UNIT-1

Object oriented thinking

Overview of C language:

1. C language is known as structure oriented language or procedure oriented language

2. Employs top-down programming approach where a problem is viewed as a sequence of tasks to be performed.

3. All program code of c can be executed in C++ but converse many not be possible

4. Function overloading and operator overloading are not possible.

5. Local variables can be declared only at the beginning of the block.

6. Program controls are through jumps and calls to subroutines. 7.Polymorphism, encapsulation and inheritance are not possible.

For solving the problems, the problem is divided into a number of modules. Each module is a subprogram.

8. Data abstraction property is not supported by procedure oriented language.

9. Data in procedure oriented language is open and can be accessed by any function.

Overview of C++ language:

1. C++ can be considered as an incremental version of c language which consists all programming language constructs with newly added features of object oriented programming.

2. c++ is structure(procedure) oriented and object oriented programming language.

3. The file extension of C++ program is “.CPP”

4. Function overloading and operator overloading are possible.

5. Variables can be declared in inline i.e when required

6. In c++ more emphasis is give on data rather than procedures

7. Polymorphism, encapsulation and inheritance are possible.

8. Data abstraction property is supported by c++.

9. Data access is limited. It can be accessed by providing various visibility modes both for data and member functions. there by providing data security by data hiding

10. Dymanic binding is supported by C++

11..It supports all features of c language

12. It can be called as an incremental version of c language

Object oriented programming

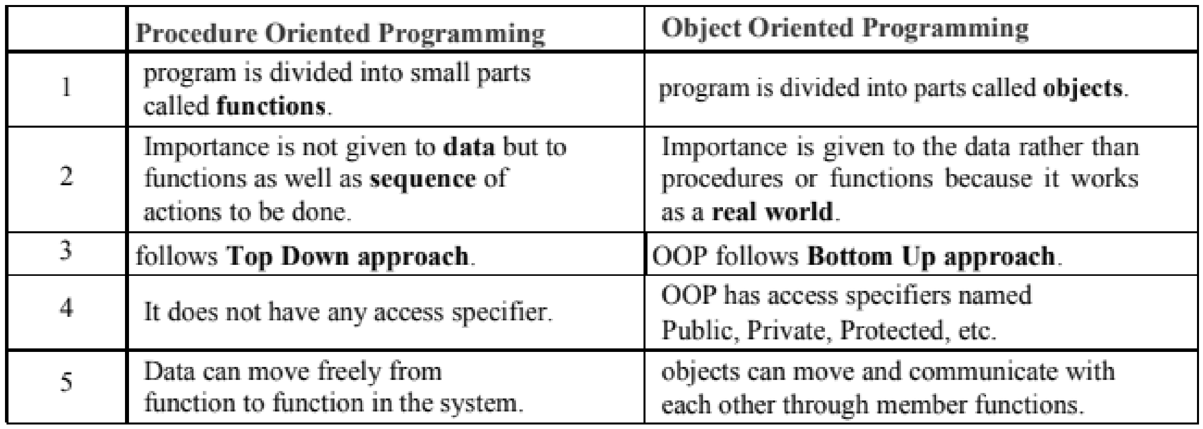
Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

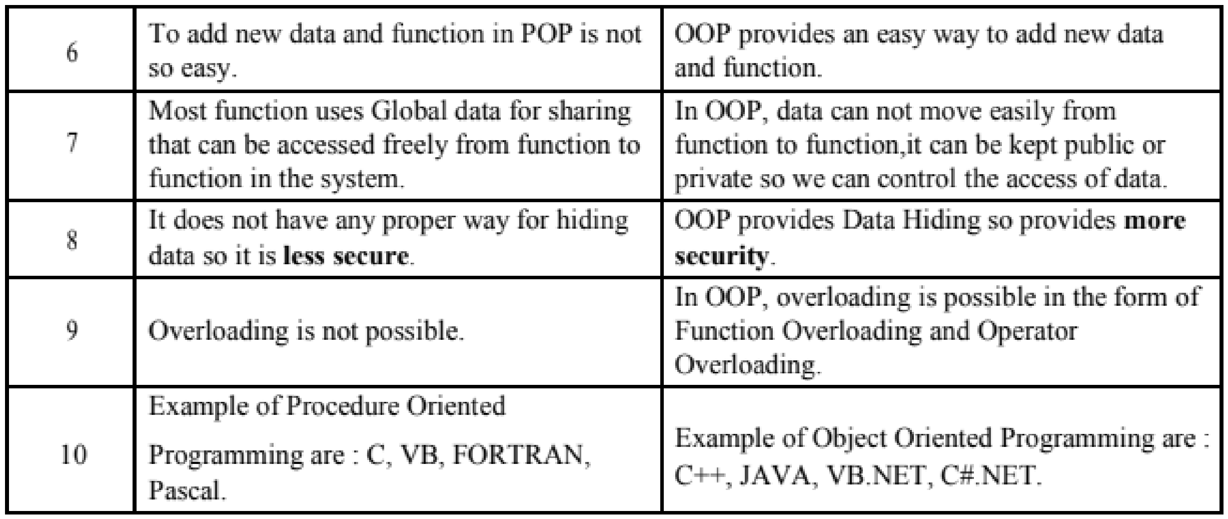
Procedural object oriented programming

Procedural programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself.

Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)



	Procedure Oriented Programming	Object Oriented Programming
1	program is divided into small parts called **functions**.	program is divided into parts called **objects**.
2	Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	Importance is given to the data rather than procedures or functions because it works as a **real world**.
3	follows **Top Down approach**.	OOP follows **Bottom Up approach**.
4	It does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
5	Data can move freely from function to function in the system.	objects can move and communicate with each other through member functions.



6	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
7	Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
8	It does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
9	Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
10	Example of Procedure Oriented Programming are : C, VB, FORTRAN, Pascal.	Example of Object Oriented Programming are : C++, JAVA, VB.NET, C#.NET.

Principles(or CONCEPTS) of object oriented programming:

1. Encapsulation

2. Data abstraction

3. Polymorphism

4. Inheritance

5. Dynamic binding

6. Message passing

Encapsulation: Wrapping of data and functions together as a single unit is known as encapsulation. By default data is not accessible to outside world and they are only accessible through the functions which are wrapped in a class. prevention of data direct access by the program is called data hiding or information hiding

Data abstraction :

Abstraction refers to the act of representing essential features without including the back ground details or explanation. Classes use the concept of abstraction and are defined as a list of attributes such as size, weight, cost and functions to operate on these attributes. They encapsulate all essential properties of the object that are to be created. The attributes are called as data members as they hold data and the functions which operate on these data are called as member functions.

Class use the concept of data abstraction so they are called abstract data type (ADT)

Polymorphism: Polymorphism comes from the Greek words "poly" and "morphism". "poly" means many and "morphism" means form i.e.. many forms. Polymorphism means the ability to take more than one form. For example, an operation have different behavior in different instances. The behavior depends upon the type of the data used in the operation.

Different ways to achieving polymorphism in C++ program:

1) Function overloading

2) Operator overloading

```
 #include<iostream.h>

using namespace std;

 int main()

{

int a=4; a=a<<2;

cout<<"a="<<a<<endl; return 0;

}
```

Inheritance:

Inheritance is the process by which one object can acquire the properties of another.

Inheritance is the most promising concept of OOP, which helps realize the goal of constructing software from reusable parts, rather than hand coding every system from scratch. Inheritance not only supports reuse across systems, but also directly facilitates extensibility within a system. Inheritance coupled with polymorphism and dynamic binding minimizes the amount of existing code to be modified while enhancing a system.

When the class child, inherits the class parent, the class child is referred to as derived class (sub class) and the class parent as a base class (super class). In this case, the class child has two parts: a derived part and an incremental part. The derived part is inherited from the class parent. The incremental part is the new code written specifically for the class child.

Dynamic binding:

Binding refers to linking of procedure call to the code to be executed in response to the call. Dynamic binding(or late binding) means the code associated with a given procedure call in not known until the time of call at run time.

Message passing:

An object oriented program consists of set of object that communicate with each other.

Objects communicates with each other by sending and receiving information .

A message for an object is a request for execution of a procedure and there fore invoke the function that is called for an object and generates result

Benefits of object oriented programming (OOPs)

Ø Reusability:

In OOP‟ s programs functions and modules that are written by a user can be reused by other users without any modification.

Ø Inheritance:

Through this we can eliminate redundant code and extend the use of existing classes.

Ø Data Hiding:

The programmer can hide the data and functions in a class from other classes. It helps the programmer to

build the secure programs.

ØReduced complexity of a problem:

The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.

Easy to Maintain and Upgrade:

OOP makes it easy to maintain and modify existing code as new objects

can be created with small differences to existing ones. Software complexity can be easily managed.

Ø Message Passing:

The technique of message communication between objects makes the interface with external systems easier.

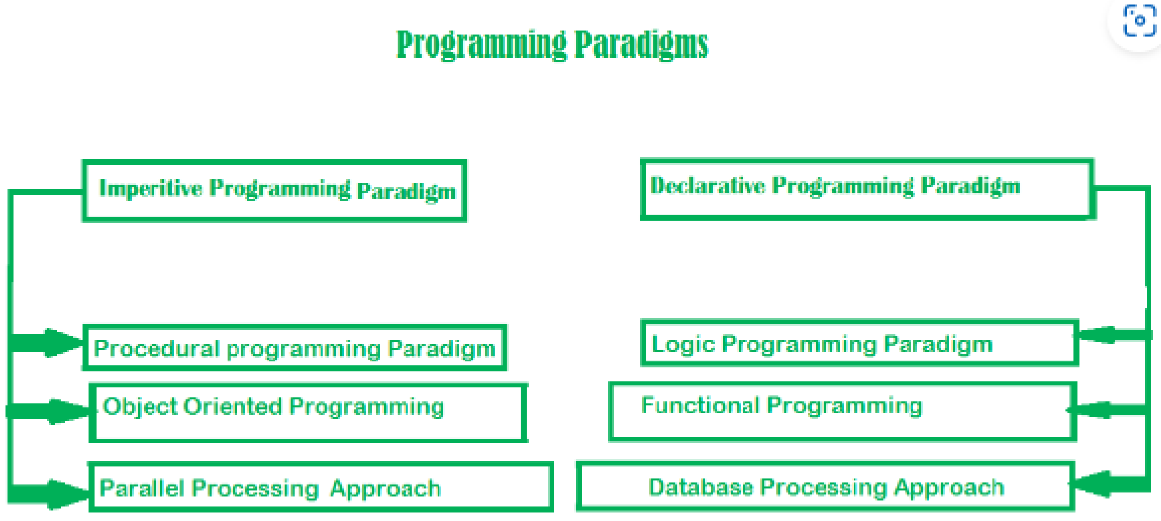
Ø Modifiability:

it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only

public interface that the external world has to a class is through the use of methods.

DIFFERENT PARADIGMS FOR PROBLEM SOLVING

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand. They are discussed below:


Programming Paradigms
Imperitive Programming Paradigm
Procedural programming Paradigm
Object Oriented Programming
Parallel Processing Approach
Declarative Programming Paradigm
Logic Programming Paradigm
Functional Programming
Database Processing Approach

1. Imperative programming paradigm:
It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consist of several statements and after execution of all the result is stored.
Advantage:
1. Very simple to implement
2. It contains loops, variables etc.
Disadvantage:
1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

```
Examples of Imperative programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson
Fortan : developed by John Backus for IBM
Basic : developed by John G Kemeny and Thomas E Kurtz
```

- C

```
// average of five number in C



int marks[5] = { 12, 32, 45, 13, 19 } int sum = 0;

float average = 0.0;

for (int i = 0; i < 5; i++) {

    sum = sum + marks[i];

}

average = sum / 5;
```

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

- **Procedural programming paradigm –**
  This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.

```
Examples of Procedural programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson
C++ : developed by Bjarne Stroustrup
Java : developed by James Gosling at Sun Microsystems
ColdFusion : developed by J J Allaire
Pascal : developed by Niklaus Wirth
```

- C++

```
#include <iostream>

using namespace std;
```

```
int main()

{

    int i, fact = 1, num;

    cout << "Enter any Number: ";

    cin >> number;

    for (i = 1; i <= num; i++) {

        fact = fact * i;

    }

    cout << "Factorial of " << num << " is: " << fact << endl;

    return 0;

}
```

Then comes OOP,

- **Object oriented programming –**
  The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. It can handle almost all kind of real life problems which are today in scenario.

Advantages:

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

- **Parallel processing approach –**
  Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.

2. Declarative programming paradigm:
It is divided as Logic, Functional, Database. In computer science the *declarative programming* is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic.It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code is actually doing. It just declares the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.

- **Logic programming paradigms –**
  It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism.
  In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

```
sum of two number in prolog:


  predicates

  sumoftwonumber(integer, integer)

clauses
```

```
  sum(0, 0).
   sum(n, r):-
        n1=n-1,
        sum(n1, r1),
        r=r1+n
```

- **Functional programming paradigms –**
  The functional programming paradigms has its roots in mathematics and it is language independent. The key principle of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions.The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like perl, javascript mostly uses this paradigm.

- **Database/Data driven programming approach –**
  This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

```
CREATE DATABASE databaseAddress;
CREATE TABLE Addr (
    PersonID int,
    LastName varchar(200),
    FirstName varchar(200),
    Address varchar(200),
    City varchar(200),
    State varchar(200)
);
```

BASIC STRUCTURE OF C++ LANGUAGE :

The program written in C++ language follows this basic structure. The sequence of sections should be as they are in the basic structure. A C program should have one or more sections but the sequence of sections is to be followed.

1. Documentation section

2. Linking section

3. Definition section

4. Global declaration section & class declarations

5.Member function definition

6. Main function

section main()

{

Declaration section Executable section

} >>

1. DOCUMENTATION SECTION : comes first and is used to document the use of logic or reasons in your program. It can be used to write the program's objective, developer and logic details. The documentation is done in C language with /* and */ . Whatever is written between these two are called comments.

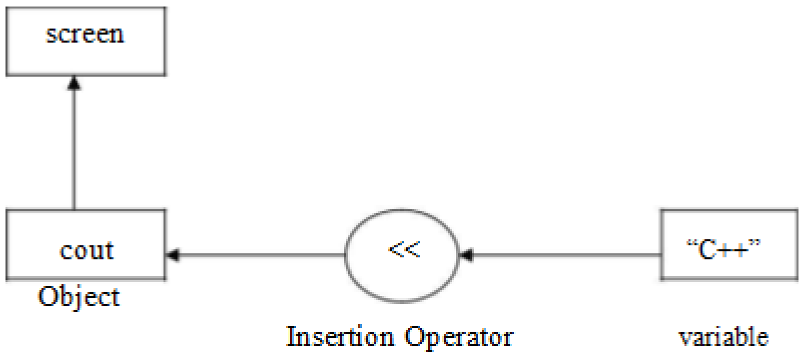
2. LINKING SECTION : This section tells the compiler to link the certain occurrences of keywords or functions in your program to the header files specified in this section.

e.g. #include<iostream>

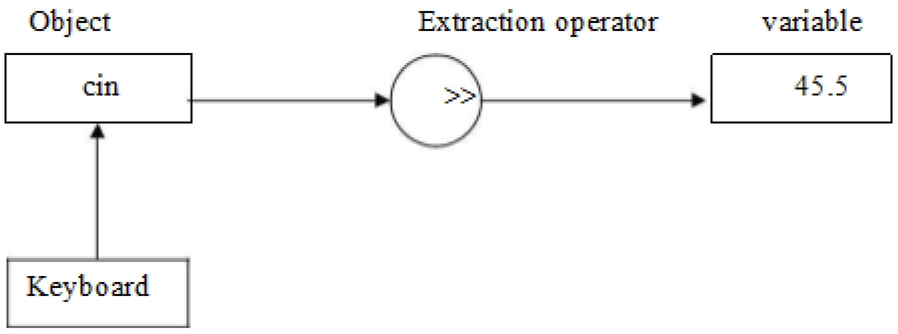
using namespace std;

- directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for cout and cin.

- cout is a predefined object that represents the standard output stream. The operator << is an insertion operator, causes the string in double quotes to be displayed on the screen.



The statement cin>>n; is an input statement and causes the program to wait for the user to type in a number. The number keyed is placed on the variable "n". The identifier cin is a predefined object in C++ that corresponds to the standard input stream. The operator >> is known as extraction operator. It extracts the value from the keyboard and assigns it to the value variable on its right.



3. DEFINITION SECTION : It is used to declare some constants and assign them some value. e.g. #define MAX 25

Here #define is a compiler directive which tells the compiler whenever MAX is found in the program replace it with 25.

4. GLOBAL DECLARATION SECTION : Here the variables and class definations which are used through out the program (including main and other functions) are declared so as to make them global(i.e accessible to all parts of program). A CLASS is a collection of data and functions that act or manipulate the data. The data components of a class are called data members and function components of a class are called member functions

A class ca also termed as a blue print or prototype that defines the variable or functions common to all objects of certain kind. It is a user defined data type

e.g.int i; //this declaration is done outside and before main()

5. SUB PROGRAM OR FUNCTION SECTION

This has all the sub programs or the functions which our program needs.

void display()

{

cout<<"C++ is better that C";

}

SIMPLE „C++" PROGRAM:

#include<iostream> using namespace std; void display()

{

cout<<"C++ is better that C";

}

int main()

{

display() return 0;

}

6. MAIN FUNCTION SECTION : It tells the compiler where to start the execution from main()

{

point from execution starts

}

main function has two sections

1. declaration section : In this the variables and their data types are declared.

2. Executable section or instruction section : This has the part of program which actually performs the task we need.

namespace:

namespace is used to define a scope that could hold global identifiers. ex:-namespace scope for c++ standard library.

A classes ,functions and templates are declared within the namespace named std using namespace std;-->directive can be used.

user defined name space:

syntax for defining name space is

namespace namespace_name

{

//declarations of variables.functions,classes etc...

}

ex: #include<iostream.h>

using namespace std;

namespace sample

{`

it m;

void display(int n)

{

cout<<"in namespace N="<<n<<endl;

```
}

}

using namespace sample; int main()

{

int a=5; m=100;

display(200);

cout<<"M in sample name space:"<<sample::m; return 0;}

#include<iostream.h>
```

This directive causes the preprocessor to add content of iostream file to the program. some old versions of C++ used iostream.h .if complier does not support ANSI (american nation standard institute) C++ then use header file iostream.h

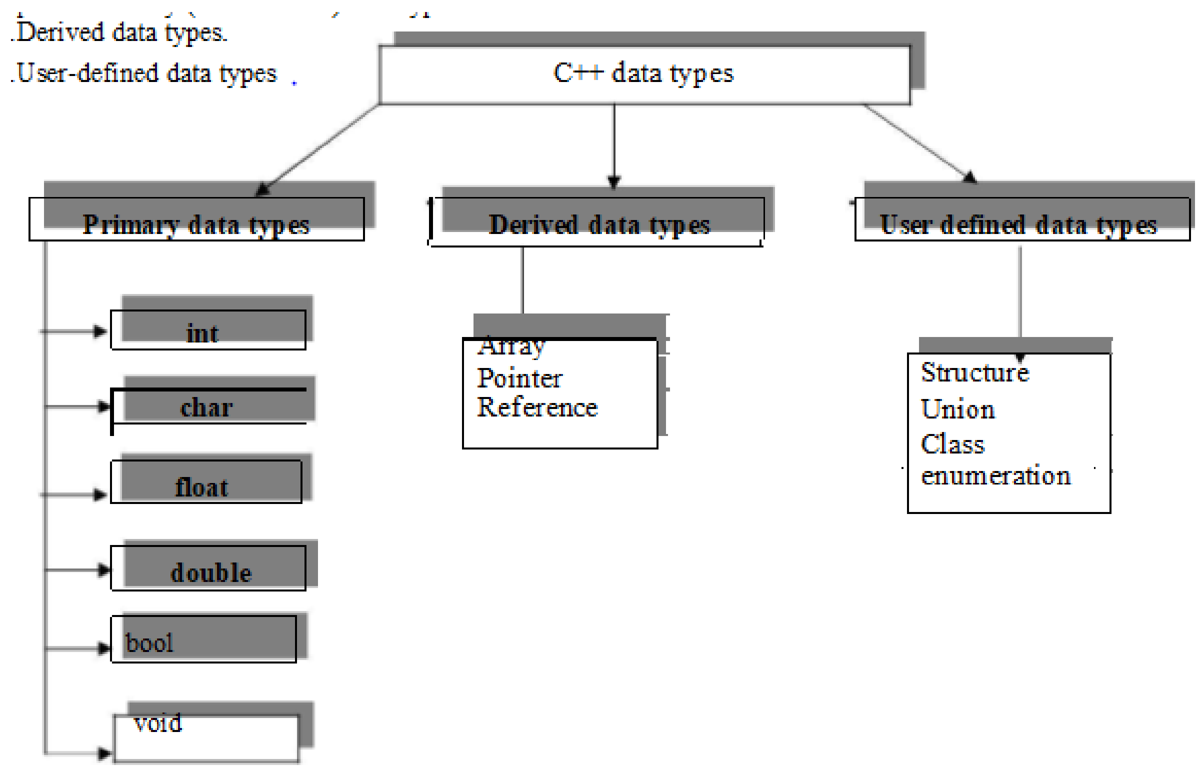
DATA TYPES:

A data type is used to indicate the type of data value stored in a variable. All C compilers support a variety of data types. This variety of data types allows the programmer to select the type appropriate to the needs of the application as well as the machine. ANSI C supports the following classes of data types:

1.Primary (fundamental) data types.

2. Derived data types.

3. User-defined data type



Primary data types:

1. integer data type

2.character data type

3.float point data type

4.Boolean data type

5.void data type

integer data type:-

This data type is used to store whole numbers. These numbers do not contain the decimal part. The size of the integer depends upon the world length of a machine (16-bit or 32-bit). On a 16-bit machine, the range of integer values is - 32,768 to +32,767.integer variables are declared by keyword int. C provides control over range of integer values and storage space occupied by these values through the data types: short int, int, long int in both signed and unsigned forms.

Signed integers: (16-bit machine):

A signed integer uses 1 bit for sign and 15 bits for the magnitude of the number

MSB(most significant bit)

=100(10)

0000000001100100(2)

Representation of negative number :

-100(10)=1111111110011100(2)

NOTE: Signed bit (MSB BIT): 0 represents positive integer, 1 represents negative numbers

Unsigned integers: Unsigned integers use all 16 bits to store the magnitude. Stores numbers does not have any sign & Size qualifier and range of integer data

DATA TYPE MEMORY REQUIRED OR STORAGE SIZE IN BYTES RANGE FORMAT SPECIER TURBO C (16 BIT) GCC/ COMPILERS IN LINUX (32 BIT) TURBO C (16 BIT) GCC (32 BIT)

DATA TYPE	MEMORY REQUIRED OR STORAGE SIZE IN BYTES		RANGE		FORMAT SPECIER
	TURBO C (16 BIT)	GCC/ COMPILERS IN LINUX (32 BIT)	TURBO C (16 BIT)	GCC (32 BIT)	
short int or signed short int	2	2	-32768 To 32767 (-2^{15} to $+2^{15}-1$)	-32768 To 32767 (-2^{15} to $+2^{15}-1$)	%hd
short int or signed short int	2	2	0 to 65535 (0 to +2 -1)	0 to 65535 (0 to +2 -1)	%hu
signed int or int	2	4	-32768 To 32767 (-2^{15} to $+2^{15}-1$)	-2,147,843,648 to 2,147,843,647 (-2^{31} to $+2^{31}-1$)	%d or %i
unsigned int	2	4	0 to 65535 (0 to $+2^{16}-1$)	0 to 4,294,967,295 (0 to$2^{32}-1$)	%u
long int or signed long int	4	4	-2,147,843,648 to 2,147,843,647 (-2^{31} to $+2^{31}-1$)	-2,147,843,648 to 2,147,843,647 (-2^{31} to $+2^{31}-1$)	%ld
unsigned long int	4	4	0 to 4,294,967,295 (0 to$2^{32}-1$)	0 to 4,294,967,295 (0 to$2^{32}-1$)	%lu
long long int or signed long long int	Not supported	8	--------	-9223372036854775808 To 9223372036854775807 (-2^{63} to $+2^{63}-1$)	%Ld

Character data type: (char)

A single character can be defined as a character data type. Character data type occupies one byte of memory for storage of character. The qualifiers signed or unsigned can be applied on char data type. char is the key word used for declaring variables

size and range of character data type on 16 bit or 32 bit machine can be shown below

Data type MEMORY REQUIRED OR STORAGE SIZE (in bytes) RANGE FORMAT SPECIER

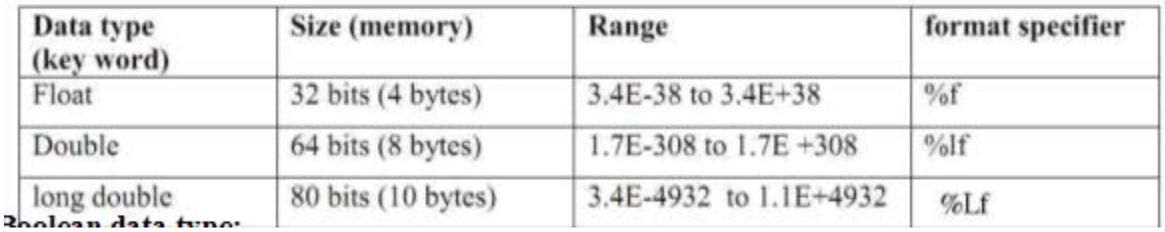
Data type	MEMORY REQUIRED OR STORAGE SIZE (in bytes)	RANGE	FORMAT SPECIER
char or signed char	1	-128 to 127(-2^{7} to $2^{7}-1$)	%c
Unsigned signed char	1	0 to 256 (0 to 2 -1)	%c

Floating Point Types:

Floating point number represents a real number with 6 digits precision occupies 4 bytes of memory. Floating point variables are declared by the keyword float.

Double floating point data type occupies 8 bytes of memory giving 14 digits of precision. These are also known as double precision numbers. Variables are declared by keyword double

long double refers to a floating point data type that is often more precise than double precision.



Data type (key word)	Size (memory)	Range	format specifier
Float	32 bits (4 bytes)	3.4E-38 to 3.4E+38	%f
Double	64 bits (8 bytes)	1.7E-308 to 1.7E +308	%lf
long double	80 bits (10 bytes)	3.4E-4932 to 1.1E+4932	%Lf

Boolean or logical data type is a data type, having two values (usually denoted true and false), intended to represent the truth values of logic and Boolean algebra. It is named after George Boole, who first defined an algebraic system of logic in the mid 19th century. The Boolean data type is the primary result of conditional statements, which allow different actions and change control flow depending on whether a programmer-specified Boolean condition evaluates to true or false.

C99 added a Boolean (true/false) type which is defined in the <stdbool.h> header Boolean variable is defined by kkey word bool; Ex:

bool b;

where b is a variable which can store true(1) of false (0)

Void type

The void type has no values. This is usually used to specify the return type of functions. The type of the function said to be void when it does not return any value to the calling function. This is also used for declaring general purpose pointer called void pointer.

Derived data types.

Derived datatypes are Arrays , pointer and references are examples for derived data types.

User-defined data types:

The data types defined by the user are known as the user-defined data types. They are structure,union,class and enumeration

Variables:A named memory location is called variable.

OR

It is an identifier used to store the value of particular data type in the memory.

Since variable name is identifier we use following rules which are same as of identifier Boolean data type:-

Rules for declaring Variables names:

Ø The first character must be an alphabet or underscore.

Ø It must consist of only letters, digits and underscore.

Ø Identifiers may have any length but only first 31 characters are significant.

Ø It must not contain white space or blank space.

Ø We should not use keywords as identifiers.

Ø Upper and lower case letters are different.

Ø Variable names must be unique in the given scope

Ex:int a,b,a;//is in valid

Int a,b;//is valid

Variable declaration: The declaration of variable gives the name for memory location and its size and specifies the range of value that can be stored in that location.

Syntax:

Data type variable name;

Ex:

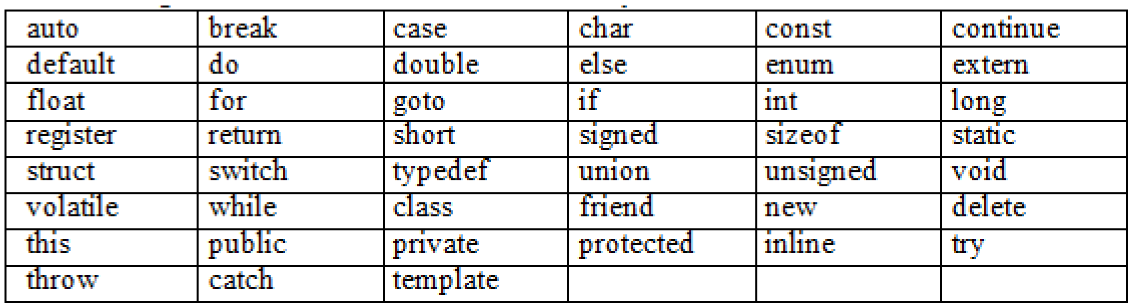
int a=10;

float x=2.3;

KEYWORDS :

There are certain words, called keywords (reserved words) that have a predefined meaning in C++“ language. These keywords are only to be used for their intended purpose and not as identifiers.

The following table shows the standard „C++“ keywords



auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while	class	friend	new	delete
this	public	private	protected	inline	try
throw	catch	template			

OPERATORS AND EXPRESSIONS

An operator is a symbol which represents a particular operation that can be performed on data. An operand is the object on which an operation is performed.

By combining the operators and operands we form an expression. An expression is a sequence of operands and operators that reduces to a single value.

C operators can be classified as

1. Arithmetic operators

2. Relational operators

3. Logical operators

4. Assignment operators

5. Increment or Decrement operators

6. Conditional operator

7. Bit wise operators

8. unary operator

9. Special operators

10.Additional operators in c++

1. ARITHMETIC OPERATORS : All basic

operator	meaning
+	add
-	subtract
*	multiplication
/	division
%	modulo division(remainder)

An arithmetic operation involving only real operands(or integer operands) is called real arithmetic(or integer arithmetic). If a combination of arithmetic and real is called mixed mode arithmetic.

```
/*C program on Integer Arithmetic Expressions*/

#include<iostraem.h>

void main()

{

int a, b;

cout<"Enter any two integers";

 cin>>a>>b;
cout<<"a+b"<< a+b;

 cout<<"a-b"<< a-b;

cout<<"a*b"<< a*b;

 cout<<"a/b"<< a/b;

 cout<<"a%b"<< a%b;

}
```

OUTPUT:

```
a+b=23 a-b=17 a*b=60 a/b=6 a% b=2
```

2. RELATIONAL OPERATORS : We often compare two quantities and depending on their relation take certain decisions for that comparison we use relational operators.

operator	meaning
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

```
/* C program on relational operators*/

#include<iostream.h>

 void main()

{

int a,b;
```

```
 clrscr();

cout<<"Enter a, b values:";

cin>>a>>b;

cout<<"a>b"<< a>b;

cout<<"a>=b"<< a>=b;

cout<<"a<b"<< a<b;

cout<<"a<=b"<< a<=b;

cout<<"a==b"<< a==b;

cout<<"a!=b"<< a!=b;

} OUTPUT:

Enter a, b values: 5 9

a>b: 0 //false

a<b: 1 //true

a>=a: 1 //true

a<=b: 1 //true

a==b: 0 //false

a!=b: 1 //true
```

3. LOGICAL OPERATORS:

Logical Data: A piece of data is called logical if it conveys the idea of true or false. In C++ we use int data type to represent logical data. If the data value is zero, it is considered as false. If it is non -zero (1 or any integer other than 0) it is considered as true. C++ has three logical operators for combining logical values and creating new logical values:

Note:Below program works in compiler that support C99 standards

```
#include<iostream.h>

#include<stdbool.h>

int main()

{

bool a,b;

/*logical and*/

a=0;b=0;

cout<<" a&&b "<< a&&b<<endl;

a=0;b=1;

cout<<" a&&b "<< a&&b<<endl;

a=1;b=0;

cout<<" a&&b "<< a&&b<<endl;

a=1;b=1;

cout<<" a&&b "<< a&&b<<endl;

/*logical or*/

a=0;b=0;

cout<<" a||b "<< a||b<<endl;

a=0;b=1;

cout<<" a||b "<< a||b<<endl;

a=1;b=0;

cout<<" a||b "<< a||b<<endl;

a=1;b=1;
cout<<" a||b "<< a||b<<endl;

/*logical not*/

a=0;

cout<<" a||b "<< a||b<<endl;

a=1;

cout<<" a||b "<< a||b<<endl;

return 0;

} OUTPUT:

0&&0=0

0&&1=0

1&&0=0

1&&1=1

0||0=0

0||1=1

1||0=1

1||1=1

!0 =1

!1 =0
```

4. ASSIGNMENT OPERATOR:

The assignment expression evaluates the operand on the right side of the operator (=) and places its value in the variable on the left.

Note: The left operand in an assignment expression must be a single variable. There are two forms of assignment:

•Simple assignment

•Compound assignment

Simple assignment :

In algebraic expressions we found these expressions. Ex: a=5; a=a+1; a=b+1;

Here, the left side operand must be a variable but not a constant. The left side variable must be able to receive a value of the expression. If the left operand cannot receive a value and we assign one to it, we get a compile error.

Compound Assignment:

A compound assignment is a shorthand notation for a simple assignment. It requires that the left operand be repeated as a part of the right expression. Syntax: variable operator+=value

Ex:

A+=1; it is equivalent to A=A+1;

Advantages of using shorthand assignment operator:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.

2. The statement is more concise and easier to read.

3. The statement is more efficient.

5.INCREMENT (++) AND DECREMENT (--) OPERATORS:

The operator ++ adds one to its operand where as the operator - - subtracts one from its operand. These operators are unary operators and take the following form:

Both the increment and decrement operators may either precede or follow the operand.

Postfix Increment/Decrement :(a++/a--)

In postfix increment (Decrement) the value is incremented (decremented) by one. Thus the a++ has the same effect as a=a+1; a--has the same effect as a=a-1.

The difference between a++ and a+1 is, if ++ is after the operand, the increment takes place after the expression is evaluated.

The operand in a postfix expression must be a variable. Ex1:

int a=5;

B=a++; Here the value of B is 5. the value of a is 6. Ex2:

int x=4; y=x--; Here the value of y is 4, x value is 3 Prefix Increment/Decrement (++a/ --a)

In prefix increment (decrement) the effect takes place before the expression that contains the operator is evaluated. It is the reverse of the postfix operation. ++a has the same effect as a=a+1.

--a has the same effect as a=a-1. Ex: int b=4;

A= ++b;

In this case the value of b would be 5 and A would be 5.

The effects of both postfix and prefix increment is the same: the variable is incremented by

1. But they behave differently when they used in expressions as shown above. The execution of these operators is fast when compared to the equivalent assignment statement.

```
#include<iostream.h>

int main()

{

int a=1; int b=5;

++a;

cout<<"a="<<a<<endl;
--b;

cout<<"b="<<b<<endl;

 cout<<"a="<<a++<<endl;

 cout<<"a="<<a<<endl;

cout<<"b="<<b--<<endl;

cout<<"b="<<b<<endl;

return 0;

}
```

```
a=2 b=4 a=2 a=3 b=4 b=3
```

6. CONDITIONAL OPERATOR OR TERNARY OPERATOR:

A ternary operator requires two operands to operate Syntax:

```
#include<iostream.h>

void main()

{

int a, b,c;

cout<<"Enter a and b values:";

cin>>a>>b;

c=a>b?a:b;

cout<<"largest of a and b is "<<c;

}
```

```
Enter a and b values:1 5 largest of a and b is 5
```

7. BIT WISE OPERATORS : C supports special operators known as bit wise operators for manipulation of data at bit level. They are not applied to float or double.

operator	meaning
&	Bitwise AND
^	Bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

Bitwise AND operator (&)

The bitwise AND operator is a binary operator it requires two integral operands (character or integer). It does a bitwise comparison as shown below:

Shift Operators

The shift operators move bits to the right or the left. These are of two types:

. •Bitwise shift right operator

. •Bitwise shift left

Bitwise shift right operator

It is a binary operator it requires two integral operands. The first operand is the value to be shifted and the second operand specifies the number of bits to be shifted. When bits are shifted to right,the bits at the right most end are deleted and a zero is inserted at the MSB bit.

```
#include<iostream.h>

void main()

{

int x,shift;
cout<<”Enter a number:”;
 cin>>x;
cout<<”enter now many times to right shift: “;
 cin>>shift;
cout<<”Before Right Shift:”<<x
x=x>>shift;
cout<<”After right shift:”<<x;
}
```

Run1:

```
Enter a number:8
enter now many times to right shift:1 Before Right Shift:8
After right shift:4
```

Flow Control statements:-The flow of execution of statements in a program is called as control. Control statement is a statement which controls flow of execution of the program. Control statements are classified into following categories.

1.Sequential control statements

2.Conditional control statements

3.Unconditional control statements

1. Sequential control statements:-

(type) expression;

Or

type (expression);

Sequential control

statements ensures that the instructions(or

statements) are executed in the same order in which

they appear in the program. i.e. By default system

executes the statements in the program in sequential order.

2. Conditional control statements

:Statements that are executed when a condition is true. These statements are divided into three categories. they are

1. Decision making statements 2.Switch case control statement or

3.Loop control statements or repetations

1.Decision making statements:- These statements are used to control the flow of execution of a program by making a decision depending on a condition, hence they are named as decision making statements.

Decision making statements are of four types 1.Simple if

2. if else 3.nested if else 4.If else ladder

1.Simple if statement: if the test expression is true then if statement executes statements that immediately follow if

Syntax:

```
If(test expression)

{

List of statements;

}
```

```
/*largest of two numbers*/

#include<stdio.h> int main()

{
```

```
int a,b;

cout<<“Enter any two integers:”; cin>>a>>b;

if(a>b)

if(b>a)

cout<<“A is larger than B\n A=”<<a;

cout<<“B is larger than A\n A=”<<b;

return 0;

}
```

2. if –else statement:

If test expression is true block of statements following if are executed and if test expression is false then statements in else block are executed

```
if (test expression)

{

}

else

{

}

statement block1;

statement block2;

/*largest of two numbers*/

#include<iostream.h>

int main()
```

```
{

int a,b;

cout<<”Enter any two integers:”; cin>>a>>b;

if(a>b) cout<<“A is larger than B\n A=”<<a; else cout<<“B is larger than A\n
A=”<<b;

return 0;

}
```

3. Nesting of if-else statements It's also possible to nest one if statement inside another. When a series of decisions are to be made.

If –else statement placed inside another if else statement Syntax:

```
If(test expression)

{If(test expression) {

//statements

}

else

{ //statements

}

}

else

{If(test expression) {

//statements

}

else
```

```
{ //statements
}
}
/*largest of three numbers*/ #include<iostream.h> #include<conio.h>
int main()
{
int a,b,c;
cout<<"Enter a,b,c values:"; cin>>a>>b>>c;
if(a>b)
{
}
else
{if(b>c)
if(a>c)
{
}
else
{
}
{
cout<<"A ia largest among three numbers\n"; cout"A= "<<a;
cout<<"C ia largest among three numbers\n"; cout<<"c= "<<c;
cout<<"B ia largest among three numbers\n"; cout<<"B="<<b;

}

getch();

}

else

{

}

cout<<"C ia largest among three numbers\n"; cout<<"c="<<c;

return 0;

}
```

4. if else ladder

```
if(condition1)

statement1; else if(condition2)

statement 2; else if(condition3)

statement n;

else

default statement.

statement-x;
```

The nesting of if-else depends upon the conditions with which we have to deal.

The condition is evaluated from top to bottom.if a condition is true the statement associated with it is executed.

When all the conditions become false then final else part containing default statements will be executed.

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

{

int per; cout<<”Enter

percentage”; cin>>per; if(per>=80)

cout<<”Secured

Distinction”<<endl; else if(per>=60) cout<<”Secured First

Division”<<endl; else if(per>=50)

cout<<”Secured Second Division”<<endl; else if(per>=40)

cout<<”Secured Third Division”<<endl;

else cout<<”Fail”<<endl

}
```

THE SWITCH STATEMENT or MULTIWAY SELECTION :

In addition to two-way selection, most programming languages provide another selection concept known as multiway

selection. Multiway selection chooses among several alternatives. C has two different ways to implement multiway selection: the switch statement and else-if construct

If for suppose we have more than one valid choices to choose from then we can use switch statement in place of if statements.

```
switch(expression)

{.

case value-1: case value-2:

block-1 break;

block-2 break;
default:

}

default block;

/*program to simulate a simple calculator */

#include<iostream.h> int main()

{

float a,b; char opr;

cout<<"Enter number1 operator number2

cin>>a>>oper>>b;

switch(opr)

{

case '+':

cout<<"Sum : "<<(a + b)<<endl; break;

case '-': cout<<"Difference : "<<(a -b)<<endl; break;

case '*': cout<<”Product : "<<a * b<<endl; break;

case '/': cout<<”Quotient :"<<(a / b)<<endl; break;

default: cout<<”Invalid Operation!"<<endl;

}

return 0;

}
```

Loop control statements or repetitions:

A block or group of statements executed repeatedly until some condition is satisfied is called Loop.

The group of statements enclosed within curly brace is called block or compound statement. We have two types of looping structures.

One in which condition is tested before entering the statement block called entry control. The other in which condition is checked at exit called exit controlled loop.

Loop statements can be divided into three categories as given below 1.while loop statement

2.do while loop statement 3.for loop statement

1. WHILE STATEMENT :

While(test condition)

{

body of the loop

}

It is an entry controlled loop. The condition is evaluated and if it is true then body of loop is executed. After execution of body the condition is once again evaluated and if is true body is executed once again. This goes on until test condition becomes false.

c program to find sum of n natural numbers */ #include<iostream.h>

int main()

{

int i = 1,sum = 0,n;

cout<<"Enter N"<<end; cin>>n;

while(i<=n)

{

sum = sum + i; i = i + 1;

```
}

cout<<”Sum of first”<<n<”natural numbers is:”<<sum<<endl; return 0;

}
```

2. DO WHILE STATEMENT :

The while loop does not allow body to be executed if test condition is false. The do while is an exit controlled loop and its body is executed at least once.

```
do

{

body

}while(test condition);
```

```
/*c program to find sum of n natural numbers */ #include<stdio.h>

int main()

{

int i = 1,sum = 0,n;

cout<<”Enter N"<<endl; cin>>n

do{

sum = sum + i; i = i + 1;

} while(i<=n);

cout<<”Sum of first”<< n<<” natural numbers is:”<<sum; return 0;

}
```

Note: if test condition is false. before the loop is being executed then While loop executes zero number of times where as do--while executes one time

3. FOR LOOP : It is also an entry control loop that provides a more concise structure

Syntax:

```
for(initialization; test expression; increment/decrement)

{

statements;

}
```

For statement is divided into three expressions each is separated by semi colon;

1. initilization expression is used to initialize variables 2.test expression is responsible of continuing the loop. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it .If test expression is false loop terminates 3.increment/decrement expression consists of increment or decrement operator This process continues until test condition satisfies.

```
/*c program to find sum of n natural numbers */

#include<stdio.h>

int main()

{

int i ,sum = 0,n;

cout<<”Enter N"; cin>>n;

for(i=1;i<=n;i++)

{

sum = sum + i;

}

cout<<“Sum of first”<<n<<” natural numbers is:%d”<<sum; return 0;

}
```

Nested loops:Writing one loop control statement within another loop control statement is called nested loop statement

Ex:

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

```
/*program to print prime numbers upto a given number*/ #include<stdio.h>

#include<conio.h> void main()

{

int n,i,fact,j; clrscr();

cout<<"enter the number:"; cin>>n

for(i=1;i<=n;i++)

{fact=0;

//THIS LOOP WILL CHECK A NO TO BE PRIME NO. OR

NOT. for(j=1;j<=i;j++)

{

if(i%j==0) fact++;

}

if(fact==2)

cout<<i<<"\t";

}

getch( );

}
```

Output:

Enter the number : 5

2 3 5

Unconditional control statements:

Statements that transfers control from on part of the program to another part unconditionally Different unconditional statements are

1) goto 2)break 3)continue

1. goto :- goto statement is used for unconditional branching or transfer of the program execution to the labeled statement.

/*c program to find sum of n natural numbers */ #include<stdio.h>

int main()

{

int i ,sum = 0,n;

cout<<”Enter N"; cin>>n;

i=1; L1:

sum = sum + i;

i++;

if(i<=n) goto L1;

cout<<“Sum of first “<<n<” natural numbers is”<<sum; return 0;

}

break:-when a break statement is encountered within a loop ,loop is immediately exited and the program continues with the statements immediately following loop

/*c program to find sum of n natural numbers */ #include<stdio.h>

int main()

```
{

int i ,sum = 0,n;

cout<<”Enter N"; cin>>n;

i=1; L1:

sum = sum + i; i++;

if(i>n) break; goto L1;

cout<<”Sum of first”<<n<<”natural numbers is: ”<<sum; return 0;

}
```

Continue:It is used to continue the iteration of the loop statement by skipping the statements after continue statement. It causes the control to go directly to the test condition and then to continue the loop.

```
/*c program to find sum of n positive numbers read from keyboard*/
#include<stdio.h>

int main()

{

int i ,sum = 0,n,number; cout<<Enter N"; cin>>n; for(i=1;i<=n;i++)

{

cout<<“Enter a number:”; cin>>number; if(number<0) continue; sum = sum +
number;

}

cout<<“Sum of”<<n<<” numbers is:”<<sum; return 0;

}
```

A function is a set of statements that take inputs, do some specific computation, and produce output. The idea is to put some commonly or repeatedly done tasks together and make a **function** so that instead of writing the same code again and again for different inputs, we can call the function. In simple terms, a function is a block of code that only runs when it is called.

Syntax:

Return type function_name(parameters)

Scope Resolution operator:

Scope:-Visibility or availability of a variable in a program is called as scope. There are two

types of scope. i)Local scope ii)Global scope

Local scope: visibility of a variable is local to the function in which it is declared.

Global scope: visibility of a variable to all functions of a program

Scope resolution operator in "::" .

This is used to access global variables if same variables are declared as local and global

```
#include<iostream.h>

int a=5;

void main()

{

int a=1;
cout<<"Local a="<<a<<endl;

cout<<"Global a="<<::a<<endl;

}
```

Objects as Function Arguments: Objects can be used as arguments to

functions This can be done in three ways

a. Pass-by-value or call by value

b. Pass-by-address or call by address

c. Pass-by-reference or call by reference

a.Pass-by-value – A copy of object (actual object) is sent to function and assigned to the object of called

function (formal object). Both actual and formal copies of objects are stored at different memory

locations. Hence, changes made in formal object are not reflected to actual object. write a program to

swap values of two objects

write a program to swap values of two objects

```
#include<iostream.h>

using namespace std;

class sample2;

class sample1

{
```

```
int a;

public:


void getdata(int x);

friend void display(sample1 x,sample2 y);

friend void swap(sample1 x,sample2 y);

};

void sample1::getdata(int x)

{

a=x;

}

class sample2

{

int b;

public:

void getdata(int x);

friend void display(sample1 x,sample2 y);

friend void swap(sample1 x,sample2 y);

};

void sample2::getdata(int x)
```

```
{

b=x;

}

void display(sample1 x,sample2 y)

{

cout<<"Data in object 1 is"<<endl;

cout<<"a="<<x.a<<endl;

cout<<"Data in object 2 is"<<endl;

cout<<"b="<<y.b<<endl;

}

void swap(sample1 x,sample2 y)

{

int t;

t=x.a;

x.a=y.b;

y.b=t;

}

int main()

{

sample1 obj1;
sample2 obj2;

obj1.getdata(5);

obj2.getdata(15);

cout<<"Before Swap of data between Two objects\n

"; display(obj1,obj2);

swap(obj1,obj2);

cout<<"after Swap of data between Two objects\n ";

display(obj1,obj2);

}
```

```
Before Swap of data between Two objects

Data in object 1 is a=5

Data in object 2 is b=15

after Swap of data between Two objects

Data in object 1 is a=5

Data in object 2 is b=15
```

b. Pass-by-address: Address of the object is sent as argument to function.

Here ampersand(&) is used as address operator and arrow (->) is used as de referencing

operator. If any change made to formal arguments then there is a change to actual arguments write a program to swap values of two objects

```
#include<iostream.h>


using namespace std;

class sample2;

class sample1

{

int a;

public:

void getdata(int x);

friend void display(sample1 x,sample2 y);

friend void swap(sample1 *x,sample2 *y);

};

void sample1::getdata(int x)

{

a=x;

}

class sample2

{

int b;
```

```
public:

void getdata(int x);

friend void display(sample1 x,sample2 y);

friend void swap(sample1 *x,sample2 *y);

};

void sample2::getdata(int x)

{

b=x;

}

void display(sample1 x,sample2 y)

{

cout<<"Data in object 1 is"<<endl;

cout<<"a="<<x.a<<endl;

cout<<"Data in object 2 is"<<endl;

cout<<"b="<<y.b<<endl;

}

void swap(sample1 *x,sample2 *y)

{

int t;

t=x->a;
```

```
x->a=y->b;

y->b=t;

}

int main()

{

sample1 obj1;

sample2 obj2;

obj1.getdata(5);

obj2.getdata(15);

cout<<"Before Swap of data between Two objects\n ";

display(obj1,obj2);

swap(&obj1,&obj2);

cout<<"after Swap of data between Two objects\n ";

display(obj1,obj2);

}
```

```
Before Swap of data between Two objects

Data in object 1 is a=5

Data in object 2 is b=15

after Swap of data between Two objects

Data in object 1 is a=15
```

Datatype & reference variable =variable;

Data in object 2 is b=5

c.Pass-by-reference:A reference of object is sent as argument to function.

Reference to a variable provides alternate name for previously defined variable. If any change made to

reference variable then there is a change to original variable.

A reference variable can be declared as follows

Ex:

```
int x=5;

int &y=x;
```

Write a program to find sum of n natural numbers using reference variable

```
#include<iostream.h>

using namespace std;

int main()

{
```

```
int i=0;

int &j=i;

int s=0;

int n;

cout<<"Enter n:";

cin>>n;

while(j<=n)

{

s=s+i;

i++;

}

cout<<"sum="<<s<<endl;

}
```

Output:

```
Enter n:10

sum=55
```

write a program to swap values of two objects

```
#include<iostream.h>

using namespace std;

class sample2;
class sample1
{
int a;
public:
void getdata(int x);
friend void display(sample1 x,sample2 y);
friend void swap(sample1 &x,sample2 &y);
};

void sample1::getdata(int x)
{
a=x;
}
class sample2
{
int b;
public:
void getdata(int x);
friend void display(sample1 x,sample2 y);
friend void swap(sample1 &x,sample2 &y);
};

void sample2::getdata(int x)

{

b=x;

}

void display(sample1 x,sample2 y)

{

cout<<"Data in object 1 is"<<endl;

cout<<"a="<<x.a<<endl;

cout<<"Data in object 2 is"<<endl;

cout<<"b="<<y.b<<endl;

}

void swap(sample1 &x,sample2 &y)

{

int t;

t=x.a;

x.a=y.b;

y.b=t;

}

int main()
```

```
{
sample1 obj1;
sample2 obj2;
obj1.getdata(5);
obj2.getdata(15);
cout<<"Before Swap of data between Two objects\n ";
display(obj1,obj2);
swap(obj1,obj2);
cout<<"after Swap of data between Two objects\n ";
display(obj1,obj2);
}
```

Output:

```
Before Swap of data between Two objects
Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
Data in object 1 is a=15
Data in object 2 is b=5
```

INLINE FUNCTIONS:

Definition:

An inline function is a function that is expanded in line when it is invoked. Inline expansion

makes a program run faster because the overhead of a function call and return is eliminated. It

is defined by using key word “inline”

Necessity of Inline Function:

➢

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable

when a function is likely to be called many times.

➢ Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as

jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling

function.

➢

When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and

therefore, the usual error checking does not occur during compilation.

➢

C++ has different solution to this problem. To eliminate the cost of calls to small functions, C++

proposes a new feature called inline function.

General Form:

```
inline function-header

{

function body;


}
```

Eg:

```
#include<iostream.h>

inline float mul(float x, float y)

{

return (x*y);

}

inline double div(double p, double q)
```

```
{
return (p/q);
}
int main()
{
float a=12.345;
float b=9.82;
cout<<mul(a,b);
cout<<div(a,b);
return 0;
}
```

Properties of inline function:

1.Inline function sends request but not a command to compiler

2.Compiler my serve or ignore the request

3.if function has too many lines of code or if it has complicated logic then it is executed as

normal function

Situations where inline does not work:

➢

A function that is returning value , if it contains switch ,loop or both then it is treated as

normal function.

➢

if a function is not returning any value and it contains a return statement then it is treated as normal function

➢

If function contains static variables then it is executed as normal function

➢

If the inline function is declared as recursive function then it is executed as normal function.

DEFAULT ARGUMENTS

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

1) The following is a simple C++ example to demonstrate the use of default arguments. Here, we don't have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

- CPP

```
// CPP Program to demonstrate Default Arguments

#include <iostream>
```

```
using namespace std;



// A function with default arguments,

// it can be called with

// 2 arguments or 3 arguments or 4 arguments.

int sum(int x, int y, int z = 0, int w = 0) //assigning default
values to z,w as 0

{

    return (x + y + z + w);

}



// Driver Code

int main()

{

    // Statement 1

    cout << sum(10, 15) << endl;



    // Statement 2

    cout << sum(10, 15, 25) << endl;



    // Statement 3
    cout << sum(10, 15, 25, 30) << endl;

    return 0;

}
```

Output

```
25

50

80
```

Explanation: In statement 1, only two values are passed, hence the variables z and w take the default values of 0. In statement 2, three values are passed, so the value of z is overridden with 25. In statement 3, four values are passed, so the values of z and w are overridden with 25 and 30 respectively.

RECURSIVE FUNCTION

A **recursive** function is a function that makes calls to itself. It works like the loops we described before, but sometimes it the situation is better to use recursion than loops.

Every recursive function has two components: a **base case** and a **recursive step**. The **base case** is usually the smallest input and has an easily verifiable solution. This is also the mechanism that stops the function from calling itself forever. The **recursive step** is the set of all cases where a **recursive call**, or a function call to itself, is made.

As an example, we show how recursion can be used to define and compute the factorial of an integer number. The factorial of an integer nn is 1×2×3×...×(n−1)×n1×2×3×...×(n−1)×n. The recursive definition can be written:

f(n)={1n×f(n−1)if n=1otherwisef(n)={1if n=1n×f(n−1)otherwise

The base case is n=1n=1 which is trivial to compute: f(1)=1f(1)=1. In the recursive step, nn is multiplied by the result of a recursive call to the factorial of n−1n−1.

TRY IT! Write the factorial function using recursion. Use your function to compute the factorial of 3.

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

```
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

DYNAMIC MEMORY ALLOCATION

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

- **The stack** − All variables declared inside the function will take up memory from the stack.
- **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

new and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements −

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the 'delete' operator as follows −

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how 'new' and 'delete' work −

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

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable
   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Value of pvalue : 29495
```

UNIT -II

C++ Classes and Data Abstraction
Introduction of Classes,Class Definition, class structure , class objects, class scope, This pointer, Friends to a class, Static class members, Constant member functions, Constructors , and destructors, Dynamic creation and destruction of objects, Data Abstraction, ADT and information Hiding.

Introduction of Class:

An object-oriented programming approach is a collection of objects and each object consists of corresponding data structures and procedures. The program is reusable and more maintainable.
The important aspect in oop is a **class** which has similar syntax that of structure.

class: It is a collection of data and member functions that manipulate data. The data components of class are called data members and functions that manipulate the data are called member functions.

It can also called as blue print or prototype that defines the variables and functions common to all objects of certain kind. It is also known as user defined data type or ADT(abstract data type) A class is declared by the keyword **class**.

Syntax:-

```
class class_name
{

Access specifier :
                    Variable declarations;

Access specifier :
                  function declarations;


};
```

Access Specifiers:

Access specifier or access modifiers are the labels that specify type of access given to members of a class. These are used for data hiding. These are also called as visibility modes. There are three types of access specifiers

1.private
2.public
3.protected

1.Private:

If the data members are declared as private access, then they cannot be accessed from other functions outside the class. It can only be accessed by the functions declared within the class. It is declared by the key word "**private**".

2.public:

If the data members are declared public access, then they can be accessed from other functions outside the class. It is declared by the key word "**public**".

3.protected: The access level of protected declaration lies between public and private. This access specifier is used at the time of inheritance

Note: - If no access specifier is specified then it is treated by default as private. The default access specifier of structure is public where as that of a class is “private”

Example:

```
class student
{
private : int roll;
          char name[30];
public:
          void get_data()
          {
                    cout<<”Enter roll number and name”:
                    cin>>roll>>name;
          }
          void put_data()
          {
                    cout<<”Roll number:”<<roll<<endl;
                    cout<<”Name          :”<<name<<endl;
          }
};
```

Object:- Instance of a class is called object.

Syntax:

class_name object_name;

Ex:student s;

Accessing members: - dot operator is used to access members of class

Object-name.function-name(actual arguments);

Ex:

```
s.get_data();
s.put_data();
```

Note:

1. If the access specifier is not specified in the class the default access specifier is private
2. All member functions are to be declared as public if not they are not accessible outside the class.

Object: Instance of a class is called as object.
Syntax:
Class_name object name;

Example: student s;

in the above example s is the object. It is a real time entity that can be used

Write a program to read data of a student

```
#include<iostream>
using namespace std;
class student
{
private:
          int roll;
          char name[20];


public:
          void getdata()
```

```
            {
            cout<<"Enter Roll number:";
                    cin>>roll;
                    cout<<"Enter Name:";
                    cin>>name;
            }
            void putdata()
            {
                  cout<<"Roll no:"<<roll<<endl;
cout<<Name:"<<name<<endl;
            }
            };
            int main()
            {
            student s;
                    s.getdata();
                    s.putdata();
                    returm 0;
            }
```

Scope Resolution operator:

Scope:-Visibility or availability of a variable in a program is called as scope. There are two types of scope. i)Local scope ii)Global scope

Local scope: visibility of a variable is local to the function in which it is declared.

Global scope: visibility of a variable to all functions of a program

Scope resolution operator in **"::" .**

This is used to access global variables if same variables are declared as local and global

```
        #include<iostream.h>
        int a=5;
        void main()
        {
        int a=1;
                cout<<"Local a="<<a<<endl;
                cout<<"Global a="<<::a<<endl;
        }
```

Class Scope:

Scope resolution operator(::) is used to define a function outside a class.

```
#include  <iostream>
using namespace std;
class sample
 {
public:
  void output(); //function declaration
};
 // function definition outside the
 class void sample::output()
 {
  cout << "Function defined outside the class.\n";

};
```

```
 int main()
 {
 sample   obj;
 obj.output();
 return 0;
 }
```

Output of program:

```
Function defined outside the class.
```

Write a program to find area of rectangle

```
#include<iostream.h>
class rectangle
{
int L,B;
public:
        void get_data();
        void area();
};

void rectangle::get_data()
{
        cout<<”Enter Length of rectangle”;
        cin>>L;
        cout<<”Enter breadth of rectangle”;
        cin>>B;
}
int rectangle::area()
{
        return L*B;
}
int main()
{
rectangle r;
        r.get_data();
        cout<<”Area of rectangle is”<<r.area();
return 0;
}
```

FRIEND FUNCTIONS: The private members cannot be accessed from outside the class. i.e., a nonmember function cannot have an access to the private data of a class. In C++ a nonmember function can access private by making the function friendly to a class.

Definition: A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator for defining. It can access private members of a class. It is declared by using keyword “friend”

Ex:

```
 class sample
{
int x,y;
public:
        sample(int a,int b);
friend int sum(sample s);
};
sample::sample(int a,int b)
{
x=a;y=b;
}
int sum(samples s)
{
int sum;
        sum=s.x+s.y;
        return 0;
}
void main()
{
Sample obj(2,3);
int res=sum(obj);
cout<< “sum=”<<res<<endl;
}
```

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a normal
function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot
membership operator with each member name.
- It can be declared either in the public or private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

```
#include<iostream.h>
class sample
{
int a;
int b;
public:
        void setvalue()
        {
        a=25;
        b=40;
        }
        friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a+s.b)/2.0;
}
int main()
{
sample X;
        X.setvalue();
        cout<<”Mean value=”<<mean(X);
        return 0;
}
```

write a program to find max of two numbers using friend function for two different classes

```
#include<iostream> using
namespace std;
class sample2;
class sample1
{
int x;
public:
        sample1(int a);
        friend void max(sample1 s1,sample2 s2)
};
sample1::sample1(int a)
{
x=a;
}
class sample2
{
int y;
public:
        sample2(int b);
        friend void max(sample1 s1,sample2 s2)
};
Sample2::sample2(int b)
{
        y=b;
}
void max(sample1 s1,sample2 s2)
{
If(s1.x>s2.y)
        cout<<"Data member in Object of class sample1 is larger "<<endl;
else
        cout<<"Data member in Object of class sample2 is larger "<<endl;
}

void main()
{
sample1 obj1(3);
sample2 obj2(5);
max(obj1,obj2);
}
```

Write a program to add complex numbers using friend function

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

```
{
float real,img;
public:
        complex();
        complex(float x,float y)
        friend complex add_complex(complex c1,complex c2);
};
complex::complex()
{
        real=img=0;
}
complex::complex(float x,float y)
{
        real=x;img=y;
}
complex add_complex(complex c1,complex c2)
{
complex t;
        t.real=c1.real+c2.real;
        t.img=c1.img+c2.img;
        return t;
}
void complex::display ()
{
        if(img<0)
        {img=-img;
                cout<<real<<"-i"<<img<<endl
        }
        else
        {
                cout<<real<<"+i"<<img<<endl
        }
}

int main()
{
complex obj1(2,3);
complex obj2(-4,-6);
        complex obj3=add_compex(obj1,obj2);
        obj3.display();
        return 0;
}
```

Friend Class: A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

Example:

```
#include <iostream.h>
 class sample_1
{
```

public:

```
friend class sample_2;//declaring friend class int a,b;

void getdata_1()
{
                        cout<<"Enter A & B values in class sample_1";
                        cin>>a>>b;
void display_1()
              {
                        cout<<"A="<<a<<endl;
                        cout<<"B="<<b<<endl;
              }
    };
    class sample_2
    {


public:

int c,d,sum; sample_1
obj1;

void getdata_2()
{
                        obj1.getdata_1();
                        cout<<"Enter C & D values in class sample_2";
                        cin>>c>>d;
              }
              void sum_2()
              {
                        sum=obj1.a+obj1.b+c+d;
              }

              void display_2()
              {
                        cout<<"A="<<obj1.a<<endl;
                        cout<<"B="<<obj1.b<<endl;
                        cout<<"C="<<c<<endl;
                        cout<<"D="<<d<<endl;
                        cout<<"SUM="<<sum<<endl;
              }
    };
    int main()
    {
    sample_1 s1;
              s1.getdata_1();
              s1.display_1();
    sample_2 s2;
              s2.getdata_2();
              s2.sum_2();

              s2.display_2();

              }
```

STATIC CLASS MEMBERS

Static Data Members
Static Member Functions

Static Data Members:

A data member of a class can be qualified as static. A static member variable has certain special characteristics:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how
- many objects are created.
- Static data member is defined by keyword **„static‟**

Syntax:

Data type class name::static_variable Name;

Ex: int item::count;

```
#include<iostream.h>
#include<conio.h>
class item
{
        static int count;
        int number;
public:
        void getdata(int a)
        {
                number=a;
                count++;
        }
        void getcount()
        {
        cout<<"count is"<<count;
        }
};
int item::count;//decleration
int main()
{
item a,b,c;
        a.getcount();
        b.getcount();
        c.getcount();
        a.getdata(100);
        b.getdata(200);
        c.getdata(300);
        cout<<"After reading data";
        a.getcount();
        b.getcount();
        c.getcount();
        return 0;
}
```

Output:

```
count is 0
count is 0
count is 0
After reading data
```

```
count is 3
count is 3
count is 3
```

Static Member Functions

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

A static function can have access to only other static members (functions or variables) declared in the same class.

A static member function is to be called using the class name (instead of its objects) as follows: **class-name :: function-name;**

```
#include<iostream.h>
class test
{
        int code;
        static int count;
public:
        void setcode()
        {
                code=++count;
        }
        void showcode()
        {
                cout<<"object number"<<code;
        }
        static void showcount()
        {
                cout<<"count"<<count;
        }
};
int test::count;
int main()
{
test t1,t2;
        t1.setcode();
        t2.setcode();
        test::showcount();
test t3;
        t3.setcode();
        test::showcount();
        t1.showcode();
        t2.showcode();
        t3.showcode();
        return 0;
}
```

Output:

```
count 2
count 3
object number 1
object number 2
object number 3
```

Arrays of Objects: Arrays of variables of type "class" is known as "Array of objects". An array of objects is stored inside the memory in the same way as in an ordinary array.

Syntax:

```
class class_name
{

private:
        data_type members;
public:
        data_type members;
        member functions;
    };
```

Array of objects:

Class_name object_name[size];
Where size is the size of array
Ex:
Myclass obj[10];

Write a program to initialize array of objects and print them

```
#include<iostream>
using namespace std;
class MyClass
{
int a;
public:
    void set(int x)
    {
        a=x;
    }
    int get()
    {
        return a;
    }
};
int main()
{
MyClass obj[5];
for(int i=0;i<5;i++)
obj[i].set(i);
for(int i=0;i<5;i++)
 cout<<"obj["<<i<<"].get():"<<obj[i].get()<<endl;
}
```

Output:

```
obj[0].get():0
obj[1].get():1
obj[2].get():2
obj[3].get():3
obj[4].get():4
```

Objects as Function Arguments: Objects can be used as arguments to functions This can be done in three ways

a. Pass-by-value or call by value
b. Pass-by-address or call by address
c. Pass-by-reference or call by reference

a.Pass-by-value – A copy of object (actual object) is sent to function and assigned to the object of called function (formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal object are not reflected to actual object. write a program to swap values of two objects

write a program to swap values of two objects

```
#include<iostream.h>
using  namespace  std;
class sample2;
class sample1
{
int a;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 x,sample2 y);
};
void sample1::getdata(int x)
{
        a=x;
}
class sample2
{
int b;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);

friend void swap(sample1 x,sample2 y);
};
void sample2::getdata(int x)
{
b=x;
}
void display(sample1 x,sample2 y)
{
        cout<<"Data in object 1 is"<<endl;
        cout<<"a="<<x.a<<endl;
        cout<<"Data in object 2 is"<<endl;
        cout<<"b="<<y.b<<endl;
}
void swap(sample1 x,sample2 y)
{
int t;
        t=x.a;
        x.a=y.b;
        y.b=t;
}
int main()
{
sample1 obj1;
sample2 obj2;
        obj1.getdata(5);
        obj2.getdata(15);
        cout<<"Before Swap of data between Two objects\n
        "; display(obj1,obj2);
        swap(obj1,obj2);
        cout<<"after Swap of data between Two objects\n ";
        display(obj1,obj2);
}
```

```
Before Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
```

b. Pass-by-address: Address of the object is sent as argument to function.
Here ampersand(&) is used as address operator and arrow (->) is used as de referencing operator. If any change made to formal arguments then there is a change to actual arguments

write a program to swap values of two objects

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

```
using namespace std;
class sample2;
class sample1
{
int a;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 *x,sample2 *y);

};
void sample1::getdata(int x)
{
a=x;
}
class sample2
{
int b;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 *x,sample2 *y);
};
void sample2::getdata(int x)
{
        b=x;
}
void display(sample1 x,sample2 y)
{
        cout<<"Data in object 1 is"<<endl;
        cout<<"a="<<x.a<<endl;
        cout<<"Data in object 2 is"<<endl;
        cout<<"b="<<y.b<<endl;
}
void swap(sample1 *x,sample2 *y)
{
int t;
        t=x->a;
        x->a=y->b;
        y->b=t;
}
int main()
{
sample1 obj1;
sample2 obj2;
        obj1.getdata(5);
        obj2.getdata(15);
        cout<<"Before Swap of data between Two objects\n ";
        display(obj1,obj2);
        swap(&obj1,&obj2);
        cout<<"after Swap of data between Two objects\n ";
        display(obj1,obj2);
}
```

```
Before Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
 Data in object 1 is a=15
```

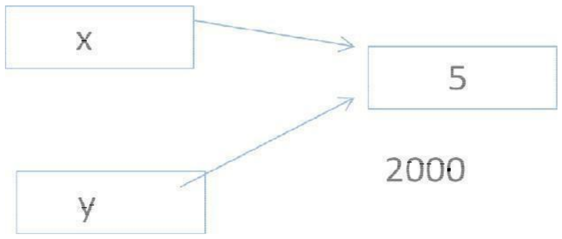
Data in object 2 is b=5

c.Pass-by-reference:A reference of object is sent as argument to function.
Reference to a variable provides alternate name for previously defined variable. If any change made to reference variable then there is a change to original variable.
A reference variable can be declared as follows

Datatype & reference variable =variable;

Ex:

```
	int x=5;
	int &y=x;
```



Write a program to find sum of n natural numbers using reference variable

```
#include<iostream.h>
using namespace std;
int main()
{
int i=0;
int &j=i;
int s=0;
int n;
cout<<"Enter n:";
cin>>n;
while(j<=n)
{
s=s+i;
i++;
}
cout<<"sum="<<s<<endl;
}
```

Output:

```
Enter n:10
sum=55
```

write a program to swap values of two objects

```
#include<iostream.h>
using  namespace  std;
class sample2;
class sample1
{
int a;
public:
	void getdata(int x);
	friend void display(sample1 x,sample2 y);
	friend void swap(sample1 &x,sample2 &y);
};
```

```
void sample1::getdata(int x)
{
a=x;
}
class sample2
{
int b;
public:
        void getdata(int x);
        friend void display(sample1 x,sample2 y);
        friend void swap(sample1 &x,sample2 &y);
};
void sample2::getdata(int x)
{
b=x;
}
void display(sample1 x,sample2 y)
{
        cout<<"Data in object 1 is"<<endl;
        cout<<"a="<<x.a<<endl;
        cout<<"Data in object 2 is"<<endl;
        cout<<"b="<<y.b<<endl;
}
void swap(sample1 &x,sample2 &y)
{
int t;
        t=x.a;
        x.a=y.b;
        y.b=t;
}
int main()
{
sample1 obj1;
sample2 obj2;
        obj1.getdata(5);
        obj2.getdata(15);
        cout<<"Before Swap of data between Two objects\n ";
        display(obj1,obj2);
        swap(obj1,obj2);
        cout<<"after Swap of data between Two objects\n ";
        display(obj1,obj2);
}
```

Output:

```
Before Swap of data between Two objects
 Data in object 1 is a=5
Data in object 2 is b=15
after Swap of data between Two objects
 Data in object 1 is a=15
Data in object 2 is b=5
```

- **Introduction to Constructors**: C++ provides a special member function called the constructor which enables an object to initialize itself when it is created.

Definition:- A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same name as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class. A constructor is declared and defined as follows:

```
class integer
{
        int m,n;
        public:
        integer( );
        ………..
        ………..
};
integer :: integer( )
{
        m=0;
        n=0;
}

int main()
{   integer obj1;
                ………..
                ………..
}
```

integer obj1; => not only creates object obj1 but also initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor function.

CHARACTERISTICS OF CONSTRUCTOR

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return type, not even void.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other c++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- They make „implicit calls" to the operators new and delete when memory allocation is required.

Constructors are of 3 types:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

1.Default Constructor: A constructor that accepts no parameters is called the **default constructor**.

```
#include<iostream.h>

#include<conio.h>
class item
{
	int m,n;
	public:
	item()
	{
		m=10;
		n=20;
	}
	void put();
};
void item::put()
{
	cout<<m<<n;
}
void main()
{
	item t;
	t.put();
	getch();
}
```

1.Parameterized Constructors:-The constructors that take parameters are called parameterized constructors.

```
#include<iostream.h>
class item
{
int m,n;
public:
	item(int x, int y)
	{
	m=x;
	n=y;
	}
};
```

When a constructor has been parameterized, the object declaration statement such as item t; may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in 2 ways: item t=item(10,20); //explicit call

item t(10,20); //implicit call

Eg:

```
#include<iostream.h>
#include<conio.h>
class item
{
int m,n;
public:
        item(int x,int y)
        {
                m=x;
                n=y;
        }
        void put();
};
void item::put()
{
        cout<<m<<n;
}
void main()
{
item t1(10,20);
item t2=item(20,30);
        t1.put();
        t2.put();
        getch();
}
```

2.Copy Constructor: A copy constructor is used to declare and initialize an object from another object.

Eg:

```
        item t2(t1);
        or
        item t2=t1;
```

1. The process of initializing through a copy constructor is known as copy initialization.
2. t2=t1 will not invoke copy constructor. t1 and t2 are objects, assigns the values of t1 to t2.
3. A copy constructor takes a reference to an object of the same class as itself as an argument. #include<iostream.h>

```
    class sample
    {
    int n;
    public:
        sample()
        {
        n=0;
        }
        sample(int a)
        {
            n=a;
        }
        sample(sample &x)
        {
            n=x.n;
        }
        void display()
        {
            cout<<n;
        }
    };
    void main()
    {
    sample A(100);
    sample B(A);
    sample C=A;
    sample D;
        D=A;
        A.display();
        B.display();
        C.display();
        D.display();
    }
```

Output: 100 100 100 100

Multiple Constructors in a Class: Multiple constructors can be declared in a class. There can be any number of constructors in a class.

```
class complex
{
float real,img;
public:
    complex()//default constructor
    {
    real=img=0;
    }
    complex(float r)//single parameter parameterized constructor
    {
    real=img=r;
    }
    complex(float r,float i) //two parameter parameterized constructor
    {
    real=r;img=i;
    }
    complex(complex&c)//copy constructor
    {
    real=c.real;
    img=c.img;

    }
    complex sum(complex c )
    {
    complex t;
    t.real=real+c.real;
    t.img=img+c.img;
    return t;
    }
    void show()
    {
    If(img>0)
            cout<<real<<"+i"<<img<<endl;
    else
            {
            img=-img;
            cout<<real<<"-i"<<img<<endl;
            }
    }
};

void main()
{
complex c1(1,2);
complex c2(2,2);
compex c3;
c3=c1.sum(c3);
c3.show();

}
```

DESTRUCTORS:A destructor, is used to destroy the objects that have been created by a constructor.

Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded **by a tilde.**

Eg: ~item() { }

1. A destructor never takes any argument nor does it return any value.
2. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.
3. It is a good practice to declare destructors in a program since it releases memory space for future use.

```
#include<iostream>
using namespace std;
class Marks
{
public:
  int maths;
  int science;

  //constructor
  Marks() {
    cout << "Inside Constructor"<<endl;
    cout << "C++ Object created"<<endl;
  }
```

```
//Destructor
```

```
  ~Marks() {
    cout << "Inside Destructor"<<endl;
    cout << "C++ Object destructed"<<endl;
  }
};

int main( )
{
  M
  ar
  k
  s
  m
  1;
  M
  ar
  k
  s
  m
  2;
  re
  tu
  r
  n
  0;
}
```

Output:

```
        Inside
        Construct
        or C++
        Object
        created
        Inside
        Construct
        or C++
        Object
        created
        Inside
        Destructor
        C++ Object
```

```
destructed
Inside
Destructor
C++ Object destructed
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OOPS THROUGH C++ NOTES

INHERITANCE AND POLYMORPHISM

Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object-Oriented Programming.

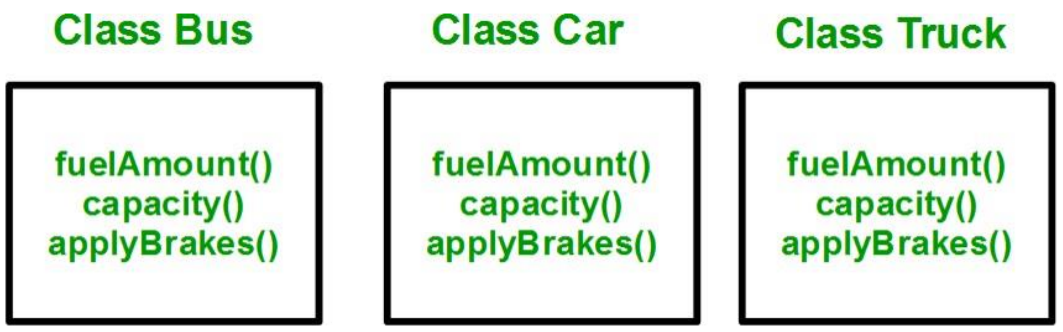
SubClass: The class that inherits properties from another class is called Subclass or Derived class.

Super Class: The class whose properties are inherited by sub class is called Base Class or Superclass.

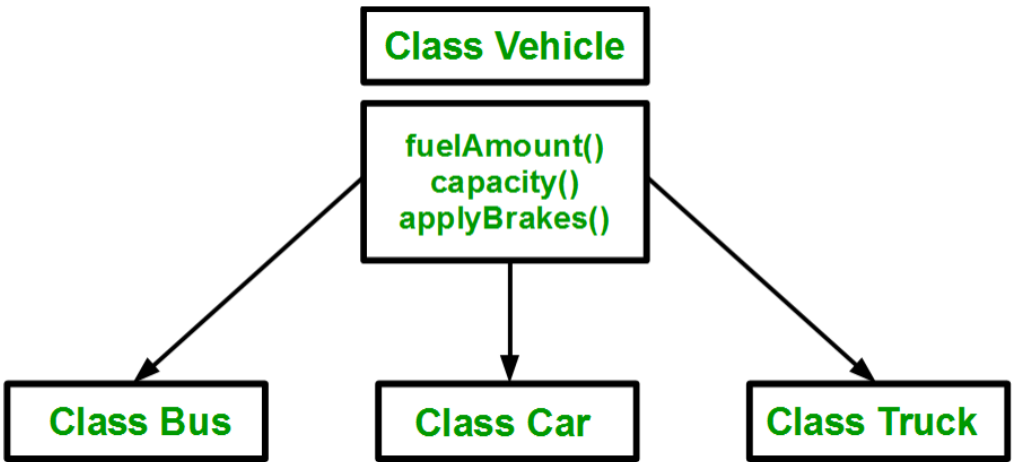
1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuel Amount(), capacity(), apply Brakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class.



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Implementing inheritance in C++: For creating a sub-class which is inherited from the base class. We have to follow the below syntax.

Syntax:

```
Class sub  class_name :access_mode base_class_name

{

  //bodyofsubclass

};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which youwant to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note:A derived class doesn’t inherit ***access*** to private data members .However ,It does inherit a full parent object ,which contains any private members which that class declares.

```
//C++program to demonstrate implementation
//of Inheritance

#include
<iostream>using
namespace std;

//Base class
class Parent
{
   Public
     :int
     id_p;
};

// Sub class inheriting from Base
Class(Parent)classChild : publicParent
{
    Public
      int id_c;
};

//main function
int main()
   {
        Child obj1;

        //An object of class child has all data members
        // and member functions of class parent
        obj1.id_c=7;
        obj1.id_p =91;
        cout<< "Child id is " <<obj1.id_c
        <<endl;cout<<"Parent id
        is"<<obj1.id_p<<endl;
        return0;
   }
```

Output:

```
Child id is 7

Parent id is 91
```

In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class'Child'.

Modes of Inheritance

1.**Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

2.**Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

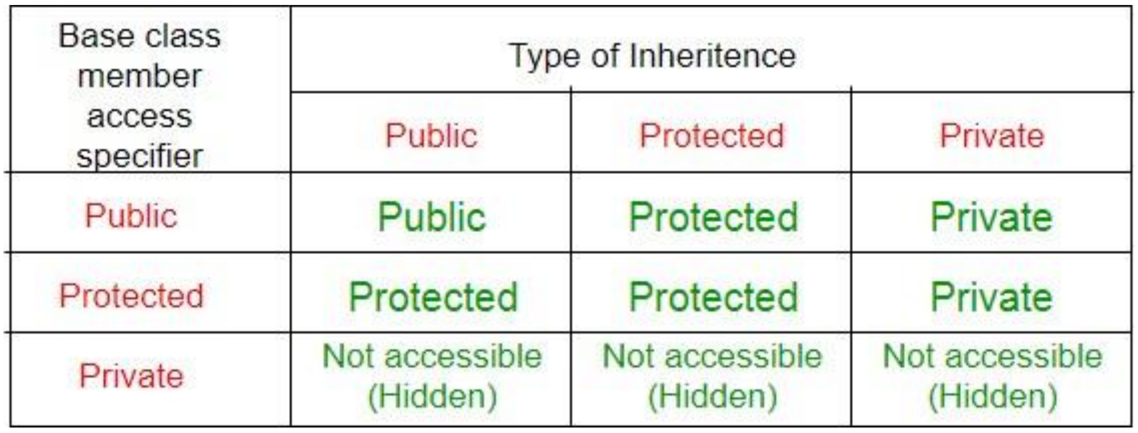
3.**Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B ,C and D all contain the variables x, y and z in below example .It is just question of access.

```
//C++Implementation to show that a derived class
//doesn't inherit access to private data members.
// However, it does inherit a full parent object
classA
{
public:
      int
 x;protecte
   d:inty;
private:
     intz;
};
```

```
classB : publicA
{
   // x is public
   //y is protected
   //z Is not accessible from B
};

class  C:protected A
{
   //x is protected
   //y is protected
   //z is not accessible from C
};

class D :private A     // 'private' is default for classes
{
   //x is private
   //y is private
   //z Is not accessible from D
};
```

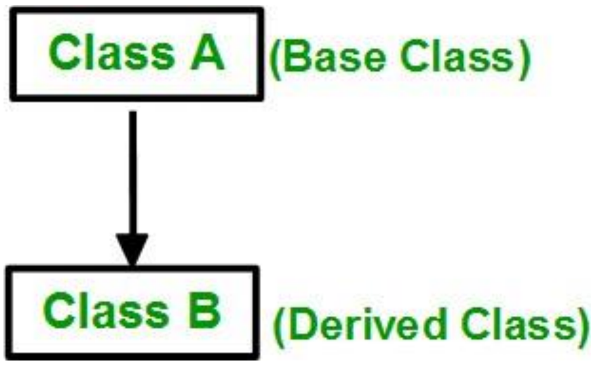
The below tables summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public ,protected and private modes:



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

1.**Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e.one sub class is inherited by one base class only.



Syntax:

```
Class sub class_name:access_mode base_class

{

//body of sub class

};
```

//C++program toexplain

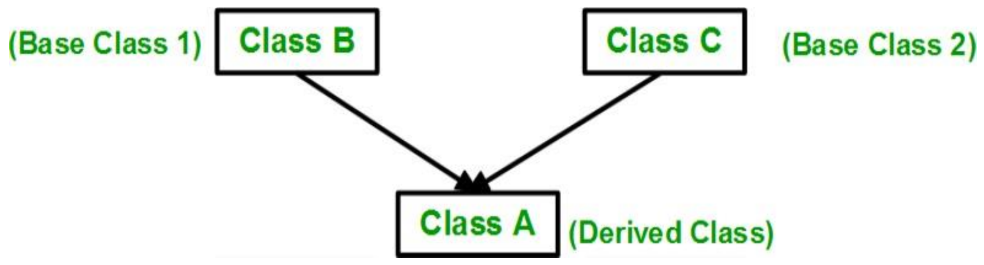
// Single inheritance

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

Output:

```
Salary:60000
Bonus:5000
```

2.Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes i.e., one **sub class** is inherited from more than one **baseclass.**



Syntax:

```
classsubclass_name: access_modebase_class1,access_modebase_class2,....
{
  //body of subclass
};
```

Here, the number of base classes will be separated by a comma(‘,‘) and access mode for every base class must be specified.

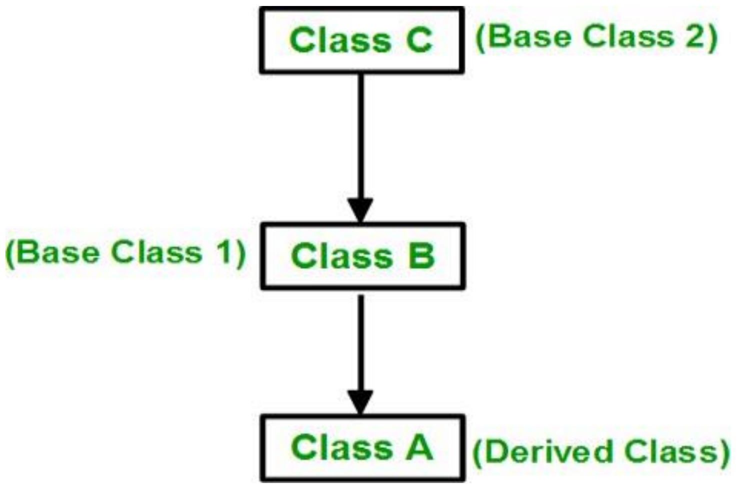
//C++program to explain multiple inheritance

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
        a = n;
    }
};

class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();

    return 0;
}
```

Output:

```
The value of a is:10
The value of b is :20
Addition of a and b is:30
```

3.**Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class. As shown in below diagram, class A has class B and class C as parent classes. Depending on the relation the level of inheritance can be extended to any level.

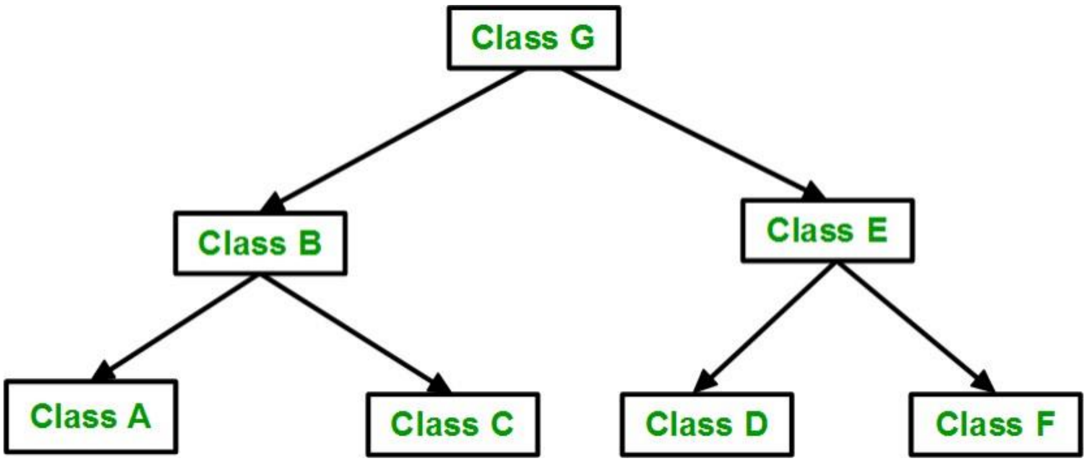


```
            //C++programtoimplementMultilevel Inheritance
#include <iostream>
using namespace std;
class base //single base class
{
            public:
            int x;
            void getdata()
            {
            cout<< "Enter value of x= "; cin>> x;
            }
};
class derive1 : public base // derived class from base class
{
            public:
            int y;
            void readdata()
            {
            cout<< "\nEnter value of y= "; cin>> y;
            }
};
class derive2 : public derive1   // derived from class derive1
{
            private:
            int z;
            public:
            void indata()
            {
            cout<< "\nEnter value of z= "; cin>> z;
            }
            void product()
            {
            cout<< "\nProduct= " << x * y * z;
            }
};
int main()
{
    derive2 a;      //object of derived class
a.get data();
a.read data();
a.in data();
a.product();
    return 0;
}                     //end of program
```

Output

```
Enter value of x= 2

Enter value of y= 3

Enter value of z= 3

Product= 18
```

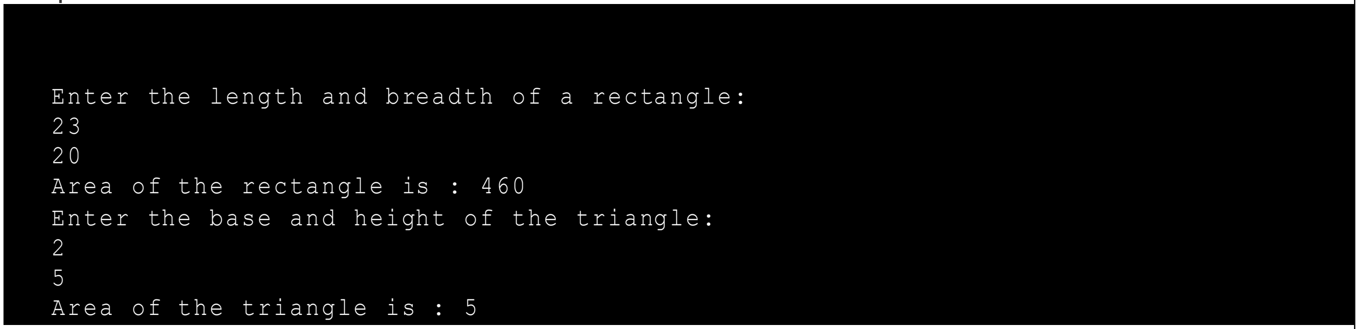
4.**Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e., more than one derived class is created from a single base class.



//C++program to implement Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class Shape                  // Declaration of base class.
{
    public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};
class Rectangle : public Shape  // inheriting Shape class
{
    public:
    int rect_area()
    {
        int result = a*b;
        return result;
    }
};
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

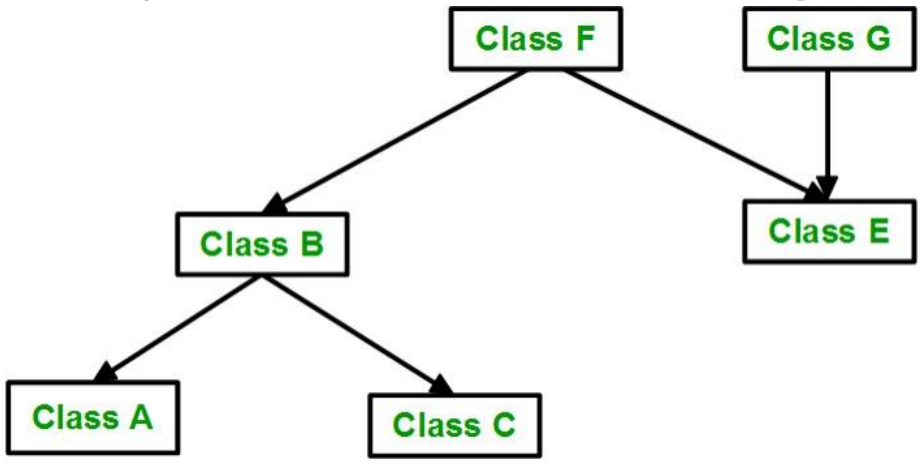
Output:



```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is460Area of the triangle is: 5
```

5.**Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritance:



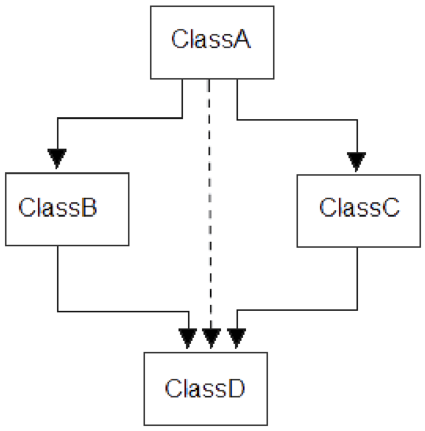
// C++ program for Hybrid Inheritance

```
#include <iostream>
using namespace std;
class A
{
    protected:
    int a;
    public:
    void get_a()
    {
       std::cout << "Enter the value of 'a' : " << std::endl;
       cin>>a;
    }
};

class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
        std::cout << "Enter the value of 'b' : " << std::endl;
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

A special case of hybrid inheritance : Multipath inheritance:
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity canaries in this type of inheritance.



Consider the following program:
//C++program demonstrating ambiguity in Multipath Inheritance

```
#include<iostream.h>
#include<conio.h>cla
ssClassA
       {
              public:
              inta;
       };

       classClassB: publicClassA
       {
              public:
              int b;
       };
       classClassC: publicClassA
       {
              public:
              intc;
       };

       Class ClassD :public ClassB,public ClassC
       {
              public:
              int d;
};

voidmain()
{

      ClassD obj;

      //obj.a =10;                    //Statement1,Error
      //obj.a =100;                  //Statement2,Error

      obj.ClassB::a=10;          //Statement3
      obj.ClassC::a=100;        //Statement4

      obj.b = 20;
      obj.c =30;
      obj.d = 40;

      cout<< "\n A from ClassB:
      "<<obj.ClassB::a;cout<<"\nAfromClassC:"<<o
      bj.ClassC::a;

      cout<<       "\n       B       :
      "<<obj.b;cout<<  "\n  C  :
      "<<obj.c;cout<<"\nD:"<
      <obj.d;

}
```
Output:

```
A from ClassB: 10A
from ClassC: 100B :
20
C : 30
D: 40
```

In the above example, both Class B & Class C inherit Class A, they both have single copy of Class A. However Class D inherit both Class B & Class C, therefore Class D have two copies of Class A, one from Class B and another from Class C.
If we need to access the data member a of Class A through the object of Class D, we must specify the path from which a will be accessed, whether it is from Class B or Class C, bco'z compiler can't differentiate between two copies of Class A in Class D.

There are 2 ways to avoid this ambiguity:

1. **Use scope resolution operator**
2. **Use virtual base class**

Avoiding ambiguity using scope resolution operator:
Using scope resolution operator we can manually specify the path from which data member will be accessed ,as shown in statement 3 and 4, inthe above example.

```
obj.ClassB::a=10;          //Statement3
obj.ClassC::a=100;        //Statement4
```
Note: Still,there are two copies of ClassA in ClassD.

Avoiding ambiguity using virtual baseclass:

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

    class ClassA
    {
          public:
          inta;
    };

    class ClassB : virtual public ClassA
    {
          public:
          int b;
    };
    class ClassC : virtua public ClassA
    {
          public:
          intc;
    };

    class classD : public classB, public classC
    {
          public:
          int d;
    };

    void main()
    {

          classD obj;

          obj.a =10;        //Statement3
          obj.a =100;       //Statement4

          obj.b = 20;
          obj.c =30;
          obj.d = 40;

          cout<< "\n A : "<<obj.a;
          cout<< "\n B : "<<obj.b;
          cout<< "\n C : "<<obj.c;
          cout<<"\nD:"<<obj.d;

    }
```

Output:

```
A: 100
B: 20
C: 30
D: 40
```

According to the above example, ClassD has only one copy of ClassA, t herefore, statement4 will overwrite the value of a, given at statement3.

Constructor and Destructor Execution in Inheritance

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e. the order of invocation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.

The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only. This is why the constructor of **base class is called first to initialize all the inherited members.**

```
#include <iostream>
using namespace estd;
// base class
classParent{
   public:
      // base class constructor
   Parent(){
       cout<< "Inside base class"<<endl;
   }};
// sub class
classChild : publicParent{
   public:
    //sub class constructor
   Child(){
       cout<< "Inside sub class"<<endl;
   }};
// main function
intmain() {
   // creating object of sub class
   Child obj;
   return0;
}
```

Output:

```
Inside base class

Inside sub class
```

Order of constructor call for Multiple Inheritance

For multiple inheritance order of constructor call is, the base class's constructors are called in the order of inheritance and then the derived class's constructor.

```
// C++ program to show the order of constructor calls
// in Multiple Inheritance
#include <iostream>
usingnamespacestd;

// first base class
classParent1
{
    Public:
    // first base class's Constructor
    Parent1()
    {
        cout<< "Inside first base class"<<endl;
    }
};

// second base class
classParent2
{
    public:

    // second base class's Constructor
    Parent2()
    {
        cout<< "Inside second base class"<<endl;
    }
};

// child class inherits Parent1 and Parent2
classChild : publicParent1, publicParent2
{
    public:

    // child class's Constructor
    Child()
    {
        cout<< "Inside child class"<<endl;
    }
};

// main function
intmain() {

    // creating object of class Child
    Child obj1;
    return0;
}
```

Output:

```
Inside first base class

Inside second base class

Inside child class
```

How to call the parameterized constructor of base class in derived class constructor?

To call the parameterized constructor of base class when derived class's parameterized constructor is called, you have to explicitly specify the base class's parameterized constructor in derived class as shown in below program:

```
// C++ program to show how to call parameterised Constructor
// of base class when derived class's Constructor is called

#include <iostream>
usingnamespacestd;

 // base class
 classParent {
     intx;

public:
    // base class's parameterised constructor
    Parent(inti)
    {
        x = i;
        cout<< "Inside base class's parameterised "
                "constructor"
             <<endl;
    }
};

// sub class
classChild : publicParent {
public:
    // sub class's parameterised constructor
    Child(intx):Parent(x)
    {

        cout<< "Inside sub class's parameterised "
                "constructor"
             <<endl;
    }
};

// main function
intmain()
{

    // creating object of class Child
    Child obj1(10);
    return0;
}
```

Output:

```
Inside base class's parameterised constructor

Inside sub class's parameterised constructor
```

Important Points:

- Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
- To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.
- The parameterized constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterized constructor of sub class.

Order of Destructor Calling

Destructors are called *automatically* when an object variable goes out of scope. Because the base class destructor is inherited, and because the derived class object "is" a base class object, both the derived class destructor (even if it is the "default" destructor) and the base class destructor are called automatically. The order in which these are called is bottom-up, and the destructors run to completion before the parent destructor is called. Thus, the destructors *execute in bottom-up order*.

Code Example

```
#include<iostream>
Using namespace std;
class Base
{
public:
Base  ()
    {
std::cout<< "Base class Constructor\n";
    }
    ~Base ()
    {
std::cout<< "Base class Destructor\n";
    }
};

class Derived : public Base
{
public:
Derived  () : Base()
    {
std::cout<< "Derived class Constructor\n";
    }
    ~Derived ()
    {
std::cout<< "Derived class Destructor\n";
    }
};
```

Output:

```
Base class Constructor

Derived classConstructor
```

Derived class Destructor

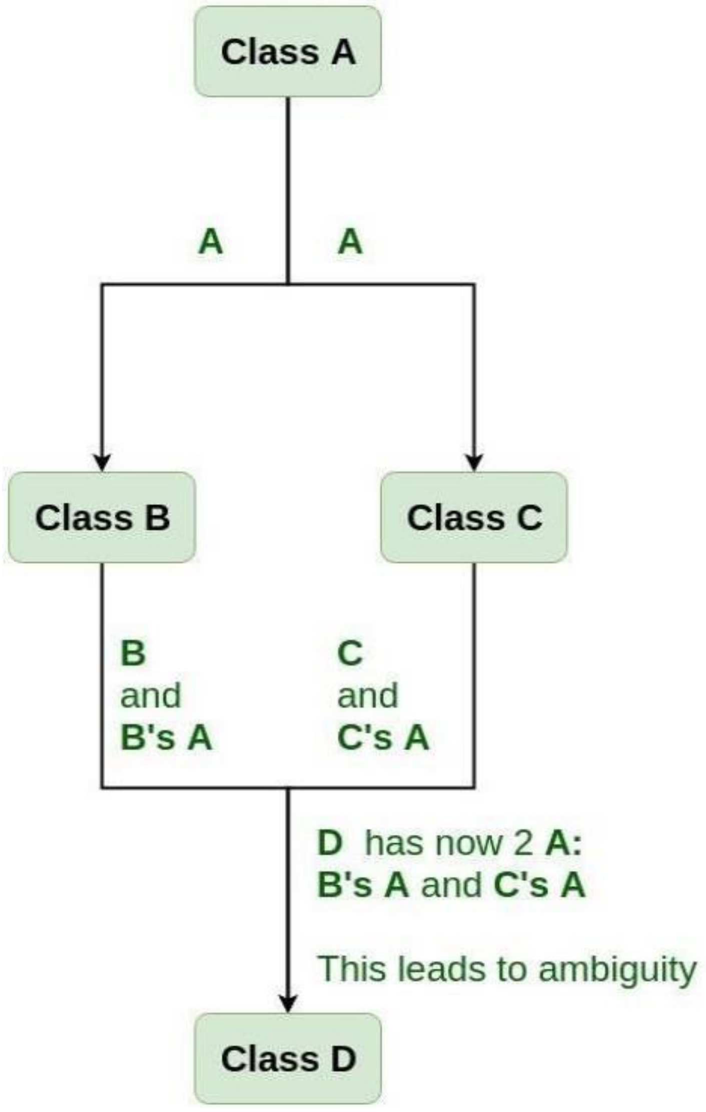
Base class Destructor

Virtual base class in C++

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class's name with the word virtual.

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:
Consider the situation where we have one class **A. This** class is **A** is inherited by two other classes **B** and **C**. Both these classes are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A**are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class**A** is accessed by an object of class **D, ambiguity** arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Example: To show the need of Virtual Base Class in C++#include<iostream>
usingnamespacestd;

```
class A
{public:
   voidshow()
   {
      cout<<"Hello form A \n";
   }
};
```

```
classB : public A {
};

classC : public A {
};

classD : publicB, public C {
};

int main()
{
    D
    object;object.s
    how();
}
```

CompileErrors:

```
prog.cpp: Infunction'intmain()':

prog.cpp:29:9:error:requestformember'show'isambiguousobje

  ct.show();

         ^

prog.cpp:8:8: note: candidates are: void

   A::show()voidshow()

        ^

prog.cpp:8:8:note:                  voidA::show()
```

How to resolve this issue?
To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual baseclass** by placinga keyword **virtual** as :

Syntax for Virtual BaseClasses:

```
Syntax1:
class B : virtual public A
```

```
{
};
```

Syntax2:

```
class C : public virtual A
{
};
```

Note:virtual canbe written before or after the **public**.Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies thatuse multiple inheritances. When a base class is specified as a virtual base, it can act as anindirect base more than once without duplication of its data members. A single copy of itsdatamembers is shared by all thebaseclasses thatusevirtual base.

Example1

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

class A
{public:
   int a;
   A()//constructor
   {
      a=10;
   }
};

classB : publicvirtual A {
};

classC : publicvirtual A {
};

classD : publicB, public C {
};

int main()
{
   D object; // object creation of class
   dcout<<"a ="<<object.a<<endl;

   return0;
}
```

Output:

```
a=10
```

Explanation : The class **A** has just one data member **a** which is **public**. This class isvirtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual baseclassand no duplicationof data member **a** is done.

Example2:

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



class A
{public:
   voidshow()
   {
      cout<<"Hello from A \n";
   }
};

classB : publicvirtual A {
};

classC : publicvirtual A {
};

classD : publicB, public C {
};

int main()
{
   D
   object;object.s
   how();
}
```

Output:

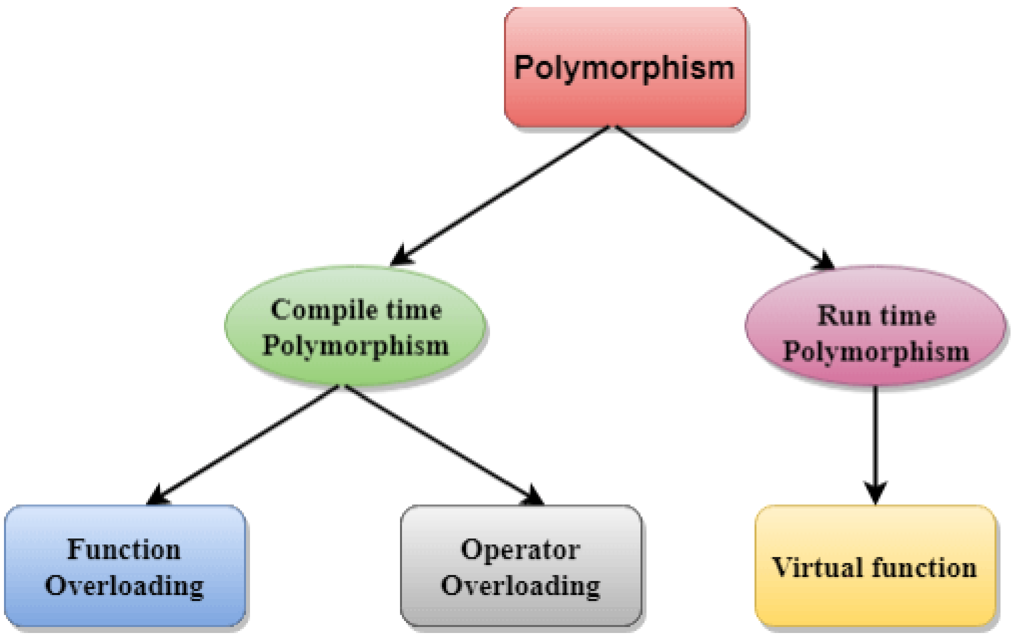
```
HellofromA
```

VIRTUAL FUNCTIONS and POLYMORPHISM

Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word.. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

There are two types of polymorphism in C++:



- **Compile time polymorphism**: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
 class A                          //  base class declaration.
{
    int a;
    public:
    void display()
    {
        cout<< "Class A ";
        }
    };
class B : public A                     //  derived class declaration.
{
    int b;
    public:
   void display()
   {
        cout<<"Class B";
   }
};
```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism**: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	**Run time polymorphism**
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.

It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {           cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape {                            //  base class
    public:
virtual void draw(){                     // virtual function
cout<<"drawing..."<<endl;
    }
};
class Rectangle: public Shape                //  inheriting Shape class.
{
 public:
 void draw()
   {
       cout<<"drawing rectangle..."<<endl;
    }
};
class Circle: public Shape                   //  inheriting Shape class.

{
 public:
 void draw()
   {
      cout<<"drawing circle..."<<endl;
   }
};
int main(void) {
    Shape *s;                           //  base class pointer.
    Shape sh;                           // base class object.
       Rectangle rec;
        Circle cir;
      s=&sh;
     s->draw();
        s=&rec;
     s->draw();
    s=?
    s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                          //    class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

```
30
55
```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
    std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
     // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
   private:
      int num;
   public:
       Test(): num(8){}
       void operator ++()         {
          num = num+2;
       }
       void Print() {
           cout<<"The Count is: "<<num;
       }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```cpp
#include <iostream>
using namespace std;
class A
{

    int x;
      public:
      A(){}
    A(int i)
    {
       x=i;
    }
    void operator +(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

```
The result of the addition of two objects is : 9
```

Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
public:
void eat()
   {
      cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

```
Eating bread...
```

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor

  Consider the situation when we don't use the virtual keyword.

```cpp
#include <iostream>
using namespace std;
class A
{
  int x=5;
   public:
   void display()
   {
       std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
       std::cout << "Value of y is : " <<y<< std::endl;
   }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;
}
```

Output:

```
Value of x is : 5
```

In the above example, *a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Working Mechanism of Virtual Function

A virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. It is one that is declared as virtual in the base class using the virtual keyword. The virtual nature is inherited in the subsequent derived classes and the virtual keyword need not be re-stated there. The whole function body can be replaced with a new set of implementations in the derived class.

Binding:

Binding refers to the act of associating an object or a class with its member. If we can call a method fn() on an object o of a class c, we say that the object o is binded with the method fn(). This happens at compile time and is known as static or compile – time binding. The calls to the virtual member functions are resolved during run-time. This mechanism is known as dynamic binding. The most prominent reason why a virtual function will be used is to have a different functionality in the derived class. The difference between a non-virtual member function and a virtual member function is, the non-virtual member functions are resolved at compile time.

Working Process of Virtual Functions:

Whenever a program has a virtual function declared,

- a v – table is constructed for the class. The v-table consists of addresses to the virtual functions for classes that contain one or more virtual functions.
- The object of the class containing the virtual function contains a virtual pointer that points to the base address of the virtual table in memory.
- Whenever there is a virtual function call, the v-table is used to resolve to the function address. An object of the class that contains one or more virtual functions contains a virtual pointer called the vptr at the very beginning of the object in the memory.
- Hence the size of the object in this case increases by the size of the pointer.
- This vptr contains the base address of the virtual table in memory.
- The virtual tables are class specific, i.e., there is only one virtual table for a class irrespective of the number of virtual functions it contains.
- This virtual table in turn contains the base addresses of one or more virtual functions of the class.

- At the time when a virtual function is called on an object, the vptr of that object provides the base address of the virtual table for that class in memory.
- This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call.

The following code shows how we can write a virtual function in C++ and then use the same to achieve dynamic or runtime polymorphism.

```
#include <iostream.h>
class base
{
public:
virtual void display()
{
cout<<"\nBase";
}
};
class derived : public base
{
public:
void display()
{
cout<<"\nDerived";
}
};

void main()
{

base *ptr = new derived();
ptr->display();
}
```

In the above example, the pointer is of type base but it points to the derived class object. The method display() is virtual in nature. Hence in order to resolve the virtual method call, the context of the pointer is considered, i.e., the display method of the derived class is called and not that of the base. If the method was non virtual in nature, the display() method of the base class would have been called.

A constructor cannot be virtual because at the time when the constructor is invoked the virtual table would not be available in the memory. Hence we cannot have a virtual constructor.

Conclusion:

Virtual methods should be used judiciously as they are slow due to the overhead involved in searching the virtual table. They also increase the size of an object of a class by the size of a pointer.

Pure Virtual Function

A pure virtual function is a virtual function that has no definition within the class.

- A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.
- It can be considered as an empty function means that the pure virtual function does not have any definition relative to the base class.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.
- A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.

Syntax: `virtual void display() = 0;`

Example:

```
#include <iostream>
using namespace std;
// Abstract class
class Shape
{
   public:
   virtual float calculateArea() = 0; // pure virtual function.
};
class Square : public Shape
```

```
{
    float a;
    public:
    Square(float l)

    {
        a = l;
    }
    float calculateArea()
    {
        return a*a;
    }
};
class Circle : public Shape
{
    float  r;
    public:

    Circle(float x)
    {
        r = x;
    }
    float calculateArea()
    {
        return 3.14*r*r ;
    }
};

};
int main()
{

    Shape *shape;
    Square s(3.4);
    Circle c(7.8);
    shape =&s;
    int a1 =shape->calculateArea();
    shape = &c;
```

```
int a3 = shape->calculateArea();
cout << "Area of the square is " <<a1<<endl;
cout << "Area of the circle is " <<a3<<endl;
return 0;
 }
```

Similarities between virtual function and pure virtual function

1. These are the concepts of Run-time polymorphism.
2. Prototype i.e. Declaration of both the functions remains the same throughout the program.
3. These functions can't be global or static.

Difference between virtual function and pure virtual function in C++

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
`virtual<func_type><func_name>()`
`{`
`  // code`
`}`
	Syntax:
	`virtual<func_type><func_name>()`
	`  = 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.

Virtual function	Pure virtual function
made.	
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class object cannot be created, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Below we have a simple example where we have defined an abstract class,

```
//Abstract base class
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
   public:
    void show()
    {
        cout << "Implementation of Virtual Function in Derived class\n";
    }
};

int main()
{
    Base obj;   //Compile Time Error
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

```
Output:

Implementation of Virtual Function in Derived class
```

Virtual Destructor

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
// CPP program without virtual destructor
// causing undefined behavior
#include <iostream>
using namespace std;
class base {
  public:
    base()
    {
cout << "Constructing base\n";
}
    ~base()
    {
 cout<< "Destructing base\n";
}
};
class derived: public base {
  public:
    derived()
     {
cout << "Constructing derived\n";
}
```

```
    ~derived()
       {
cout << "Destructing derived\n";
}
};

int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following:

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
// A program with virtual destructor
#include <iostream>

using namespace std;

class base {
  public:
    base()
    {
 cout << "Constructing base\n";
}
    virtual ~base()
    {
cout << "Destructing base\n";
}
};

class derived : public base {
  public:
    derived()
    {
```

```
cout << "Constructing derived\n";
}
    virtual ~derived()
    {
cout << "Destructing derived\n";
 }
};
int main()
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Output:

```
Constructing base

Constructing derived

Destructing derived

Destructing base
```

UNIT - IV
C++ I/O

C++ supports two complete I/O systems.

- Inherits from C.
- Object-oriented I/O system defined by C++

NOTE: C++ programs can also use the C++-style header #include<cstdio>
.

I/O using C functions

It includes,

- Console I/O
- Streams
- Files

Console I/O

This can be divided into Unformatted and Formatted Console I/O.

Unformatted Console I/O

a. **Reading and Writing Characters**

The simplest of the console I/O functions are getchar( ) and putchar().

getchar( ) : It reads a character from the keyboard. It waits until a key is pressed and then returns its value. The key pressed is also automatically echoed to the screen.

Prototype: int getchar(void);

putchar( ): It prints or writes a character to the screen at the current cursor position.

Prototype: int putchar(int c);

Program:

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

int main()
{
    char ch;
    cout<<"\n Enter a character in lower case: ";
    ch = getchar();
    cout<<"\nThe entered character is ";

    putchar(ch);
    cout<<"\nCharacter in UPPER CASE: ";
    putchar(ch - 32);
    return 0;
}
```

Output:

```
    Enter a character in lower case: t
    The entered character is t
    Character in UPPER CASE: T
```

b. **Alternatives to getchar( )**

getchar( ) is not useful in an interactive environment. Two of the most common alternative functions, getch( ) and getche( )

getch( ) : It waits for a keypress, after which it returns immediately. It does not echo the character to the screen.
Prototype: int getch(void);

getche( ) : It is the same as getch( ), but the key is echoed.
Prototype: int getche(void);

c. Reading and Writing Strings

gets( ) : It reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. You may type characters at the keyboard until you press ENTER. The carriage return does not become part of the string; instead, a null terminator is placed at the end and gets( ) returns.
Prototype: char *gets(char *str);

Problem with gets( ) : It performs no boundary checks on the array that is receiving input. Thus, it is possible for the user to enter more characters than the array can hold. One alternative is the fgets( ) function.

puts( ): It writes its string argument to the screen followed by a newline.
Prototype: int puts(const char *str);

Program:
```
include <iostream>
#include<cstdio>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    gets(str);
    cout << "You entered: " << str;

    char str1[] = "Happy New Year";
    char str2[] = "Happy Birthday";

    puts(str1);
    /*  Printed on new line since '/n' is added */
    puts(str2);

    return 0;
}
```

Output:
```
main.cpp:18:5: warning: ‘char* gets(char*)’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.cpp:18:13: warning: ‘char* gets(char*)’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.cpp:(.text+0x31): warning: the `gets' function is dangerous and should not be used.
Enter a string: rt
You entered: rtHappy New Year
Happy Birthday
```

Function	Operation
getchar( )	Reads a character from the keyboard; waits for carriage return.
getche( )	Reads a character with echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
getch( )	Reads a character without echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
putchar( )	Writes a character to the screen.
gets( )	Reads a string from the keyboard.
puts( )	Writes a string to the screen.

Formatted Console I/O

The functions printf( ) and scanf( ) perform formatted output and input. Both functions can operate on any of the built-in data types, including characters, strings, and numbers.

printf( ): It writes data to the console.
Prototype: int printf(const char *control_string, ...);

The control_string consists of two types of items. The first type is composed of characters that will be printed on the screen. The second type contains format specifiers that define the way the subsequent arguments are displayed. A format specifier begins with a percent sign and is followed by the format code.

Code	Format
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%%	Prints a % sign

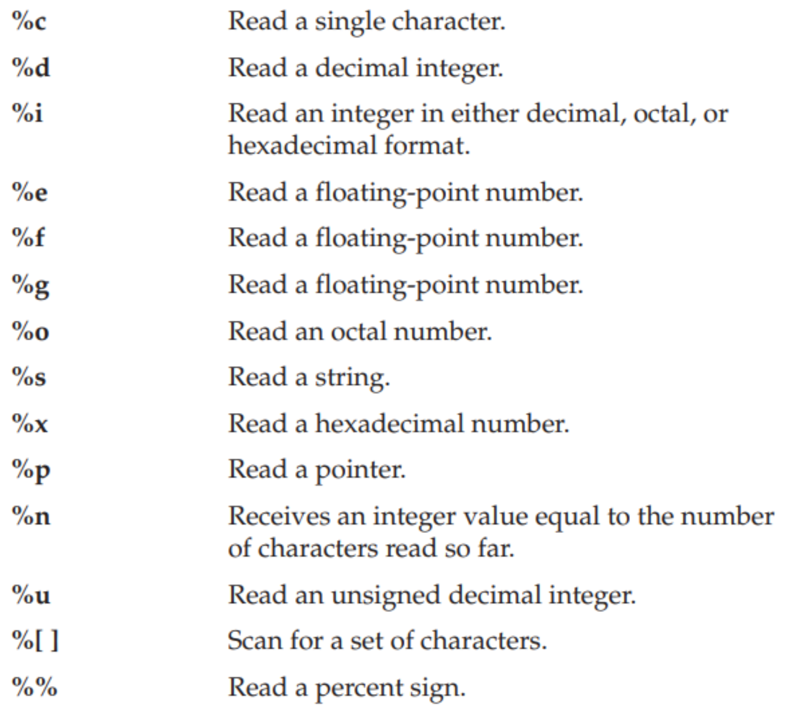
scanf( ): Its reads data from the keyboard.
Prototype: int scanf(const char *control_string, ...);

The control_string determines how values are read into the variables pointed to in the argument list. The control string consists of three classifications of characters:

- Format specifiers
- White-space characters
- Non-white-space characters

Format specifiers

The input format specifiers are preceded by a % sign and tell scanf( ) what type of data is to be read next.



%c	Read a single character.
%d	Read a decimal integer.
%i	Read an integer in either decimal, octal, or hexadecimal format.
%e	Read a floating-point number.
%f	Read a floating-point number.
%g	Read a floating-point number.
%o	Read an octal number.
%s	Read a string.
%x	Read a hexadecimal number.
%p	Read a pointer.
%n	Receives an integer value equal to the number of characters read so far.
%u	Read an unsigned decimal integer.
%[ ]	Scan for a set of characters.
%%	Read a percent sign.

White-space characters

A white-space character in the control string causes scanf( ) to skip over one or more leading white-space characters in the input stream. A white-space character is a space, a tab, vertical tab, form feed, or a newline.

Non-white-space characters

A non-white-space character in the control string causes scanf( ) to read and discard matching characters in the input stream. For example, "%d,%d" causes scanf( ) to read an integer, read and discard a comma, and then read another integer

Program:

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

int main()
{
    int f;
    printf("  ff");
    scanf("%d",f);
    printf(f);
    return 0;
}
```

Output: ff f

Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. The file system transforms each into a logical device called a stream. There are two types of streams: text and binary.

- **Text Streams**

A text stream is a sequence of characters. Standard C allows (but does not require) a text stream to be organized into lines terminated by a newline character.

Certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

- **Binary Streams**

A binary stream is a sequence of bytes that have a one-to-one correspondence to those in the external device that is, no character translations occur. Also, the number of bytes written (or read) is the same as the number on the external device.

The Standard Streams

As it relates to the C file system, when a program starts execution, three streams are opened automatically. They are stdin (standard input), stdout (standard output), and stderr (standard error).

Using freopen( ) to Redirect the Standard Streams

You can redirect the standard streams by using the freopen( ) function. This function associates an existing stream with a new file. Thus, you can use it to associate a standard stream with a new file.

Prototype: FILE *freopen(const char *filename, const char *mode, FILE *stream);

filename is a pointer to the filename you wish associated with the stream pointed to by stream. The file is opened using the value of mode, which may have the same values as those used with fopen( ). freopen( ) returns stream if successful or NULL on failure.

Program

```
#include <cstdio>
#include <cstdlib>

int main()
{
    FILE* fp = fopen("test1.txt","w");
    fprintf(fp,"%s","This is written to test1.txt");

    if (freopen("test2.txt","w",fp))
    fprintf(fp,"%s","This is written to test2.txt");
    else
    {
    printf("freopen failed");
    exit(1);
}

    fclose(fp);
    return 0;
}
```

Output:

Files

In C/C++, a *file* may be anything from a disk file to a terminal or printer. Each stream that is associated with a file has a file control structure of type **FILE**.

➢ **File System Basics**

The C file system is composed of several interrelated functions. C++ programs may also use the C++-style header **<cstdio>**.

➢ **The File Pointer**

The file pointer is the common thread that unites the C I/O system. A *file pointer* is a pointer to a structure of type **FILE**. It points to information that defines various things about the file, including its name, status, and the current position of the file.

In order to read or write files, your program needs to use file pointers

Prototype : FILE *fp;

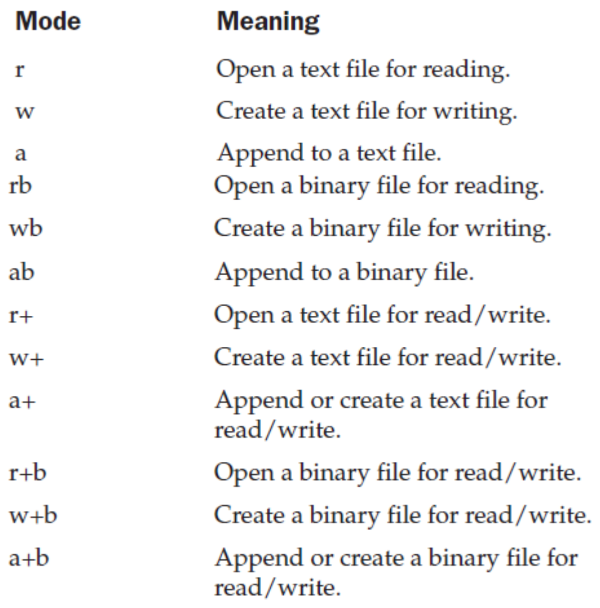
File Operations

- **fopen( )** : It opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file.

Prototype: FILE *fopen(const char **filename*, const char **mode*);

where *filename* is a pointer to a string of characters that make up a valid filename and may include a path specification.

The legal values for *mode are,*



Mode	Meaning
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

- **fclose( ):** It closes a stream that was opened by a call to **fopen( )**.

Prototype: int fclose(FILE **fp*);

where *fp* is the file pointer returned by the call to **fopen( )**. The function returns **EOF** if an error occurs.

Program:

```
#include <cstdio>
#include <cstring>
```

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

int main()
{
    int c;
    FILE *fp;
    fp = fopen("file.txt", "w+r");
    char str[20] = "Hello World!";
    if (fp)
    {
        for(int i=0; i<strlen(str); i++)
        putc(str[i],fp);
    }
    fclose(fp);
}
```

Output:

```
        Hello World!
```

- **putc( )** and **fputc( )**

These two equivalent functions writes characters to a file that was previously opened for writing using the **fopen( )** function.

Prototype : int putc(int *ch*, FILE **fp*);

where *fp* is the file pointer returned by **fopen( )** and *ch* is the character to be output. The file pointer tells **putc( )** which file to write to.

If a **putc( )** operation is successful, it returns the character written. Otherwise, it returns **EOF**.

Program:

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

int main()
{
    char str[] = "Testing putc() function";
    FILE *fp;

    fp = fopen("file.txt","w");

    if (fp)
    {
      for(int i=0; i<strlen(str); i++)
      {
      putc(str[i],fp);
      }

      for(int i=0; i<strlen(str); i++)
      {
      fputc(str[i],fp);
      }
    }
    else
      perror("File opening failed");
      fclose(fp);
   return 0;
}
```

Output:

```
        Testing putc() functionTesting putc() function
```

- **getc( )** and **fgetc( )**
  These two equivalent functions reads characters from a file opened in read mode by **fopen( )**.

Prototype : int getc(FILE **fp*);

where *fp* is a file pointer of type **FILE** returned by **fopen( )**. The **getc( )** function returns an **EOF** when the end of the file has been reached. **getc( )** also returns **EOF** if an error occurs.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
   int c;
   FILE *fp;

   fp = fopen("file.txt","r");

   if (fp)
   {
      while(feof(fp) == 0)
      {
      c = getc(fp);
      putchar(c);
      }
      while(feof(fp) == 0)
      {
      c = fgetc(fp);
      putchar(c);
      }
   }
   else
      perror("File opening failed");
      fclose(fp);
   return 0;
}
```

Output:

```
Testing putc() functionTesting putc() function
```

- **feof( )**: It determines when the end of the file has been encountered.

Prototype: int feof(FILE **fp*);

feof( ) returns true if the end of the file has been reached; otherwise, it returns 0.

- **fputs( ) and fgets( )**

These functions work just like **putc( )** and **getc( )**, but instead of reading or writing a single character, they read or write strings.

Prototypes:

int fputs(const char **str*, FILE **fp*);
char *fgets(char **str*, int *length*, FILE **fp*);

The **fputs( )** function writes the string pointed to by *str* to the specified stream. It returns **EOF** if an error occurs.

The **fgets( )** function reads a string from the specified stream until either a newline character is read or *length* −1 characters have been read.

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int count = 10;
    char str[10];
    FILE *fp;

    fp = fopen("file.txt","w+");
    fputs("An example file\n", fp);
    fputs("Filename is file.txt\n", fp);

    rewind(fp);

    while(feof(fp) == 0)
    {
        fgets(str,count,fp);
        cout << str << endl;
     }

    fclose(fp);
    return 0;
}
```

Output:
```
An exampl
e file

Filename
is file.t
xt

xt
```

- **rewind( )**

The **rewind( )** function resets the file position indicator to the beginning of the file specified as its argument. That is, it "rewinds" the file.

Prototype: void rewind(FILE **fp*);

where *fp* is a valid file pointer.

Program:

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

int main()
{
   int c;
   FILE *fp;
   fp = fopen("file.txt", "r+w");
   if (fp)
   {
     while ((c = getc(fp)) != EOF)
     putchar(c);

     rewind(fp);
     putchar('\n');

     while ((c = getc(fp)) != EOF)
     putchar(c);
   }
   fclose(fp);
   return 0;
}
```

Output:

```
welcome
welcome
```

- **ferror( )**

  The **ferror( )** function determines whether a file operation has produced an error.

Prototype: int ferror(FILE **fp*);

where *fp* is a valid file pointer. It returns true if an error has occurred during the last file operation; otherwise, it returns false.

Program:

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

int main()
{
   int ch;
   FILE* fp;
   fp = fopen("file.txt","w");

   if(fp)
   {
     ch = getc(fp);
     if (ferror(fp))
     cout << "Can't read from file";
   }
```

```
    fclose (fp);
    return 0;
}
```

Output:

```
Can't read from file
```

- **remove( )**

  The **remove( )** function erases the specified file.

Prototype: int remove(const char **filename*);

It returns zero if successful; otherwise, it returns a nonzero value.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    char filename[] =  "file.txt";

    /*Deletes the file if exists */
    if (remove(filename) != 0)
    perror("File deletion failed");
    else
    cout << "File deleted successfully";

    return 0;
}
```

Output:

```
File deleted successfully
```

- **fflush( )**

  If you wish to flush the contents of an output stream, use the **fflush( )** function.

Prototype: int fflush(FILE **fp*);

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int x;
    char buffer[1024];
    setvbuf(stdout, buffer, _IOFBF, 1024);
    printf("Enter an integer - ");
    fflush(stdout);
    scanf("%d",&x);
    printf("You entered %d", x);
    return(0);
}
```

Output:
```
Enter an integer - 89
You entered 89
```

- **fread( )** and **fwrite( )**.
  These functions allow the reading and writing of blocks of any type of data.

Prototypes:

size_t fread(void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);
size_t fwrite(const void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);

For **fread( )**, *buffer* is a pointer to a region of memory that will receive the data from the file. For **fwrite( )**, *buffer* is a pointer to the information that will be written to the file. The value of *count* determines how many items are read or written, with each item being *num_bytes* bytes in length. Finally, *fp* is a file pointer to a previously opened stream.

The **fread( )** function returns the number of items read. This value may be less than *count* if the end of the file is reached or an error occurs. The **fwrite( )** function returns the number of items written. This value will equal *count* unless an error occurs.

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int retVal;
    FILE *fp;
    char buffer[] = "Writing to a file using fwrite.";
    fp = fopen("data.txt","wb");

    retVal = fwrite(buffer,sizeof(buffer),1,fp);
    cout << "fwrite returned " << retVal;

    return 0;
}
```

Output:
```
fwrite returned 1
```

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
 FILE *fp;
 char buffer[100];

 fp = fopen("data.txt","rb");
 while(!feof(fp))
 {
```

```
 fread(buffer,sizeof(buffer),1,fp);
 cout << buffer;
 }

 return 0;
}
```

Output:

```
Writing to a file using fwrite.
```

Program:

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

    FILE *fp;
    double d = 12.23;
    int i = 101;
    long l = 123023L;
    if((fp=fopen("test", "wb+"))==NULL)
    {
      printf("Cannot open file.\n");
      exit(1);
    }
    fwrite(&d, sizeof(double), 1, fp);
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&l, sizeof(long), 1, fp);
    rewind(fp);
    fread(&d, sizeof(double), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread(&l, sizeof(long), 1, fp);
    printf("%f %d %ld", d, i, l);
    fclose(fp);
    return 0;
}
```

Output:

```
12.230000 101 123023
```

- **fseek( )**

    You can perform random-access read and write operations using the C I/O system with the help of **fseek( )**, which sets the file position indicator.

Prototype : int fseek(FILE **fp*, long int *numbytes*, int *origin*);

Here, *fp* is a file pointer returned by a call to **fopen( )**. *numbytes* is the number of bytes from *origin* that will become the new current position, and *origin* is one of the following macros:

Origin Macro	**Name**
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

Program:

```
#include <cstdio>
```

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

int main(int argc, char *argv[])
{
    FILE * pFile;
    pFile = fopen ( "example.txt" , "wb" );
    fputs ( "This is an apple." , pFile );
    fseek ( pFile , 9 , SEEK_SET );
    fputs ( " sam" , pFile );
    fclose ( pFile );
    return 0;
}
```

Output:
After this code is successfully executed, the file example.txt contains:

```
        This is a sample.
```

- **fprintf( ) and fscanf( )**
  These functions behave exactly like **printf( )** and **scanf( )** except that they operate with files.

Prototypes:

int fprintf(FILE *fp*, const char *control_string*,. . .);
int fscanf(FILE *fp*, const char *control_string*,. . .);

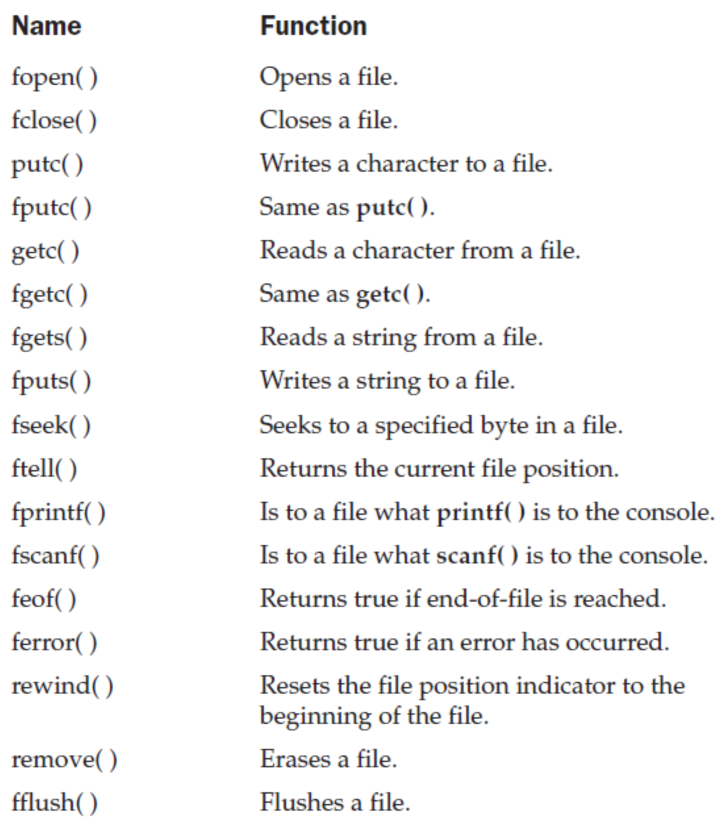
where *fp* is a file pointer returned by a call to **fopen( )**. **fprintf( )** and **fscanf( )** direct their I/O operations to the file pointed to by *fp*.

Program:

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

int main(int argc, char *argv[])
{
   FILE *fp;
   char name[50];
   int age;

   fp = fopen("example.txt","w");
   fprintf(fp, "%s %d", "Tim", 9);
   fclose(fp);
   fp = fopen("example.txt","r");
   fscanf(fp, "%s %d", name, &age);
   fclose(fp);
   printf("Hello %s, You are %d years old\n", name, age);
   return 0;
}
```

Output:

```
Hello Tim, You are 9 years old
```



Name	Function
fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as **putc( )**.
getc( )	Reads a character from a file.
fgetc( )	Same as **getc( )**.
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file what **printf( )** is to the console.
fscanf( )	Is to a file what **scanf( )** is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

Object-oriented I/O system defined by C++
C++ Streams

The C++ I/O system operates through streams. A *stream* is a logical device that either produces or consumes information.

A stream is linked to a physical device by the I/O system. All streams behave the same, the same I/O functions can operate

Stream classes' hierarchy

Standard C++ provides support for its I/O system in **<iostream>** header, which gives a set of class hierarchies is defined that supports I/O operations.
The C++ I/O system is built upon two related but different class hierarchies.

- **basic_streambuf**

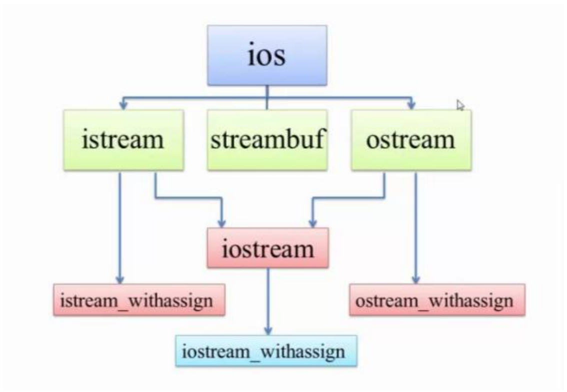
Low-level I/O class is called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly.

- **basic_ios**

The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O.

- **ios_base**

A base class for **basic_ios** is called **ios_base. basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively.



istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.
Class hierarchies for 8-bit characters and wide characters.

Class	Character-based Class	Wide-Character-based Class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

C++'s Predefined Streams

When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**. By default, the standard streams are used to communicate with the console.

Standard C++ also defines these four additional streams: **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

Operator Overloading
Overloading << and >>

<< and the >> operators are overloaded in C++ to perform I/O. the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream. The functions that overload the insertion and extraction operators are generally called *inserters* and *extractors*, respectively.

Creating Your Own Inserters

It is quite simple to create an inserter for a class that you create.
General form:
ostream &operator<<(ostream &*stream*, *class_type obj*)

```
{
    // body of inserter
    return stream;
}
```

The function returns a reference to a stream of type **ostream**. The first parameter to the function is a reference to the output stream. The second parameter is the object being inserted.

Within an inserter function, you may put any type of procedures or operations that you want. inserters cannot be members of the class for which they are defined seems to be a serious problem because they cannot access the private elements of a class. Solution is to Make the inserter a **friend** of the class

An inserter need not be limited to handling only text. An inserter can be used to output data in any form like CAD plotters, graphics images, dialog boxes etc.

Creating Your Own Extractors

Extractors are the complement of inserters.

General form

```
istream &operator>>(istream &stream, class_type &obj)
{
    // body of extractor
    return stream;
}
```

Extractors return a reference to a stream of type **istream**, which is an input stream. The first parameter must also be a reference to a stream of type **istream**. The second parameter must be a reference to an object of the class for which the extractor is overloaded. This is so the object can be modified by the input (extraction) operation.

Program:

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

class Box
{
    double height;
    double width;
    double vol ;

    public :
    friend istream & operator >> (istream &, Box &);
    friend ostream & operator << (ostream &, Box &);
};

istream & operator >> (istream &stream, Box &b)
{
    cout << "Enter Box Height: " ; stream >> b.height ;
    cout << "Enter Box Width : " ; stream >> b.width ;
    return (stream) ;
}
ostream & operator << (ostream &stream, Box &b)
{
    stream << endl << endl;
    stream << "Box Height : " << b.height << endl ;
    stream << "Box Width  : " << b.width << endl ;

    b.vol = b.height * b.width ;
    stream << "The Volume of Box : " << b.vol << endl;

    return(stream) ;
}

 int main()
 {
     Box b1;
     cin >> b1;
     cout << b1;
}
```

Output:

```
Enter Box Height: 1
Enter Box Width : 2

Box Height : 1
Box Width  : 2
The Volume of Box : 2
```

Creating Your Own Manipulator Functions

We can customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons.

- You can consolidate a sequence of several separate I/O operations into one manipulator.
- When you need to perform I/O operations on a nonstandard device. For example, you might use a manipulator to send control codes to a special type of printer or to an optical recognition system.

Types of manipulators

There are two basic types of manipulators:

- Those that operate on input streams
- Those that operate on output streams.

Apart from this , there is one more classification,

- Manipulators that take an argument

  The procedures necessary to create a parameterized manipulator vary widely from compiler to compiler, and even between two different versions of the same compiler. For this reason, you must consult the documentation to your compiler for instructions on creating parameterized manipulators
- Manipulators that don't.take an argument

  The creation of parameterless manipulators is straightforward and the same for all compilers.

General Form

```
ostream &manip-name(ostream &stream)
{
   // your code here
   return stream;
}
```

manip-name is the name of the manipulator. a reference to a stream of type **ostream** is returned. This is necessary if a manipulator is used as part of a larger I/O expression.

Using an output manipulator is particularly useful for sending special codes to a device. For example, a printer may be able to accept various codes that change the type size or font, or that position the print head in a special location. If these adjustments are going to be made frequently, they are perfect candidates for a manipulator.

General Form

```
istream &manip-name(istream &stream)
{
  // your code here
  return stream;
}
```

An input manipulator receives a reference to the stream for which it was invoked. This stream must be returned by the manipulator.

Program:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>
using namespace std;

// A simple output manipulator that sets the fill character to * and sets the field width to 10.
ostream &star_fill(ostream &stream)
{
      stream << setfill('*') << setw(10);
      return stream;
}

// A simple input manipulator that skips leading digits.
istream &skip_digits(istream &stream)
{
      char ch;/*w  w  w .  j  ava 2s.c  o  m*/
      do
      {
      ch = stream.get();
      } while(!stream.eof() && isdigit(ch));

      if(!stream.eof()) stream.unget();
      return stream;
}
int main()
{
      string str;
      // Demonstrate the custom output manipulator.
      cout << 512 << endl;
      cout << star_fill << 512 << endl;
      // Demonstrate the custom input manipulator.
      cout << "Enter some characters: ";
      cin >> skip_digits >> str;
      cout << "Contents of str: " << str;
       return 0;
}
```

Output:

```
512
*******512
Enter some characters: abc
```

```
Contents of str: abc
```

File streams and String streams
String streams

C++ provides a <sstream> header , which uses the same public interface to support I/O between a program and string object.

The string streams is based on **istringstream(subclass of istream)**, and **ostringstream(subclass of ostream) and bidirectional stringstream(subclass of iostream)**,

General Form:

```
typedef  basic_istringstream<char>istringstream;
typedef  basic_ostringstream<char>ostringstream;
```

Stream input can be used to validate input data,stream output can be used to format the output.

Ostringstream constructors

```
explicit ostringstream(ios::openmode mode=ios::out);//default with empty string
explicit ostringstream(const string &str, ios::openmode
mode=ios::out);//with initial str
string str() const;//get contents
void str(const string &s)//set contents
```

Example:

```
ostringstream sout;
//write into string buffer
sout<<"apple"<<endl;
sout<<"orange"<<endl;
//get contents
cout<<sout.str()<<endl;
```

ostringstream is responsible for dynamic memory allocation and management.

istringstream constructors

```
explicit istringstream(ios::openmode mode=ios::in); //default with empty string
explicit istringstream(const string &str, ios::openmode mode=ios::in); //with initial str
```

Example:

```
istringstream sin("123    12.34    hello");
//read from buffer
int I;
double d;
string s;
sin>>i>>d>>s;
cout<<i<<","<<d<<","<<s<<endl;
```

stringstream constructors

```
explicit stringstream(ios::openmode mode = ios::in | ios::out);
explicit stringstream(const string &str,
ios::openmode mode = ios::in | ios::out);
```

Program:

```
// Demonstrate string streams.
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main()
{
    stringstream s("This is initial string.");
    // get string
    string str = s.str();
    cout << str << endl;
    // output to string stream
    s << "Numbers: " << 10 << " " << 123.2;
    int i;
    double d;
    s >> str >> i >> d;
    cout << str << " " << i << " " << d;
    return 0;
}
```

Output:

```
This is initial string.
Numbers: 10 123.2
```

File streams
Formatted file streams

To perform file I/O, you must include the header **<fstream>** in your program. It defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. Remember, **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios**

Opening and Closing a File
open()

In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream.

There are three types of streams:

Input

To create an input stream, you must declare the stream to be of class **ifstream**.

Output

To create an output stream, you must declare it as class **ofstream**.

Input/Output

Streams that will be performing both input and output operations must be declared as class **fstream**.

General form for creating streams

```
    ifstream in; // input
    ofstream out; // output
    fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using **open( )**. This function is a member of each of the three stream classes.

Prototype:

void ifstream::open(const char **filename*, ios::openmode *mode* = ios::in);
void ofstream::open(const char **filename*, ios::openmode *mode* = ios::out | ios::trunc);
void fstream::open(const char **filename*, ios::openmode *mode* = ios::in | ios::out);

filename is the name of the file; it can include a path specifier. The value of *mode* determines how the file is opened. It must be one or more of the following values

- **ios::app :** Including **ios::app** causes all output to that file to be appended to the end. This value can be used only with files capable of output.
- **ios::ate :** Including **ios::ate** causes a seek to the end of the file to occur when the file is opened.
- **ios::in :** The **ios::in** value specifies that the file is capable of input.
- **ios::out :** The **ios::out** value specifies that the file is capable of output.
- **ios::binary :** The **ios::binary** value causes a file to be opened in binary mode. By default, all files are opened in text mode.
- **ios::trunc :** The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length.

Checking open() is successful or not

a. If **open( )** fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded.

Example:

```
 if(!mystream) {
cout << "Cannot open file.\n";
// handle error
}
```

b. You can also check to see if you have successfully opened a file by using the **is_open( )** function, which is a member of **fstream**, **ifstream**, and **ofstream**.

Prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise.

Example:

```
if(!mystream.is_open()) {
cout << "File is not open.\n";
// ...
```

close()

To close a file, use the member function **close( )**

Prototype: mystream.close();

The **close( )** function takes no parameters and returns no value.

Reading and Writing Text Files

It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using **cin** and **cout**, substitute a stream that is linked to a file.

Program:

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

int main()
{
   ifstream in("INVNTRY"); // input
   if(!in)
   {
       cout << "Cannot open INVENTORY file.\n";
       return 1;
```

```
    }
    char item[20];
    float cost;
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in.close();

    ofstream out;
    out.open("INVNTRY");// output, normal file
    if(!out)
    {
          cout << "Cannot open INVENTORY file.\n";
          return 1;
    }
    out << "Radios " << 39.95 << endl;
    out << "Toasters " << 19.95 << endl;
    out << "Mixers " << 24.80 << endl;
    out.close();
    return 0;
}
```

Output:

```
Radios 39.95
Toasters 19.95
Mixers 24.8
```

Unformatted and Binary I/O

There will be times when you need to store unformatted (raw) binary data, not text. When performing binary operations on a file , openshould use **ios::binary** mode specifier

- **get( )**
  get( ) will read a character

General form: istream &get(char &*ch*);

reads a single character and puts that value in *ch*. It returns a reference to the stream

Overloading of get()

The **get( )** function is overloaded in several different ways.

Prototypes:

istream &get(char **buf*, streamsize *num*);
reads characters into the array pointed to by *buf* until either *num*-1

istream &get(char **buf*, streamsize *num*, char *delim*);
reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered.

int get( );
returns the next character from the stream

- **put( )**
  put( ) will write a character.

General form: ostream &put(char *ch*);

writes *ch* to the stream and returns a reference to the stream.

- **read( ) and write( )**

  Used to read and write blocks of binary data.

Prototypes:

istream &read(char **buf*, streamsize *num*);

reads *num* characters from the invoking stream and puts them in the buffer pointed to by *buf*.

ostream &write(const char **buf*, streamsize *num*);

writes *num* characters to the invoking stream from the buffer pointed to by *buf*.

- **getline( )**

  It also performs input. It is a member of each input stream class.

Prototypes:

istream &getline(char **buf*, streamsize *num*);

reads characters into the array pointed to by *buf* until either *num*-1

istream &getline(char **buf*, streamsize *num*, char *delim*);

reads characters into the array pointed to by *buf* until either *num*−1 characters have been read, the character specified by *delim* has been found

- **Detecting EOF**

  You can detect when the end of the file is reached by using the member function **eof( )**

Prototype: bool eof( );

It returns true when the end of the file has been reached; otherwise it returns false.

- **ignore( ) Function**

  You can use the **ignore( )** member function to read and discard characters from the input stream.

Prototype:

istream &ignore(streamsize *num*=1, int_type *delim*=EOF);

It reads and discards characters until either *num* characters have been ignored (1 by default) or the character specified by *delim* is encountered (**EOF** by default).

- **peek( )**

  You can obtain the next character in the input stream without removing it from that stream by using **peek( ).**

Prototype: int_type peek( );

It returns the next character in the stream or **EOF** if the end of the file is encountered.

- **putback( )**

  You can return the last character read from a stream to that stream by using **putback( ).**

Prototype : istream &putback(char *c*);

where *c* is the last character read.

- **flush( )**

  We can force the information to be physically written to disk before the buffer is full by calling **flush( )**.

Prototype: ostream &flush( );

Random Access

You perform random access by using the **seekg( )** and **seekp( ).**

- **seekg( )**

  The **seekg( )** function moves the associated file's current get pointer *offset* number of characters from the specified *origin*, which must be one of these three values:

ios::beg	Beginning-of-file
ios::cur	Current location
ios::end	End-of-file

Prototype: istream &seekg(off_type *offset*, seekdir *origin*);

- **seekp( )**

The **seekp( )** function moves the associated file's current put pointer *offset* number of characters from the specified *origin*

Prototype: ostream &seekp(off_type *offset*, seekdir *origin*);

Obtaining the Current File Position

You can determine the current position of each file pointer by using these functions:

Prototypes: pos_type tellg( );
pos_type tellp( );

Here, **pos_type** is a type defined by **ios** that is capable of holding the largest value that either function can return.

You can use the values returned by **tellg( )** and **tellp( )** as arguments to the following forms of **seekg( )** and **seekp( )**, respectively.

istream &seekg(pos_type *pos)*;
ostream &seekp(pos_type *pos*);

Error handling during files operations

We have been opening and using the files for reading and writing on the assumption that everything is fine with the files. This may not be true always true.

For instance, one of the following things may happen when dealing with the files,

1. A file which we are attempting to open for reading does not exists
2. The file name used for a new file may already exists
3. We may attempt an invalid operation such as reading past the EOF.
4. There may not be any space in the disk for storing more data.
5. We may use an invalid file name.
6. We may attempt to perform an operation when the file is not opened for that purpose.

We can handle these types of error situations in the following ways,

a. I/O Status

The C++ I/O system maintains status information about the outcome of each I/O operation. The current state of the I/O system is held in an object of type **iostate**, which is an enumeration defined by **ios** that includes the following members.

Name	Meaning
ios::goodbit	No error bits set
ios::eofbit	1 when end-of-file is encountered; 0 otherwise
ios::failbit	1 when a (possibly) nonfatal I/O error has occurred;0 otherwise
ios::badbit	1 when a fatal I/O error has occurred; 0 otherwise

There are two ways in which you can obtain I/O status information.

- Call the **rdstate( )** function.

Prototype: iostate rdstate( );

It returns the current status of the error flags.

- We can determine if an error has occurred is by using one or more of these functions:

bool bad( );	The **bad( )** function returns true if **badbit** is set.
bool eof( );	returns true when end of the file has reached
bool fail( );	The **fail( )** returns true if **failbit** is set.

bool good( );	The **good( )** function returns true if there are no errors. Otherwise, it returns false.

Clearing an Error

Once an error has occurred, it may need to be cleared before your program continues. To do this, use the **clear( )** function.

Prototype: void clear(iostate *flags*=ios::goodbit);

If *flags* is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set *flags* as you desire.

Formatted I/O.

The C++ I/O system allows you to format I/O operations. There are two related but conceptually different ways that you can format data.

1. Directly access members of the **ios** class.(flags and functions in ios class)
2. Special functions called *manipulators*

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted.

a. Flags

The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

- When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.
- When the **left** flag is set, output is left justified.
- When **right** is set, output is right justified.
- When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character.
- **oct** flag causes output to be displayed in octal.
- Setting the **hex** flag causes output to be displayed in hexadecimal.
- To return output to decimal, set the **dec** flag.
- Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.
- By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.
- Setting **showpos** causes a leading plus sign to be displayed before positive values.
- Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.
- By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.
- When **unitbuf** is set, the buffer is flushed after each insertion operation.
- When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.
- Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**.
- Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**.
- Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the Format Flags

To set a flag, use the **setf( )** function. This function is a member of **ios**.

Common form: fmtflags setf(fmtflags *flags*);

This function returns the previous settings of the format flags and turns on those flags specified by *flags*.

Example: stream.setf(ios::showpos);

stream is the stream you wish to affect.

NOTE: The format flags are defined within the **ios** class, you must access their values by using **ios** and the scope resolution operator.

Clearing Format Flags

The complement of **setf( )** is **unsetf( )**. This member function of **ios** is used to clear one or more format flags.

General form: void unsetf(fmtflags *flags*);

The flags specified by *flags* are cleared.

Overloaded Form of setf( )

There is an overloaded form of **setf( ) .**

General form: fmtflags setf(fmtflags *flags1*, fmtflags *flags2*);

In this version, only the flags specified by *flags2* are affected. the most common use of the two-parameter form of **setf( )** is when setting the number base, justification, and format flags.

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

int main ()
{
    cout.setf (ios::uppercase | ios::scientific);
    cout << 100.12;                    // displays 1.001200E+02
    cout.unsetf (ios::uppercase);  // clear uppercase
    cout << " \n" << 100.12 << endl;       // displays 1.001200e+02
    //OVERLOADED FORM OF setf
    cout.setf (ios::showpoint | ios::showpos, ios::showpoint);
    cout << 100.0<<endl;                       // displays 100.000, not +100.000
    //TWO PARAMETER FORM  of setf
    cout.setf(ios::hex, ios::basefield);
    cout << 100; // this displays 64

    return 0;
}
```

Output:
```
1.001200E+02
1.001200e+02
1.000000e+02
64
```

Setting All Flags

The **flags( )** function has a second form that allows you to set all format flags associated with a stream.

Prototype: fmtflags flags(fmtflags *f*);

When you use this version, the bit pattern found in *f* is used to set the format flags associated with the stream. Thus, all format flags are affected. The function returns the previous settings.

Example: cout.flags(f);

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

void showflags();
int main()
{
    // show default condition of format flags
    showflags();
    // showpos, showbase, oct, right are on, others off
    ios::fmtflags f = ios::showpos | ios::showbase | ios::oct | ios::right;
    cout.flags(f); // set all flags
    showflags();
    return 0;
}
// This function displays the status of the format flags.
void showflags()
{
    ios::fmtflags f;
    long i;
    f = cout.flags(); // get flag settings
    // check each flag
    for(i=0x4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";
    cout << " \n";
}
```

Output:
```
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 1 0 1 0 1 1 0 0 0 0 0 0
```

b. Functions

There are three member functions defined by **ios.**

- **width( )**

By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width()** function.

Prototype; streamsize width(streamsize *w*);

Here, *w* becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The **streamsize** type is defined as some form of integer by the compiler.

After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width. If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.

- **precision( )**

When outputting floating-point values, you can determine the number of digits of precision by using the **precision( )** function.

Prototype: streamsize precision(streamsize *p*);

The precision is set to *p*, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If it is not, then the default precision will be used.

- **fill( )**

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill( )** function.

Prototype: char fill(char *ch*);

After a call to **fill( )**, *ch* becomes the new fill character, and the old one is returned.

Overloaded forms of width( ), precision( ), and fill( )

There are overloaded forms of **width( )**, **precision( )**, and **fill( )** that obtain but do not change the current setting. These forms are shown here:

```
char fill( );
streamsize width( );
streamsize precision( );
```

Program:

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


int main ()
{
    cout.precision (4);
    cout.width (10);
    cout << 10.12345 << "\n";   // displays 10.12
    cout.fill ('*');
    cout.width (10);
    cout << 10.12345 << "\n";   // displays *****10.12
  // field width applies to strings, too
    cout.width (10);
    cout << "Hi!" << "\n";         // displays *******Hi!
    cout.width (10);
    cout.setf (ios::left);  // left justify
    cout << 10.12345;              // displays 10.12*****
    return 0;
}
```

Output:

```
     10.12
*****10.12
*******Hi!
10.12*****
```

Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. many of the I/O manipulators parallel member functions of the **ios** class.

Manipulator	**Purpose**	**Input/Output**
boolalpha	Turns on **boolapha** flag.	Input/Output
dec	Turns on **dec** flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on **fixed** flag.	Output
flush	Flush a stream.	Output
hex	Turns on **hex** flag.	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
nobooalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output

noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program.

Program:
```
#include <iostream>
#include <iomanip>
using namespace std;

int main ()
{
    cout << hex << 100 << endl;
    cout << setfill ('?') << setw (10) << 2343.0;
    return 0;
}
```

Output:
```
64
??????2343
```

Advantage

The main advantage of using manipulators instead of the **ios** member functions is that they often allow more compact code to be written. You can use the **setiosflags( )** manipulator to directly set the various format flags related to a stream.

Program:

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

int main ()
{
    cout << setiosflags (ios::showpos);
    cout << setiosflags (ios::showbase);
    cout << 123 << " " << hex << 123;
    return 0;
}
```

```
+123 0x7b
```

UNIT-V
EXCEPTION HANDILING

INDRODUCTION

Common types of Errors

The common types of errors are logic errors and syntactic errors.

Logic Errors: These occur due to poor understanding of the problem and solution procedure.
Examples: Assigning a value to the wrong variable, multiplying 2 numbers instead of adding them etc.

Syntactic Errors: These occur due to poor understanding of the language itself.
Examples: Spelling mistakes, missing out quotes or brackets or semicolon etc.

Apart from these two, one more type of errors is **Exception**.

Definition of Exception: Exceptions are run time errors or unusual conditions that a program may encounter while executing.
Examples: Division by zero, access to an array outside of its bounds, running out of memory or disk space.

Exception Handling

It is a C++ built in language feature that allows us to manage run time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

BENEFITS OF EXCEPTION HANDLING

1. **Automation:** it automates much of the error-handling code that previously had to be coded "by hand" in any large program.
2. **Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
3. **Functions can handle any exceptions they choose:** A function can throw many exceptions, but may choose to handle some of them.
4. **Grouping of Error Types:** In C++, both basic types and objects can be thrown as exception. We can group or categorize them according to types.
5. **Handles** the occurring of error and allows normal execution of the program.

EXCEPTION HANDLING FUNDAMENTALS
(THE TRY BLOCK, CATCHING AN EXCDEPTION, THROWING AN EXCCEPTION)

C++ exception handling is built upon three keywords: **try, catch, and throw**

- The program statements that you want to monitor for exceptions are contained in a **try block.**
- If an exception (i.e., an error) occurs within the try block, it is thrown using **throw**
- When an exception is thrown, it is caught by its corresponding **catch statement**, which processes the exception. There can be more than one catch statement associated with a try. Which catch statement is used is determined by the type of the exception.

General form of try and catch

```
 try {
            // try block
}
catch (type1 arg)
{
            // catch block
}
catch (type2 arg)
 {
            // catch block
 }
catch (type3 arg)
 {
            // catch block
 }
             . . .

catch (typeN arg)
{
            // catch block
 }
```

General form of the throw

```
            throw exception;
```

Program:

```
// A simple exception handling example.
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                                        // start a try block
    cout << "Inside try block\n";
    throw 100;                             // throw an error
    cout << "This will not execute";
  }
  catch (int i)
  {                                        // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Abnormal Termination

The type of the exception must match the type specified in a catch statement. Usually, the code within a catch statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the catch. However, often an error cannot be fixed i.e. throw an exception for which there is no applicable catch statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function **terminate ( )** to be invoked. By default, terminate ( ) calls **abort( )** to stop your program.

Program:

```
// This example will not work.
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                                   // start a try block
    cout << "Inside try block\n";
    throw 100;                        // throw an error
    cout << "This will not execute";
  }
  catch (double i)
  {                                   // won't work for an int exception
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

Throwing an exception from outside the try block

An exception can be thrown from outside the try block as long as it is thrown by a function that is called from within try block.

Program:

```
/* Throwing an exception from a function outside the try block. */
#include <iostream>
using namespace std;
void
Xtest (int test)
{
  cout << "Inside Xtest, test is: " << test << "\n";
  if (test)
    throw test;
}

int main ()
{
  cout << "Start\n";
  try
  {                                 // start a try block
    cout << "Inside try block\n";
    Xtest (0);
    Xtest (1);
    Xtest (2);
  }
  catch (int i)
  {                                 // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

Localize a try/catch to a function

A try block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset.

Program:
```
#include <iostream>
using namespace std;
// Localize a try/catch to a function.
void
Xhandler (int test)
{
  try
  {
    if (test)
      throw test;
  }
  catch (int i)
  {
    cout << "Caught Exception #: " << i << '\n';
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);
  cout << "End";
  return 0;
}
```

Output:
```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

Catch statements when no exception is thrown

The code associated with a catch statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the catch altogether. When no exception is thrown, the catch statement does not execute.

Program:
```
#include <iostream>
using namespace std;
int main ()
{
 cout << "Start\n";
 try
 {                                    // start a try block
  cout << "Inside try block\n";
  cout << "Still inside try block\n";
 }
 catch (int i)
 {                                    // catch an error
  cout << "Caught an exception -- value is: ";
  cout << i << "\n";
 }
 cout << "End";
 return 0;
}
```

Output:

```
Start
Inside try block
Still inside try block
End
```

Using multiple catch Statements

There can be more than one catch associated with a try. However, each catch must catch a different type of exception. Which catch statement is used is determined by the type of the exception.

Program:

```
#include <iostream>
using namespace std;
// Different types of exceptions can be caught.
void
Xhandler (int test)
{
 try
 {
  if (test)
   throw test;
  else
   throw "Value is zero";
 }
 catch (int i)
 {
  cout << "Caught Exception #: " << i << '\n';
 }
 catch (const char *str)
 {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

CATCHING ALL EXCEPTIONS

Using multiple catch statements we can write a separate catch statements for each type. But this is a complicated task. In these circumstances we want an exception handler to catch all exceptions instead of just a certain type.

General form:

```
catch(...)
{
// process all exceptions
}
```

Here, the ellipsis matches any type of data.

Program:

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void
Xhandler (int test)
{
  try
  {
    if (test == 0)
```

```
      throw test;           // throw int
    if (test == 1)
      throw 'a';            // throw char
    if (test == 2)
      throw 123.23;               // throw double
  }
  catch ( ...)
  {                               // catch all exceptions
    cout << "Caught One!\n";
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (0);
  Xhandler (1);
  Xhandler (2);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught One!
Caught One!
Caught One!
End
```

One very good use for catch (...) is as the last catch of a cluster of catches which catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

RETHROWING AN EXCEPTION

If you wish to rethrow an expression from within an exception handler, you may do so by calling throw, by itself, with no exception.

Reason: It allows multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate outward to the next catch statement.

Program:

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
```

```
void Xhandler ()
{
 try
 {
   throw "hello";                    // throw a char *
 }
 catch (const char *)
 {                                   // catch a char *
   cout << "Caught char * inside Xhandler\n";
   throw;                            // rethrow char * out of function
 }
}

int main ()
{
 cout << "Start\n";
 try
 {
   Xhandler ();
 }
 catch (const char *)
 {
   cout << "Caught char * inside main\n";
 }
 cout << "End";
 return 0;
}
```

Output:

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

EXCEPTION SPECICATION (RESTRICTING EXCEPTIONS)

You can restrict the type of exceptions that a function can throw outside of itself i.e. we are restricting a function to throw only certain specified exceptions .To accomplish these restrictions, we must add a throw clause to a function definition.

General form

ret-type func-name(*arg-list*) throw(*type-list*)
{
// ...
}

only those data types contained in the comma-separated *type-list* may be thrown by the function. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected ( )** to be called. By default, this causes **abort( )** to be called, which causes abnormal program termination.

Program:

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler (int test)
throw (int, char, double)
{
  if (test == 0)
    throw test;  // throw int
  if (test == 1)
    throw 'a';  // throw char
  if (test == 2)
    throw 123.23;    // throw double
}

int main ()
{
  cout << "start\n";
  try
  {
    Xhandler (0);        // also, try passing 1 and 2 to Xhandler()
  }
  catch (int i)
  {
    cout << "Caught an integer\n";
  }
  catch (char c)
  {
    cout << "Caught char\n";
  }
  catch (double d)
  {
    cout << "Caught double\n";
  }
  cout << "end";
  return 0;
}
```

Output:

```
start
Caught an integer
```

```
end
```

A function can be restricted only in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a function may throw any type of exception so long as it is caught *within* that function.The restriction applies only when throwing an exception outside of the function.

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
/* The following statements no longer work. Instead,
they will cause an abnormal program termination. */
if(test==0) throw test;
if(test==1) throw 'a';
if(test==2) throw 123.23;
}
```

STACK UNWINDING

Stack unwinding is a process of calling all destructors for all automatic objects constructed at run time when an exception is thrown. The objects are destroyed in the reverse order of their formation.

When an exception is thrown, the runtime mechanism first searches for an appropriate matching handler (catch) in the current scope. If no such handler exists, control is transferred from the current scope to a higher block in the calling chain or in outward manner. - Iteratively, it continues until an appropriate handler has been found. At this point, the stack has been unwound and all the local objects that were constructed on the path from a try block to a throw expression have been destroyed. - The run-time environment invokes destructors for all automatic objects constructed after execution entered the try block. This process of destroying automatic variables on the way to an exception handler is called stack unwinding.

Program:

```
#include <iostream>
#include <string>

using namespace std;

class MyClass
{
private:
  string name;
public:
  MyClass (string s):name (s)
  {
  }
   ~MyClass ()
  {
    cout << "Destroying " << name << endl;
  }
};

void fa ();
void fb ();
void fc ();
void fd ();

int main ()
{
  try
  {
    MyClass mainObj ("M");
    fa ();
    cout << "Mission accomplished!\n";
  }
  catch (const char *e)
  {
    cout << "exception: " << e << endl;
    cout << "Mission impossible!\n";
  }
  return 0;
}

void fa ()
{
  MyClass a ("A");
  fb ();
  cout << "return from fa()\n";
  return;
}

void fb ()
{
  MyClass b ("B");
  fc ();
  cout << "return from fb()\n";
  return;
}

void fc ()
{
  MyClass c ("C");
  fd ();
  cout << "return from fc()\n";
  return;

}

void fd ()
{
  MyClass d ("D");
  // throw "in fd(), something weird happened.";
  cout << "return from fd()\n";
  return;
}
```

Output:

```
return from fd()
Destroying D
return from fc()
Destroying C
return from fb()
Destroying B
return from fa()
Destroying A
Mission accomplished!
Destroying M
```

EXCEPTION OBJECT

The exception object holds the error information about the exception that had occurred. The information includes the type errors i.e. logic errors or run time error and state of the program when the error occurred.

An exception object is created as soon as exception occurs and it is passed to the corresponding catch block as a parameter. The catch block contains the code to catch the occurred exception.

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error.

General Form:

```
try
{
Throw exception object;
}
catch(Exception &exceptionobject)
{
```

```
…
}
```

When a throw expression is evaluated, an exception object is initialized from the value of the expression. The exception object which is thrown gets its type from the static type of the throw expression.

Inside a catch block, the name initialized with the caught exception object is initialized with this exception object

The exception object is available only in catch block. You cannot use the exception object outside the catch block.

Program:

```
#include <iostream>
#include <cstring>
using namespace std;
class MyException
{
public:
  char str_what[80];
  int what;
    MyException ()
  {
    *str_what = 0;
    what = 0;
  }
  MyException (char *s, int e)
  {
    strcpy (str_what, s);
    what = e;
  }
};

int main ()
{
  int i;
  try
  {
    cout << "Enter a positive number: ";
    cin >> i;
    if (i < 0)
      throw MyException ("Not Positive", i);
  }
  catch (MyException e)
  {                              // catch an error
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }
  return 0;
}
```

Output:

```
Enter a positive number: -1
Not Positive: -1
```