2:
    block of statements;
    break;
  case label3:
    block of statements;
    break;
  .
  .
  .
  .
  default:
    default statement/s;
    break;
)
```

Note :- break statement is used to make an exit out of the block in which it is included.

Example 1:

```
int number;
    number=2;
switch(number)
{
    case 0 :
        cout<<"you have entered zero\n";
        break;
    case 1 :
        cout<<"you have entered one \n";
        break;
    case 2 :
        cout<<"you have entered two \n";
        break;
    default :
        cout<<"you have entered a no. other than
        0,1,2 \n";
        break;
}
```

Output:

you have entered two

Example 2:

```
void main()
{
    char x;
    x='k';
    switch(x)
    {
        case 'a' :
```

```
        printf("vowel \n");
        break;
    case 'e' :
        printf("vowel \n");
        break;
    case 'i' :
        printf("vowel \n");
        break;
    case 'o' :
        printf("vowel \n");
        break;
    case 'u' :
        printf("vowel \n");
        break;
    default :
        printf("consonant \n");
        break;
    }
}
```

Output:

consonant

Note:-

- The case labels within switch statement must not be floating point numbers.

For example **case '4.5'** : is not allowed.

- The case labels within switch statement must not be a string expression.

For example **case "mumbai"**: is not allowed.

- The case labels within switch statement must not be an expression.

For example `case 'x+y' :` is not allowed.

- ❖ **Program to enter your choice and print the message accordingly using switch case statement**

```
#include <iostream.h>
#include <conio.h>
int main()
{
    clrscr();
    int choice;
    cout << "1. Talk" << endl;
    cout << "2. Eat" << endl;
    cout << "3. Play" << endl;
    cout << "4. Sleep" << endl;
    cout << "Enter your choice : " << endl;
    cin>>choice;
    switch(choice)
    {
        case 1 : cout << "You chose to talk...talking
too much is a bad habit." << endl;
        break;
        case 2 : cout << "You chose to eat...eating
healthy foodstuff is good." << endl;
        break;
        case 3 : cout << "You chose to play...playing
too much everyday is bad." << endl;
        break;
```

```
case 4 : cout << "You chose to sleep...sleeping
enough is a good habit." << endl;

break;

default : cout << "You did not choose
anything...so exit this program." << endl;

)

getch();

)
```

OUTPUT

Enter your choice : 2

You chose to eat...eating healthy foodstuff is good.

4.2.2. Looping statements

Often in programming, it is necessary to repeat certain instructions number of times or until a certain condition is met. It is tedious to simply type a certain statement or group of statements a large number of times.

The structures that enable computers to perform certain repetitive tasks are called loops.

C++ gives you a choice of three types of loop, while, do while and for.

- The while loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loop is similar, but the test occurs after the loop body is executed. This ensures that the loop body will run at least once.

- The for loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible, and novice programmers should take care not to abuse the power it offers.

For loop

Syntax:-

```
for(initialization; test; increment)
{
/*statements */
}
```

- The initialization statement is executed exactly once before the first evaluation of the test condition.
- It is used to assign an initial value to some variable.
- The initialization statement can also be used to declare and initialize variables used in the loop.
- The test expression is evaluated each time before the statements in the for loop executes.
- If the test expression is not true the loop is not executed and execution continues normally from the statements following the FOR loop.
- If the expression is true then the statements within the braces of the loop is executed.
- After each iteration of the loop, the increment statement is executed.
- The increment action can do other things, such as decrement.

Example of for loop

```
for(i = 1; i <= 10; i++)
{
cout<<i;
}
```

O/P 1 2 3 4 5 6 7 8 9 10

- ❖ **Program TO FIND factorial of a number n!**
($n! = 1 \times 2 \times 3 \times 4 \times \dots \times (n-1) \times n$)

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int p=1, i, n;
    cout<<"accept value of n";
    cin>>n;
    for(i=1; i<=n; i++)
        p = p*i;
    cout<<"factorial of "<<n<<" is"<<p;
    getch();
}
```

- ❖ **Program TO obtain $S = 1 \times 2 + 2 \times 3 + 3 \times 4 + 4 \times 5 + \dots + 9 \times 10$.**

```
#include<iostream.h>
#include<conio.h>
void main()
{
```

```
int s=0, i;
for(i=1; i<=9; i++)
    s = s + i*(i+1);
cout<<"sum is "<<s;
getch();
}
```

❖ **TO FIND $S = 1/2 + 3/4 + 5/6 + 7/8 + 9/10$**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    float s=0, i;
    for(i=1; i<=9; i=i+2)
        s = s + i/(i+1);
    cout<<"sum is "<<s;
    getch();
}
```

Nesting of For loop

It is possible to insert a for loop within another for loop. This is called as nesting of loops.

The following program illustrates nesting of loops.

❖ **Program to print the following pattern.**

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

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int i, k;
    for(i=1; i<=4; i=i++)
    {
        for(k=1; k<=i; k=k++)
        { cout<<"*"; }
        cout<<"\n";
    }
}
```

While loop

A while loop is the most basic type of loop. This loop will run as long as the condition is true. While loop is used when we are not certain that the loop will be executed.

Syntax:

```
While(condition)
{
    Statement/s;
    change in the intial condition;
}
```

Here is an example of a while loop.

```
int a=1;
while(a<5)
{
cout<<"a is \n"<<a;
a = a+1;
}
```

Output:

```
a is 1
a is 2
a is 3
a is 4
```

❖ **Program TO FIND sum of Fibonacci series starting at 1,1 i.e.**

1+1+2+3+5+8+13+21+34+55.

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int s=2, fn=1, sn=1,tn;
    tn=fn + sn;
    while(tn<=55)
    {
        s = s+ tn;
        fn = sn;
        sn = tn;
        tn = fn + sn;
    }
    cout<<"sum is "<<s;
    getch();
}
```

do.....while loop

Do while loop checks the condition after each run. As a result, even if the condition is zero (false), it will run at least once. We can use this loop structure when we are certain about the test condition. Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead before, granting at least one execution of statement even if condition is never fulfilled.

Syntax:

```
do
{
Statement/s;
change in the intial condition;
} while (condition);
```

Note:- the terminating semicolon.

Do.....While and while loop are functionally almost identical, with one important

difference: DoWhile loop is always guaranteed to execute at least once, but while loop will not execute at all if their condition is false on the first execution.

❖ **Program TO CHECK WHETHER THE GIVEN NUMBER IS A PALLINDROME or NOT.**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int number,rem,rev=0,temp;
```

```

    cout<<" Enter the number:";

    cin>>number;

    temp= number;

do

{
    rem=number%10;

    rev=rev*10+digit;

    number=number/10;

} while(number!=0);

    if (temp == reverse)

        cout<<"the number"<< temp<<"is a palindrome";

    else

        cout<<"the number"<< temp<<"is not a palindrome";

}

```

Comparison of for, while and do...while.

for	while	do.....while
for(initialization;test condition; increment) { Body of loop }	Initialization while(test condition) { Body of loop increment }	Initialization do { Body of loop increment } while (test condition);
Example:- for(x= 1; x <= 5; x=x+1) { Y=x*x; cout<<Y<<"\n"; }	Example:- x= 1; while(x <= 5) { (Y=x*x; cout<<Y<<"\n"; x=x+1; }	Example:- x= 1; do { {Y=x*x; cout<<Y<<"\n"; x=x+1; } while(x <= 5);

Output	Output	Output
1	1	1
4	4	4
9	9	9
16	16	16
25	25	25

break statement:-

Keyword `break` allows to make an exit from a loop. When `break` statement is encountered within a loop, execution of the loop stops and statements following the loop are executed. It can be used to end an infinite loop, or to force it to end before its natural end.

Syntax of the statement is,

```
break;
```

within a loop it takes the following form,

```
while(.....)
```

```
{.....
```

```
.....
```

```
If (condition)
```

```
break;
```

```
.....
```

```
.....} ←
```

Exit from the loop ,
execution continues
with statements
following loop.

```
// break loop example
```

```
#include <iostream.h>
```

```
#include <conio.h>
void main ()
{
    int n;
    for (n=10; n>0; n--)
    {
        cout << n << ", ";
        if (n==3)
        {
            cout << "countdown stopped!";
            break;
        }
    }
    return 0;
}
```

OUTPUT:

10, 9, 8, 7, 6, 5, 4, 3, countdown stopped!

continue statement:-

Keywords continue skips a particular iteration and continues with the next. When continue statement is encountered within a loop, execution of that iteration is skipped and loop continues with next iterations.

Syntax of the statement is,

continue;

within a loop it takes the following form,

```

→ while(.....)
  {.....
  .....
  If (condition)
  continue;
  .....
  .....}

```

Exit from loop ,
execution continues
with remaining iterations.

// Example of continue within a for loop

```

#include <iostream.h>
#include <conio.h>
void main ()
{
  for (int n=10; n>0; n--)
  {
    if (n==5)
    continue;
    cout << n << ", ";
  }
  return 0;
}

```

OUTPUT:

10, 9, 8, 7, 6, 4, 3, 2, 1

goto statement:-

This is an unconditional control transfer statement i.e. control is transferred during execution without any condition check.

Syntax of the statement is,

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

Execution control is transferred when **goto label** statement is encountered and is continued from **label**:

Statements between **goto label** and **label** are skipped.

label can be defined by using same rules as used while defining variable names.

For example:

```
#include <iostream.h>  
void main()  
{  
int a,b,c;  
a=2;  
b=3;  
c=a+b;  
goto sam;  
cout<<"value of a is"<<a;  
cout<<"value of b is"<<b;  
sam:  
cout<<"value of c is"<<c;  
}
```

OUTPUT:

value of c is 5

Note :- Generally goto statement is avoided in programs.

REVIEW QUESTIONS

1. Explain nested if with an example.
2. What is the difference between = & == ?
3. Write a note on if.....else statement.
4. Explain the difference between switch case and if.....else statements.
5. Write a note on looping in C++.
6. What is the use of break and continue statements and how do they differ from each other.
7. Why is goto statement not necessary for structured programming language in C++.
8. Write a program in C++ to find the sum $1^3-2^3+3^3-4^3.....+n^3$.
9. Write a program in C++ to generate the following series
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
10. Write a program in C++ to provide an input between 0-9 and print the string equivalent of the same.
11. Write a program in C++ to accept a number and count the no. of digits in the number.

□ □ □