

Chapter-1

Concept of Object Oriented Programming

PROCEDURAL ORIENTED PROGRAMMING (pop):-

A program in a procedural language is a list of instruction where each statement tells the computer to do something. It focuses on procedure (function) & algorithm is needed to perform the derived computation.

When program become larger, it is divided into function & each function has clearly defined purpose. Dividing the program into functions & module is one of the cornerstones of structured programming.

E.g.:- c, basic, FORTRAN.

Characteristics of Procedural oriented programming:-

- It focuses on process rather than data.
- It takes a problem as a sequence of things to be done such as reading, calculating and printing. Hence, a number of functions are written to solve a problem.
- A program is divided into a number of functions and each function has clearly defined purpose.
- Most of the functions share global data.
- Data moves openly around the system from function to function.

Drawback of Procedural oriented programming (structured programming):-

- It emphasis on doing things. Data is given a second class status even through data is the reason for the existence of the program.
- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.
- It is often difficult to design because the components function and data structure do not model the real world.

For example, in designing graphics user interface, we think what functions, what data structures are needed rather than which menu, menu item and so on.

- It is difficult to create new data types. The ability to create the new data type of its own is called extensibility. Structured programming languages are not extensible.

To overcome the limitation of Procedural oriented programming languages Object oriented programming languages were developed.

OBJECT ORIENTED PROGRAMMING

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.

In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

Object Oriented Programming follows bottom up approach in program design and emphasizes on safety and security of data..

FEATURES OF OBJECT ORIENTED PROGRAMMING:

Inheritance:

- Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class.
- Derived class is also known as a child class or sub class. Inheritance helps in reusability of code , thus reducing the overall size of the program

Data Abstraction:

It refers to the act of representing essential features without including the background details .Example : For driving , only accelerator, clutch and brake controls need to be learnt rather than working of engine and other details.

Data Encapsulation:

- It means wrapping up data and associated functions into one single unit called class.
- A class groups its members into three sections :public, private and protected, where private and protected members remain hidden from outside world and thereby helps in implementing data hiding.

Modularity :

- The act of partitioning a complex program into simpler fragments called modules is called as modularity.
- It reduces the complexity to some degree and
- It creates a number of well-defined boundaries within the program .

Polymorphism:

- Poly means many and morphs mean form, so polymorphism means one name multiple forms.
- It is the ability for a message or data to be processed in more than one form.
- C++ implements Polymorphism through Function Overloading, Operator overloading and Virtual functions.

Compare procedure oriented and object oriented programming

BASIS FOR COMPARISON	POP	OOP
Basic	Procedure/Structure oriented .	Object oriented.
Approach	Top-down.	Bottom-up.
Basis	Main focus is on "how to get the task done" i.e. on the procedure or structure of a program .	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Entity accessing mode	No access specifier observed.	Access specifier are "public", "private", "protected".
Overloading/Polymorphism	Neither it overload functions nor operators.	It overloads functions, constructors, and operators.
Inheritance	Their is no provision of inheritance.	Inheritance achieved in three modes public private and protected.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected. hence data security increases.

BASIS FOR COMPARISON	POP	OOP
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Friend functions/classes	No concept of friend function.	Classes or function can become a friend of another class with the keyword "friend". Note: "friend" keyword is used only in c++
Virtual classes/ function	No concept of virtual classes .	Concept of virtual function appear during inheritance.
Example	C, VB, FORTRAN, Pascal	C++, JAVA, VB.NET, C#.NET.

Benefits of OOPs:

1. **Simplicity:** software objects model real world objects, so the complexity is reduced and the program structure is very clear;
2. **Modularity:** each object forms a separate entity whose internal workings are decoupled from other parts of the system;
3. **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;
4. **Extensibility:** adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones;
5. **Maintainability:** objects can be maintained separately, making locating and fixing problems easier;
6. **Re-usability:** objects can be reused in different programs.

APPLICATION OF OOPs.

1. Client-Server Systems

Object-oriented Client-Server Systems provide the IT infrastructure, creating object-oriented Client-Server Internet (OCSI) applications. Here, infrastructure refers to operating systems, networks, and hardware. OSCI consist of three major technologies:

- The Client Server
- Object Oriented Programming
- The Internet

2. Object Oriented Databases

They are also called Object Database Management Systems (ODBMS). These databases store objects instead of data, such as real numbers and integers. Objects consist of the following:

Attributes: Attributes are data that defines the traits of an object. This data can be as simple as integers and real numbers. It can also be a reference to a complex object.

Methods: They define the behavior and are also called functions or procedures.

3. Object Oriented Databases

These databases try to maintain a direct correspondence between the real-world and database objects in order to let the object retain their identity and integrity. They can then be identified and operated upon.

4. Real-Time System Design

Real time systems inherit complexities that makes difficult to build them. Object-oriented techniques make it easier to handle those complexities. These techniques present ways of dealing with these complexities by providing an integrated framework which includes schedulability analysis and behavioral specifications.

5. Simulation And Modelling System

It's difficult to model complex systems due to the varying specification of variables. These are prevalent in medicine and in other areas of natural science, such as ecology, zoology, and agronomic systems. Simulating complex systems requires modelling and understanding interactions explicitly. Object-oriented Programming provides an alternative approach for simplifying these complex modelling systems.

6. Hypertext And Hypermedia

OOP also helps in laying out a framework for Hypertext. Basically, hypertext is similar to regular text as it can be stored, searched, and edited easily. The only difference is that hypertext is text with pointers to other text as well.

Hypermedia, on the other hand, is a superset of hypertext. Documents having hypermedia, not only contain links to other pieces of text and information, but also to numerous other forms of media, ranging from images to sound.

7. Neural Networking And Parallel Programming

It addresses the problem of prediction and approximation of complex time-varying systems. Firstly, the entire time-varying process is split into several time intervals or slots. Then, neural networks are developed in a particular time interval to disperse the load of various networks. OOP simplifies the entire process by simplifying the approximation and prediction ability of networks.

8. Office Automation Systems

These include formal as well as informal electronic systems primarily concerned with information sharing and communication to and from people inside as well as outside the organization. Some examples are:

- Email
- Word processing
- Web calendars
- Desktop publishing

9. CIM/CAD/CAM Systems

OOP can also be used in manufacturing and design applications as it allows people to reduce the effort involved. For instance, it can be used while designing blueprints, flowcharts, etc. OOP makes it possible for the designers and engineers to produce these flowcharts and blueprints accurately.

10. AI Expert Systems

These are computer applications which are developed to solve complex problems pertaining to a specific domain, which is at a level far beyond the reach of a human brain.

It has the following characteristics:

- Reliable
- Highly responsive
- Understandable
- High-performance

Definition - What does C++ Programming Language mean?

C++ is a general-purpose object-oriented programming (OOP) language, developed by Bjarne Stroustrup, and is an extension of the C language. It is therefore possible to code C++ in a "C style" or "object-oriented style." In certain scenarios, it can be coded in either way and is thus an effective example of a hybrid language.

C++ is considered to be an intermediate-level language, as it encapsulates both high- and low-level language features. Initially, the language was called "C with classes" as it had all the properties of the C language with an additional concept of "classes." However, it was renamed C++ in 1983. It is pronounced "see-plus-plus". It is a case sensitive language.

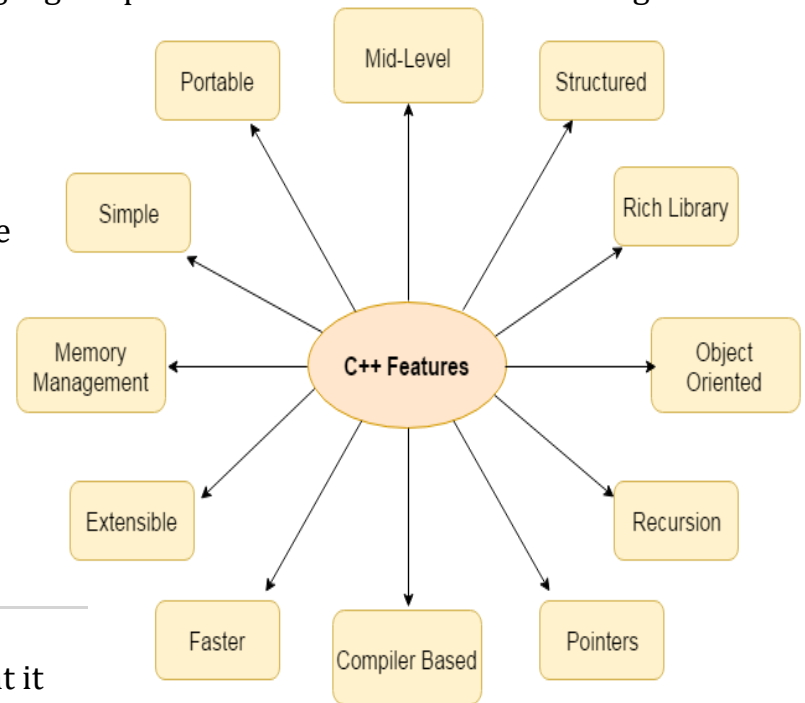
C vs C++

No.	C	C++
1)	C follows the procedural style programming .	C++ is multi-paradigm. It supports both procedural and object oriented .
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the top-down approach .	C++ follows the bottom-up approach .
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into procedures and modules	C++ programs are divided into functions and classes .
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.

C++ Features

C++ is object oriented programming language. It provides a lot of **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. Structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible
11. Object Oriented
12. Compiler based

**1) Simple**

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler based

C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

Basic Structure of C++ Program with Classes:

A C++ program can be developed from a basic structure. The general structure of C++ program with classes is as shown below (which is also called overview of a C++ program):

1. Documentation Section
2. Preprocessor Directives or Compiler Directives Section
 - (i) Link Section
 - (ii) Definition Section
3. Global Declaration Section
4. Class declaration or definition
5. Main C++ program function called main ()
6. Beginning of the program: Left brace {
 - (i) Object declaration part;
 - (ii) Accessing member functions (using dot operator);
7. End of the main program: Right brace }

Chapter-2 OBJECTS & CLASSES

The major components of Object Oriented Programming are Classes & Objects. A Class is a group of similar objects. Objects share two characteristics: They all have state and behavior. For example : Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, applying brakes). Identifying the state and behavior for realworld

objects is a great way to begin thinking in terms of object-oriented programming. These real-world observations all translate into the world of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through functions.

Classes in Programming :

It is a collection of variables, often of different types and its associated functions. Class just binds data and its associated functions under one unit there by enforcing encapsulation.

Classes define types of data structures and the functions that operate on those data structures. A class defines a blueprint for a data type.

Declaration/Definition :

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations.

```
class class_name {
access_specifier_1:
member1;
access_specifier_2:
member2;
...
...
}
```

...

} object_names;

Where *class_name* is a valid identifier for the class, *object_names* is an optional list of names for objects of this class. The body of the declaration can contain members that can either be data or function declarations, and optionally access specifiers.

[Note: the default access specifier is private.]

Access specifiers in Classes:

Access specifiers are used to identify access rights for the data and member functions of the class.

There are three main types of access specifiers in C++ programming language:

private

public

protected

Private: Class members declared as private can be used only by member functions and friends (classes or functions) of the class.

Protected: Class members declared as protected can be used by member functions and friends (classes or functions) of the class. Additionally, they can be used by classes derived from the class.

Public: Class members declared as public can be used by any function.

Importance of Access Specifiers

Access control helps prevent you from using objects in ways they were not intended to be used. Thus it helps in implementing data hiding and data abstraction.

OBJECTS in C++:

Objects represent instances of a class. Objects are basic run time entities in an object oriented system.

Creating object / defining the object of a class:

The general syntax of defining the object of a class is:-

Class_name object_name;

In C++, a class variable is known as an object. The declaration of an object is similar to that of a variable of any data type. The members of a class are accessed or referenced using object of a class.

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing / calling members of a class

All member of a class are private by default. Private member can be accessed only by the function of the class itself. Public member of a class can be accessed through any object of the class. They are accessed or called using object of that class with the help of dot operator (.).

The general syntax for accessing data member of a class is:-

Object_name.Data_member=value;

The general syntax for accessing member function of a class is:-

Object_name.Function_name (actual arguments);

The dot ('.') used above is called the dot operator or class member access operator. The dot operator is used to connect the object and the member function. The private data of a class can be accessed only through the member function of that class.

Class methods definitions (Defining the member functions)

Member functions can be defined in two places:-

- **Outside the class definition**

The member functions of a class can be defined outside the class definitions. It is only declared inside the class but defined outside the class. The general form of member function definition outside the class definition is:

```
Return_type Class_name:: function_name (argument list)
{
    Function body
}
```

Where symbol :: is a scope resolution operator.

The scope resolution operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition.

- **Inside the class definition**

The member function of a class can be declared and defined inside the class definition.

```
class sum
{
    int A, B, Total;
    public:
    void getdata ()
    { cout<<"\n
    enter the value of A
    and B";
    cin>>A>>B;
    }
    void display ()
    {
    total = A+B;
    cout<<"\n the sum of A and B="<<total;
    }
};
```

Inline function

The functions can be made inline by adding prefix inline to the function definition.

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

The compiler replaces the function call with the corresponding function code.

Inline function saves time of calling function, saving registers, pushing arguments onto the stack and returning from function.

We should be careful while using inline function. If function has 1 or 2 lines of code and simple expressions then only it should be used.

Inline expansion may not work in following situations,

If a loop, a switch or a goto exists in function body.

For function is not returning any value, if a return statement exists.

If function contains static variables.

If function is recursive.

Example:

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

```
inline int cube(int n)
{
```

```
return n*n*n;
}
int main()

{
int c;
c = cube(10);
cout<<c;
return 0;

}
```

Function call is replaced with expression so `c = cube(10);` becomes `c=10*10*10;` at compile time.

Disadvantage:

It makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called.

Object as function Arguments:-An object can be passed as an argument/parameter to functions in two ways.

1. **pass by value**

2. **pass by reference.**

Example:

```
#include< iostream.h>
#include< conio.h>
class height
{
int feet,inches;
public:
void getht(int f,int i)
{
feet=f;
inches=i;
}
void putheight()
{
cout<< "\nHeight is:"<< feet<< "feet\t"<< inches<< "inches"<< endl;
}
void sum(height a,height b)// object as argument
{
height n;
n.feet = a.feet + b.feet;
n.inches = a.inches + b.inches;
if(n.inches >=12)
{
n.feet++;
n.inches = n.inches -12;
}
cout<< endl<< "Height is "<< n.feet<< " feet and "<< n.inches<< endl;
```

```

});
void main()
{height h,d,a;
clrscr();
h.getht(6,6);
a.getht(2,9);
h.putheight();
a.putheight();
d.sum(h,a);
getch();
}
/*****OUTPUT*****/
Height is:6feet 6inches
Height is:2feet 9inches
Height is 9 feet and 3 inches

```

Array of objects :

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.

The syntax for declaring an array of objects is

```
class_name array_name [size] ;
```

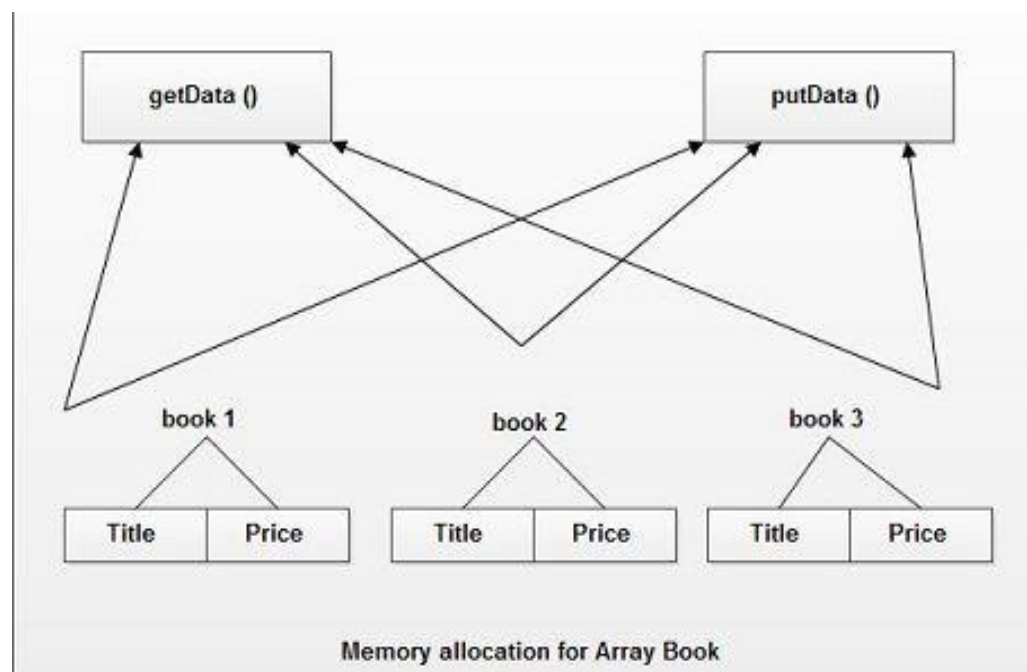
To understand the concept of an array of objects, consider this example.

Example : A program to demonstrate the concept of array of objects

```

#include<iostream>
using namespace std;
class books
{
char tit1e [30];
float price ;
public:
void getdata ();
void putdata ();
};
void books :: getdata ()
{
cout<<"Title:";
Cin>>title;
cout<<"Price:";
cin>>price;
}
void books :: putdata ()
{
cout<<"Title:"<<title<< "\n";
cout<<"Price:"<<price<< "\n";
const int size=3 ;
int main ()
{

```



```

books book[size] ;
for(int i=0;i<size;i++)
{
cout<<"Enter details of book "<<(i+1)<<"\n";
book[i].getdata();
}
for(int i=0;i<size;i++)
{
cout<<"\nBook "<<(i+1)<<"\n";
book[i].putdata() ;
}
return 0;
}

```

In this example, an array book of the type class books and size three is declared. This implies that book is an array of three objects of the class books. Note that every object in the array book can access public members of the class in the same way as any other object, that is, by using the dot operator. For example, the statement book [i] . getdata () invokes the getdata () function for the ith element of array book.

When an array of objects is declared, the memory is allocated in the same way as to multidimensional arrays. For example, for the array book, a separate copy of title and price is created for each member book[0], book[1] and book[2]. However, member functions are stored at a different place in memory and shared among all the array members. For instance, the memory space is allocated to the the array of objects book of the class books,

Friend Function:- A friend function is not a member function ,has full access rights to the private members of the class.

Syntax :-

```
class ABC
```

```
{
```

```
-----
```

```
-----
```

```
public:
```

```
-----
```

```
-----
```

```
friend void xyz(void);
```

```
}
```

Characteristics of friend function:

- 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 the private part of a class without affecting its meaning.

Uaually ,it has the objects as arguments.

Example:-

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

```
#include<conio.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;
}
void main()
{ sample x;
x.setvalue();
cout<<mean(x);
}
// A function friendly to two classes
#include<iostream.h>
class XYZ
{
int x;
public:
void setvalue(int i)
{
x=i;
}
friend void max(XYZ,ABC);
};
class ABC
{
int a;
public:
void setvalue(int i)
{
a=i;
}
friend void max(XYZ,ABC);
};
void max(XYZ m, ABC n)
{
if (m.x>=n.a)
cout<<m.x;
else
cout<<n.x;
}
void main()
{
ABC abc;
```

```

abc.setvalue(10);
XYZ xyz;
xyz.setvalue(20);
max(xyz,abc);
}

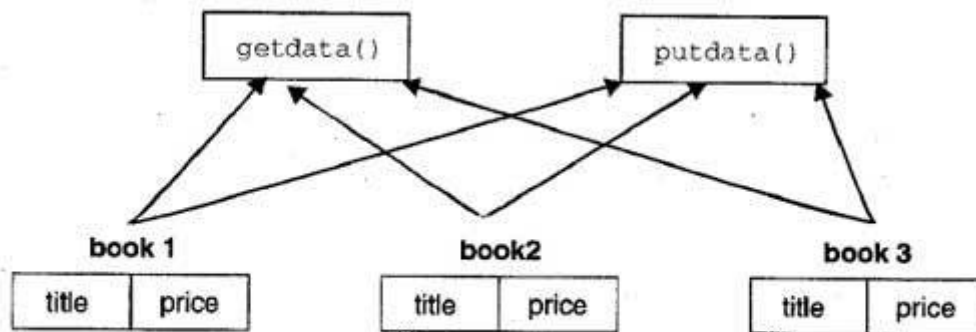
```

Memory Allocation for Objects

Before using a member of a class, it is necessary to allocate the required memory space to that member. The way the memory space for data members and member functions is allocated is different regardless of the fact that both data members and member functions belong to the same class.

The memory space is allocated to the data members of a class only when an object of the class is declared, and not when the data members are declared inside the class. Since a single data member can have different values for different objects at the same time, every object declared for the class has an individual copy of all the data members.

On the other hand, the memory space for the member functions is allocated only once when the class is defined. In other words, there is only a single copy of each member function, which is shared among all the objects. For instance, the three objects, namely, book1, book2 and book3 of the class book have individual copies of the data members title and price. However, there is only one copy of the member functions getdata () and putdata () that is shared by all the three objects



Memory Allocation for the Objects of the Class book

Function overloading with example.

Function overloading is compile time polymorphism.

Function overloading is the practice of declaring the same function with different signatures. The same function name will be used with different number of parameters and parameters of different type.

Overloading of functions with different return types is not allowed.

Compiler identifies which function should be called out of many using the type and number of arguments.

A function is overloaded when same name is given to different functions. However, the two functions with the same name must differ in at least one of the following,

- a) **The number of parameters**
- b) **The data type of parameters**
- c) **The order of appearance**

Example:

```

#include <iostream.h>
void Add(int num1, int num2)\\ Function 1: Receives 2 integer parameters
{

```

```

        cout<<num1 + num2 <<endl;
    }
    void Add(float num1, float num2)\\Function 2: Receives 2 float parameters.
    {
        cout<<num1 + num2 <<endl;
    }
    void Add(int num1, int num2, int num3)\\Function 3: Receives 3 int parameters
    {
        cout<<num1 + num2 + num3 <<endl;
    }
    int main()
    {
        float a=10.5,b=20.5;
        Add(10,20);        \\ Calls function 1
        Add(a,b);          \\ Calls function 2
        Add(1,2,3);        \\ Calls function 3
        return 0;
    }

```

Static Members of a C++ Class

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator `::` to identify which class it belongs to.

Let us try the following example to understand the concept of static data members –

```

#include <iostream>

using namespace std;

class Box {
public:
    static int objectCount;

    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout <<"Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;

        // Increase every time object is created
        objectCount++;
    }
    double Volume() {
        return length * breadth * height;
    }
}

```

```
private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
    Box Box1(3.3, 1.2, 1.5); // Declare box1
    Box Box2(8.5, 6.0, 2.0); // Declare box2

    // Print total number of objects.
    cout << "Total objects: " << Box::objectCount << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator `::`.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members –

```
#include <iostream.h>

class Box {
public:
    static int objectCount;

    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;

        // Increase every time object is created
        objectCount++;
    }
};
```

```
}  
double Volume() {  
    return length * breadth * height;  
}  
static int getCount() {  
    return objectCount;  
}  
  
private:  
    double length;    // Length of a box  
    double breadth;   // Breadth of a box  
    double height;    // Height of a box  
};  
  
// Initialize static member of class Box  
int Box::objectCount = 0;  
  
int main(void) {  
    // Print total number of objects before creating object.  
    cout << "Initial Stage Count: " << Box::getCount() << endl;  
  
    Box Box1(3.3, 1.2, 1.5); // Declare box1  
    Box Box2(8.5, 6.0, 2.0); // Declare box2  
  
    // Print total number of objects after creating object.  
    cout << "Final Stage Count: " << Box::getCount() << endl;  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

```
Initial Stage Count: 0  
Constructor called.  
Constructor called.  
Final Stage Count: 2
```

Chapter-3**Constructor in C++**

Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. A constructor is a special member function which is used to initialize the object of a class.

Note:-

- A constructor is distinct from other member functions because its name is same as the name of the class. It is executed automatically when an object is declared. The only restriction that applies to constructor is that it must not have a return type not even void.
- Constructor of a class is the first member function to be executed automatically when an object of the class is created.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).
- If a class has a constructor, each object of that class will be initialized before any use is made of the object.
- They cannot be inherited through a derived class can call the base class constructor
- A constructor may not be static.
- It is not possible to take the address of a constructor
- Member functions may be called from within a constructor.

The general syntax:

```
class classname
{
    -----
    -----
    -----
    classname( parameter list);
    -----
};
class name::classname(parameter list)
{
    -----
}
```

Example:

```
class Bank
{
    private:
    int x,y;
    -----
    -----
    public:
    Bank( );
};
Bank :: Bank( )
{
    x=0;
    y=0;
}
void main()
```

```
{
    Bank B;
}
```

When an object B is created ,constructor will be called automatically and it will initialize the data members x and y=0.

Example:-

```
#include<iostream.h>
#include<conio.h>
class fibonacci
{
    private:
    int fo,f1,fib;
    public:
    fibonacci();
    void increment();
    void display();
};
fibonacci::fibonacci()
{
    fo=0;
    f1=1;
    fib=fo+f1;
}
void fibonacci::increment()
{
    fo=f1;
    f1=fib;
    fib=fo+f1;
}
void fibonacci:: display()
{
    cout<< fib<<"\t";
}
void main()
{
    clrscr();
    fibonacci F;
    for(int i=0;i<=15;++i)
    {
        F.display();
        F.increment();
        getch();
    }
}
```

Types of Constructors in C++

- **Default constructor**
- **Parameterized Constructors**
- **Copy Constructor**

1. **Default Constructors:** Default constructor is the constructor which doesn't take any argument. It . has no parameters.

```
#include <iostream.h>
class construct {
public:
```

```

int a, b;

// Default Constructor
construct()
{
    a = 10;
    b = 20;
}

};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}

```

Output:

```

a: 10
b: 20

```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. **Parameterized constructor:-** The constructor that can take arguments are called Parameterized constructor. It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Example:

```

class Bank
{
    private:
    int x,y;
    public:
    Bank(int a, int b)
    -----
    -----
};

Bank:: Bank(int a, int b)
{
    x=a;
    y=b;
}

```

In the parameterized constructor we pass the initial values as arguments to the constructor in two ways:

i) Implicit call	ii) Explicit call
Implicit call: Bank b1(5,11);	Explicit Call: Bank B1=Bank(5,11);

Example:-

```

#include<iostream.h>
#include<conio.h>
class Bank
{
    int x,y;
    public:
    Bank(int ,int);
}

```

```

        void putdata()
        {
            cout<<x<<"\n"<<y;
        }
};
Bank :: Bank(int a,int b);
{
    x=a;
    y=b;
}
void main()
{
    clrscr();
    Bank B1(5,11);
    Bank B2=Bank(6,34);
    cout<<" object 1";
    B1.putdata();
    cout<<"object 2";
    B2.putdata();
    getch();
}

```

Uses of Parameterized constructor:

- It is used to initialize the various data elements of different objects with different values when they are created.
 - It is used to overload constructors.
3. **Copy Constructor:-** A copy constructor is a member function which initializes an object using another object of the same class. In other word, a copy constructor takes a reference to its own class as parameter i.e. a constructor having a reference to an instance of its own class as an argument is known as copy constructor.

A copy constructor has the following general syntax:

```
ClassName (ClassName &old_obj);
```

In C++, a Copy Constructor may be called in following cases:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.

Copy constructor are called in following style:

Sample S1; //default constructor

Sample S2(S1);

or

Sample S2=S1;

Example:

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

```
class sample
```

```

{
    int x;
    public:
    sample()
    {
        x=0;
    }
    sample(int a)

```

```

        {
            x=a;
        }
sample (sample & S)
{
    x=S.x
}
void display(void)
{
    cout<<x;
}
};
void main()
{
    clrscr();
    Sample P(45);
    Sample Q(P);
    Sample R=P;
    Sample T;
    T=P;
    cout<<" \n value of p:";
    P.display();
    cout<<"\n value of Q:";
    Q.display();
    cout<<" \n value of R";
    R.display();
    cout<<"\nValue of T:";
    T.display();
    getch();
}

```

Constructor Overloading in C++

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter. Likewise function overloading, a class can have more than one constructor. Since more than one constructor is defined in a class it is called **c++ constructor overloading**.

Every constructor has same name as class name but they differ in terms of either number of arguments or the datatypes of the arguments or the both. As there is more than one constructor in class it is also called **multiple constructor**.

You can have any number of Constructors in a class that differ in parameter list.

```

class Student
{
    public:
    int rollno;
    string name;
    // first constructor
    Student(int x)
    {
        rollno = x;
        name = "None";
    }
    // second constructor
    Student(int x, string str)
    {

```

```

    rollno = x;
    name = str;
}
};

int main()
{
    // student A initialized with roll no 10 and name None
    Student A(10);

    // student B initialized with roll no 11 and name John
    Student B(11, "John");
}

```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

Constructor with default Arguments

Like all functions, a constructor can have default arguments. They are used to initialize member objects. If default values are supplied, the trailing arguments can be omitted in the expression list of the constructor.

Example:

```

item (int x,int y=0); // declaration
item:: item(int x,int y)
{
    a=x;
    b=y;
}

```

when we make object of item

```

item P(11);
11 will be assign to x and 0 to y;
item Q(11,17);
11 to x and 17 to y;

```

C++ Destructors

C++ destructor is a special member function that is executed automatically when an object is destroyed that has been created by the constructor. C++ destructors are used to de-allocate the memory that has been allocated for the object by the constructor. In other words a destructor is also a member function whose name is the same as the class name but is **preceded by tilde ('~')**

A destructor takes no arguments and no return types can be specified for it. It is called automatically by the compiler when an object is destroyed. A destructor cleans up the storage (memory area of the object) that is no longer accessible.

Need for Destructors

Allocated resources (like constructor) must be de-allocated before the object is destroyed. It works as a de-allocating and releasing memory area and **it perform our works as a clean up tasks**. Therefore, a destructor is equally useful as a constructor is.

Its syntax is same as constructor except the fact that it is preceded by the tilde sign.

```
~class_name() { }; //syntax of destructor
```

```
class stud
```

```
{
stud() //constructor
{
cout << " welcome" 20
}
~stud() //destructor
{
}
};
stud s1;
```

Note:- Object Are Destroyed In The Reverse Order Of Creation. So destructor is invoked in the reverse order of constructor.

Example:-

```
#include<iostream.h>
#include<conio.h>
class sample
{
public:
sample()
{
cout<<" object is born\n";
}
~sample ()
{
cout <<" object is dead"
}
};
void main()
{
clrscr();
sample S;
cout<<" Main terminated "<<endl;
getch();
}
```

Chapter-4

Inheritance in C++

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

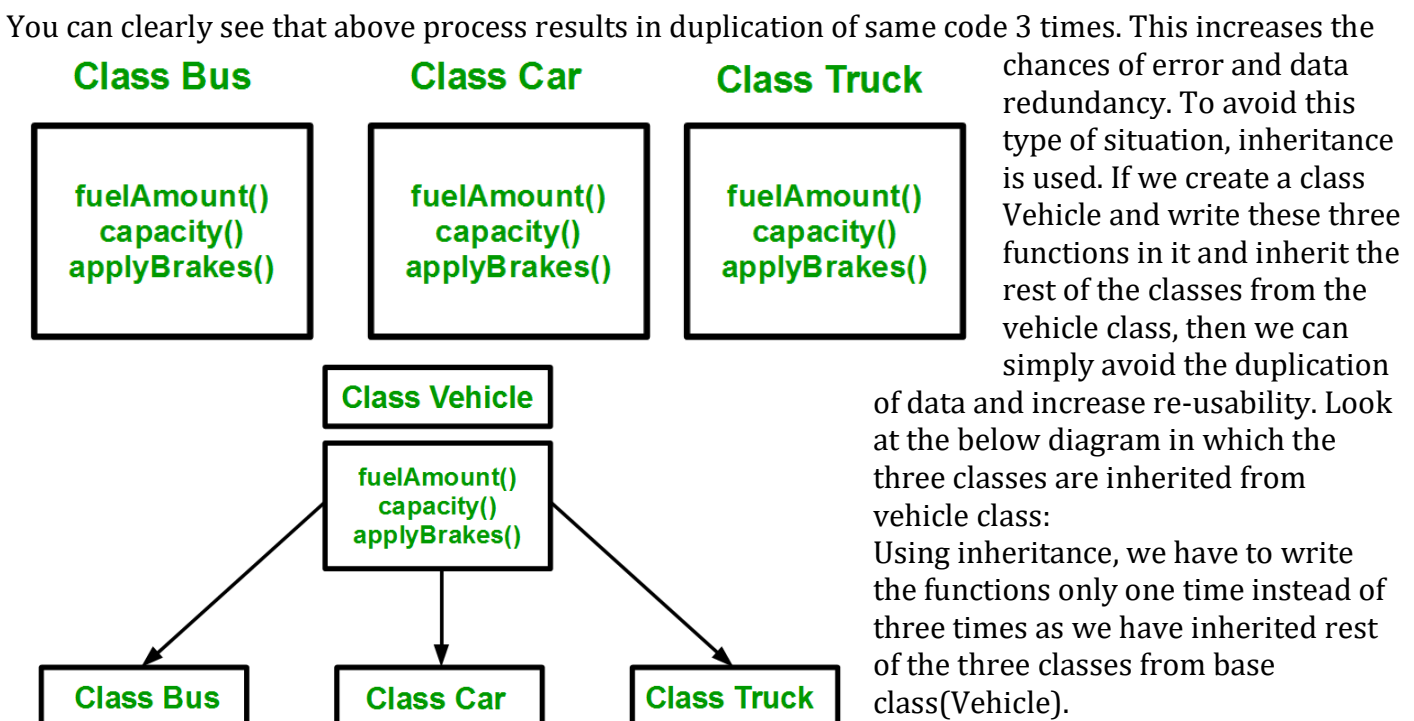
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Features or Advantages of Inheritance:

- *Reusability of Code*
- *Saves Time and Effort*
- *Faster development, easier maintenance and easy to extend*
- *Capable of expressing the inheritance relationship and its transitive nature which ensures closeness with real world problems.*

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods `fuelAmount()`, `capacity()`, `applyBrakes()` 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:



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 subclass_name : access_mode base_class_name
{
    //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want 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. In C++, the default mode of visibility is private. The private members of the base class are never inherited.

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
```

```
class A
```

```
{
public:
    int x;
protected:
    int y;
private:
    int z;
```

```
};
```

```
class B : public A
```

```
{
    // 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
};

```

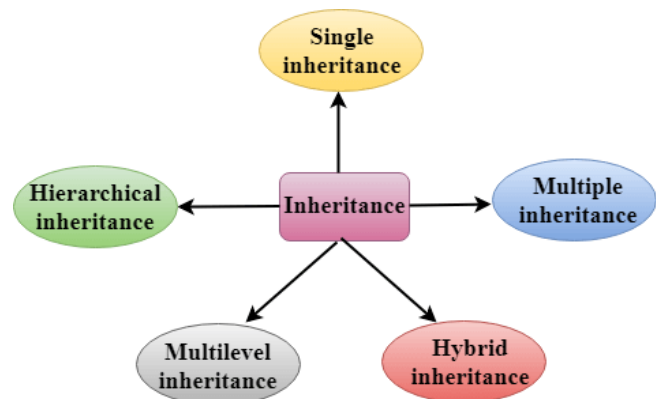
The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

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

Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.

Where 'A' is the base class, and 'B' is the derived class.

Example.

```

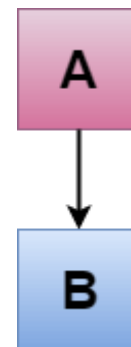
#include <iostream>
using namespace std;
class A
{
    int a = 4;
    int b = 5;
public:
    int mul()
    {
        int c = a*b;
        return c;
    }
};

```

```

class B : private A
{
public:

```



```
void display()
{
    int result = mul();
    std::cout << "Multiplication of a and b is : " << result << std::endl;
}
};
int main()
{
    B b;
    b.display();

    return 0;
}
```

Output:

Multiplication of a and b is : 20

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
#include <iostream>
using namespace std;
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
class Dog: public Animal
{
public:
    void bark(){
        cout << "Barking..." << endl;
    }
};
class BabyDog: public Dog
{
public:
    void weep() {
        cout << "Weeping...";
    }
}
```



```
};
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
    d1.weep();
    return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

Syntax of the Derived class:

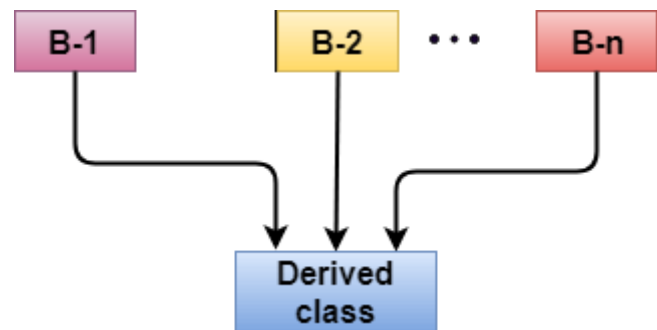
```
class D : visibility B-1, visibility B-2, ?
{
    // Body of the class;
}
```

Let's see a simple example of 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
```

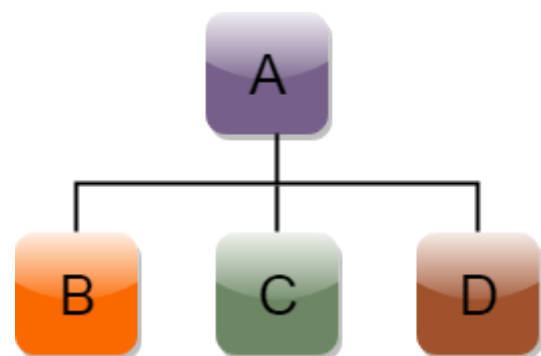
In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

Syntax of Hierarchical inheritance:

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```



Let's see a simple example:

```
#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

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

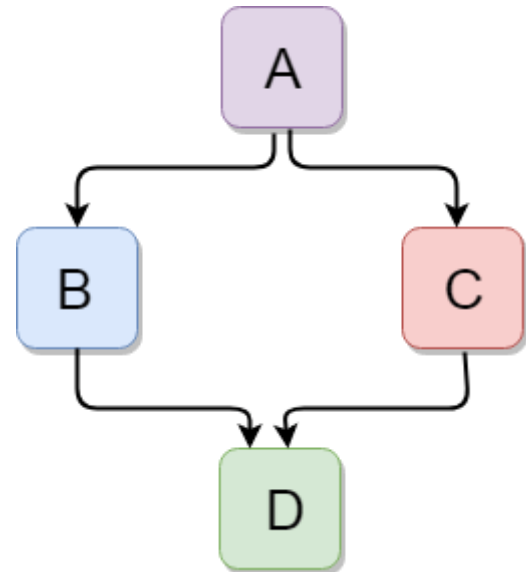
Let's see a simple example:

```
#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

```

Virtual base class in C++

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 class 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++

```

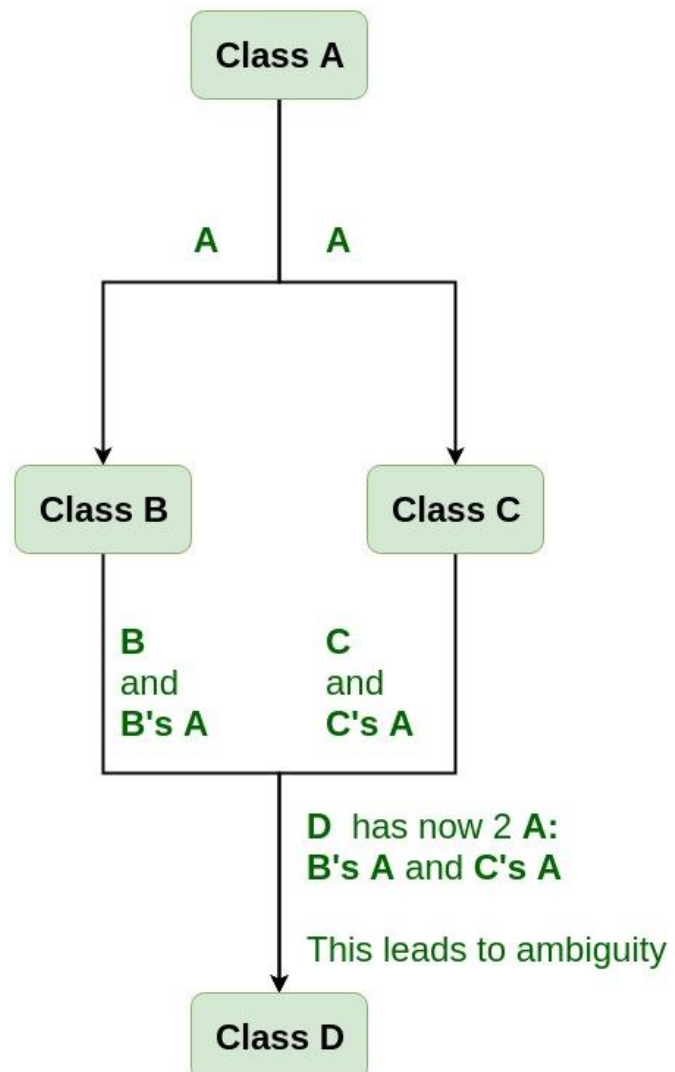
#include <iostream>
using namespace std;

```

```

class A {
public:

```



```
void show()
{
    cout << "Hello form A \n";
}
};
```

```
class B : public A {
};
```

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

```
class D : public B, public C {
};
```

```
int main()
{
    D object;
    object.show();
}
```

Compile Errors:

prog.cpp: In function 'int main()':

prog.cpp:29:9: error: request for member 'show' is ambiguous

```
    object.show();
        ^
```

prog.cpp:8:8: note: candidates are: void A::show()

```
    void show()
        ^
```

prog.cpp:8:8: note: void A::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 base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

Syntax 1:

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

Syntax 2:

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

Note: **virtual** can be 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 that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example 1

```
#include <iostream>
class A {
```

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

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

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

    return 0;
}
```

Output:

```
a = 10
```

Explanation : The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

Example 2:

```
#include <iostream>
class A {
public:
    void show()
    {
        cout << "Hello from A \n";
    }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

Output:

Hello from A

Pure Virtual Functions and Abstract Classes in C++

Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

A complete example:

A pure virtual function is implemented by classes which are derived from a Abstract class. Following is a simple example to demonstrate the same.

```
#include<iostream>
class Base
{
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Output:

fun() called

Some Interesting Facts:

1) A class is abstract if it has at least one pure virtual function.

In the following example, Test is an abstract class because it has a pure virtual function show().

```
// pure virtual functions make a class abstract
#include<iostream>
class Test
{
    int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};
```

```
int main(void)
{
    Test t;
    return 0;
}
```

Output:

Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the following virtual functions are pure within 'Test': note: virtual void Test::show()

2) *We can have pointers and references of abstract class type.*

For example the following program works fine.

```
#include<iostream>
class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};
```

```
int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Output:

In Derived

3) *If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.*

The following example demonstrates the same.

```
#include<iostream>
class Base
{
public:
    virtual void show() = 0;
};
```

```
class Derived : public Base {};
```

```
int main(void)
{
    Derived d;
    return 0;
}
```

Compiler Error: cannot declare variable 'd' to be of abstract type
'Derived' because the following virtual functions are pure within
'Derived': virtual void Base::show()

4) An abstract class can have constructors.

For example, the following program compiles and runs fine.

```
#include<iostream>
// An abstract class with constructor
class Base
{
protected:
    int x;
public:
    virtual void fun() = 0;
    Base(int i) { x = i; }
};

class Derived: public Base
{
    int y;
public:
    Derived(int i, int j):Base(i) { y = j; }
    void fun() { cout << "x = " << x << ", y = " << y; }
};

int main(void)
{
    Derived d(4, 5);
    d.fun();
    return 0;
}
```

Output:

```
x = 4, y = 5
```

Nested Classes in C++

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

For example, program 1 compiles without any error and program 2 fails in compilation.

Program 1

```
#include<iostream>
/* start of Enclosing class declaration */
class Enclosing {
private:
    int x;

    /* start of Nested class declaration */
```

```

class Nested {
    int y;
    void NestedFun(Enclosing *e) {
        cout<<e->x; // works fine: nested class can access
                    // private members of Enclosing class
    }
}; // declaration Nested class ends here
}; // declaration Enclosing class ends here

```

```

int main()
{
}

```

C++ 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.

Requirements for Overriding a Function

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

C++ Function Overriding Example

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...

Chapter-5

Polymorphism

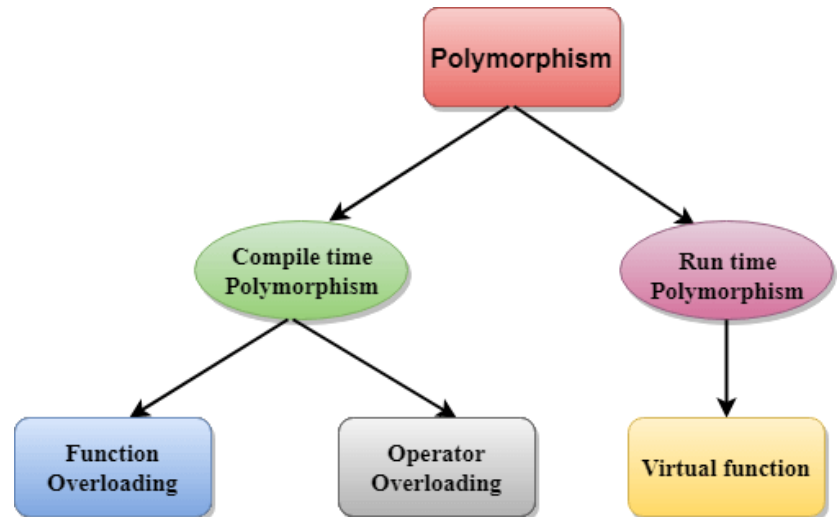
The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour 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.



```

1. class A                // base class declaration.
2. {
3.     int a;
4.     public:
5.     void display()
6.     {
7.         cout<< "Class A ";
8.     }
9. };
10. class B : public A    // derived class declaration.
11. {
12.     int b;
13.     public:
14.     void display()
15.     {
16.         cout<<"Class B";
17.     }
18. };
  
```

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.

Function overloading with example.

Function overloading is compile time polymorphism. Function overloading is the practice of declaring the same function with different signatures.

The same function name will be used with different number of parameters and parameters of different type. Overloading of functions with different return types is not allowed.

Compiler identifies which function should be called out of many using the type and number of arguments.

A function is overloaded when same name is given to different functions. However, the two functions with the same name must differ in at least one of the following,

d) The number of parameters

e) The data type of parameters

f) The order of appearance

Example:

```
#include <iostream.h>
void Add(int num1, int num2)\\ Function 1: Receives 2 integer parameters
{
    cout<<num1 + num2 <<endl;
}
void Add(float num1, float num2)\\Function 2: Receives 2 float parameters.
{
    cout<<num1 + num2 <<endl;
}
void Add(int num1, int num2, int num3)\\Function 3: Receives 3 int parameters
{
    cout<<num1 + num2 + num3 <<endl;
}
int main()
{
```

```

float a=10.5,b=20.5;
Add(10,20);          \\ Calls function 1
Add(a,b);             \\ Calls function 2
Add(1,2,3);           \\ Calls function 3
return 0;
}

```

Ambiguity in Function Overloading

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

- **Type conversion**

Let's see a simple example.

```

#include<iostream.h>
void fun(int);
void fun(float);
void fun(int i)
{
std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
fun(12);
fun(1.2);
return 0;
}

```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

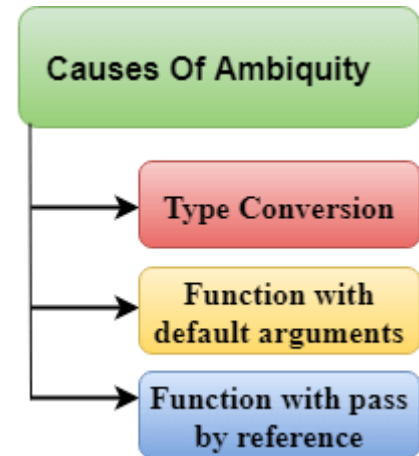
- **Function with Default Arguments**

Let's see a simple example.

```

#include<iostream.h>
void fun(int);
void fun(int,int);
void fun(int i)
{
std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
std::cout << "Value of a is : " <<a<< std::endl;
std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
fun(12);
}

```



```
return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

○ **Function with pass by reference**

Let's see a simple example.

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

```
void fun(int);
```

```
void fun(int &);
```

```
int main()
```

```
{
```

```
int a=10;
```

```
fun(a); // error, which f()?
```

```
return 0;
```

```
}
```

```
void fun(int x)
```

```
{
```

```
std::cout << "Value of x is : " <<x<< std::endl;
```

```
}
```

```
void fun(int &b)
```

```
{
```

```
std::cout << "Value of b is : " <<b<< std::endl;
```

```
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

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.

For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

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.h>
```

```
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.

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

```
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.

C++ Function Overriding Example

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

```
#include <iostream.h>  
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.

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

```
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.

C++ virtual function Example

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

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

```
class A {
```

```
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

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

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

Let's see a simple example:

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

```
class Base
```

```
{
```

```
public:
```

```
virtual void show() = 0;
```

```
};
```

```
class Derived : public Base
```

```
{  
public:  
void show()  
{  
std::cout << "Derived class is derived from the base class." << std::endl;  
}  
};  
int main()  
{  
Base *bptr;  
//Base b;  
Derived d;  
bptr = &d;  
bptr->show();  
return 0;  
}
```

Output:

Derived class is derived from the base class.

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

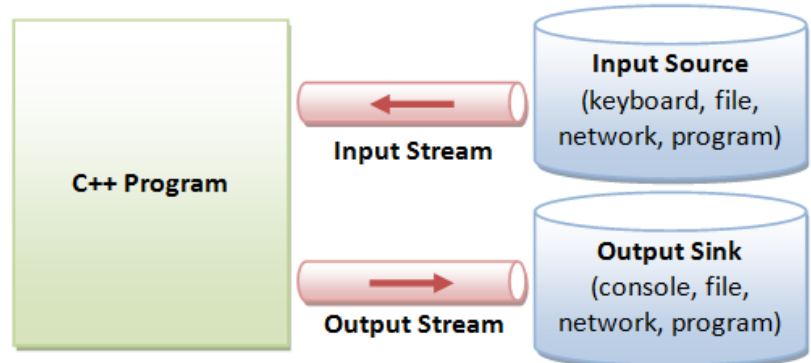
Chapter-6

I/O Operations and File Processing in C++

C++ comes with libraries which provides us with many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as **streams**.

C/C++ IO are based on *streams*, which are sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe). In input operations, data bytes flow from an *input source* (such as keyboard, file, network or another program) into the program. In output operations, data bytes flow from the program to an *output sink* (such as console, file, network or another program). Streams acts as an intermediaries between the programs and the actual IO devices, in such the way that frees the programmers from handling the actual devices, so as to archive device independent IO operations.

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.



Formatted & Unformatted I/O Operations

C++ provides both the *formatted* and *unformatted* IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators.

Unformatted data

- The printed data with default setting by the I/O function of the language is known as **unformatted data**.
- It is the basic form of input/output and transfers the internal binary representation of the data directly between memory and the file.
- **For example**, in cin statement it asks for a number while executing. If the user enters a decimal number, the entered number is displayed using cout statement. There is no need to apply any external setting, by default the I/O function represents the number in decimal format.

Formatted data

- If the user needs to display a number in hexadecimal format, the data is represented with the manipulators are known as **formatted data**.
- It converts the internal binary representation of the data to ASCII characters which are written to the output file.
- It reads characters from the input file and coverts them to internal form.
- **For example**, cout<<hex<<13; converts decimal 13 to hexadecimal d. Formatting is a representation of data with different settings (like number format, field width, decimal points etc.) as per the requirement of the user.

Managing output with Manipulators

Formatted output is very important in development field for easily read and understand. Manipulators are helper functions that make it possible to control input/output streams using operator << or operator >>.

C++ offers the several input/output manipulators for formatting, commonly used manipulators are given below...

Manipulator

endl

Declaration in

iostream.h

setw	iomanip.h
setprecision	iomanip.h
setf	iomanip.h

endl

endl manipulator is used to Terminate a line and flushes the buffer.

Difference b/w '\n' and endl

When writing output in C++, you can use either `std::endl` or `'\n'` to produce a newline, but each has a different effect.

- `std::endl` sends a newline character `'\n'` and flushes the output buffer.
- `'\n'` sends the newline character, but does not flush the output buffer.

The distinction is very important if you're writing debugging messages that you really need to see immediately, you should always use `std::endl` rather than `'\n'` to force the flush to take place immediately.

The following is an example of how to use both versions, although you cannot see the flushing occurring in this example.

```
#include <iostream.h>
int main()
{
    cout<<"USING '\n' ...'\n";
    cout<<"Line 1 \nLine 2 \nLine 3 \n";
    cout<<"USING end ..." << endl;
    cout<<"Line 1" << endl << "Line 2" << endl << "Line 3" << endl;
    return 0;
}
```

Output

```
USING '\n' ...
Line 1
Line 2
Line 3
USING end ...
Line 1
Line 2
Line 3
```

setw() and setfill() manipulators

setw manipulator sets the width of the field assigned for the output.

The field width determines the minimum number of characters to be written in some output representations. If the standard width of the representation is shorter than the field width, the representation is padded with fill characters (using `setfill`).

setfill character is used in output insertion operations to fill spaces when results have to be padded to the field width.

Syntax

```
setw([number_of_characters]);
setfill([character]);
```

Consider the example

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
    cout<<"USING setw() .....'\n";
    cout<< setw(10) << 11 << "\n";
    cout<< setw(10) << 2222 << "\n";
}
```

```

cout<< setw(10) <<33333<<"\n";
cout<< setw(10) <<4<<"\n";

cout<<"USING setw() & setfill() [type- I]...\n";
cout<< setfill('0');
cout<< setw(10) <<11<<"\n";
cout<< setw(10) <<2222<<"\n";
cout<< setw(10) <<33333<<"\n";
cout<< setw(10) <<4<<"\n";

cout<<"USING setw() & setfill() [type-II]...\n";
cout<< setfill('-')<< setw(10) <<11<<"\n";
cout<< setfill('*')<< setw(10) <<2222<<"\n";
cout<< setfill('@')<< setw(10) <<33333<<"\n";
cout<< setfill('#')<< setw(10) <<4<<"\n";
return 0;
}

```

Output

```

USING setw() .....
  11
 2222
33333
  4
USING setw() & setfill() [type- I]...
0000000011
0000002222
0000033333
0000000004
USING setw() & setfill() [type-II]...
-----11
*****2222
@@@@@33333
#####4

```

setf() and setprecision() manipulator

setprecision manipulator sets the total number of digits to be displayed, when floating point numbers are printed.

Syntax

```

setprecision([number_of_digits]);
cout<<setprecision(5)<<1234.537;
// output will be : 1234.5

```

Classes for file stream operations

File

A file is a stream of bytes stored on some secondary storage devices.

- **Text file:** A text file stores information in readable and printable form. Each line of text is terminated with an **EOL** (End of Line) character.
- **Binary file:** A binary file contains information in the non-readable form i.e. in the same format in which it is held in memory.

C++ file handling provides a mechanism to store output of a program in a file and read from a file on the disk. So far, we have been using **<iostream>** header file which provide functions **cin** and **cout** to

take input from console and write output to a console respectively. Now, we introduce one more header file **<fstream>** which provides data types or classes (**ifstream** , **ofstream** , **fstream**) to read from a file and write to a file.

Datatype	Description
ofstream	Used for creating a file and write data on files.
ifstream	Used for reading the data from files.
fstream	Used for both read and write data from files.

fstream.h:

This header file includes the definitions for the stream classes ifstream, ofstream and fstream. In C++ **file input output** facilities implemented through fstream.h header file. It contain predefines set of operation for handling file related input and output, fstream class ties a file to the program for input and output operation.

Following are the functions used in File Handling.

Function	Description
open()	It is used to create a file.
close()	It is used to close an existing file.
get()	It is used to read a single unit character from a file.
put()	It is used to write a single unit character in file.
read()	It is used to read data from file.
write()	It is used to write data into file.

Following are the operations of File Handling.

1. Naming a file.
2. Opening a file.
3. Reading data from file.
4. Writing data into file.
5. Closing a file

Opening a File

A file can be opened using:

- **By the constructor method.** This will use default streams for file input or output. This method is preferred when file is opened in input or output mode only.
Example : **ofstream file("student.dat");** or **ifstream file("student.dat");**
- **By the open() member function** of the stream. It will prefer when file is opened in various modes i.e ios::in, ios::out, ios::app, ios::ate etc.
e.g **fstream file;**
file.open("book.dat", ios::in | ios::out | ios::binary);
The open() function is used to open multiple files which uses the same stream object.
The fstream or ofstream object is used to open a file for writing and ifstream object is used to open a file for reading.

Syntax:

void open(const char *filename, ios::openmode mode);
where,

First argument ***filename** specifies the name of file and location.
Second argument **open()** member function defines the mode in which the file should be opened.

Following are the file opening modes

File mode	Description
ios::out	It open file in output mode (i.e write mode) and place the file pointer in beginning, if file already exist it will overwrite the file.
ios::in	It open file in input mode(read mode) and permit reading from the file.
ios::app	It open the file in write mode, and place file pointer at the end of file i.e to add new contents and retains previous contents. If file does not exist it will create a new file.
ios::binary	The file is opened in binary mode.
ios::ate	It open the file in write or read mode, and place file pointer at the end of file i.e input/output operations can performed anywhere in the file.
ios::trunc	If the file is already exists, using this mode the file contents will be truncated before opening the file (empties the file)..
ios::nocreate	If file does not exist this file mode ensures that no file is created and open() fails.
ios::noreplace	If file does not exist, a new file gets created but if the file already exists, the open() fails.

All these above modes can be combined using the bitwise operator (**|**). for example, ios::out | ios::app | ios::binary

eof(): This function determines the end-of-file by returning true(non-zero) for end of file otherwise returning false(zero).

Closing a File

- The close() function is used for closing a file.
- When a program terminates, it automatically closes or flushes all the streams, releases all the allocated memory and closes all the opened files. But a good practice for programmer to close all the opened files before the program termination.

Syntax:

Stream_object.close(); e.g file.close();

- A file must be closed after completion of all operation related to a file.

Writing to a File

- The insertion operator (<<) is used to write information in a file.
- The ofstream or fstream object is used instead of cout object.

Reading from a File

- The extraction operator (>>) is used to read information from a file into your program.
- The ifstream or fstream object is used instead of cin object.

Input and Output Operation

Following functions are used for I/O Operation:

Function	Description
put()	This function writes a single character to the associated stream.

get()	This function reads a single character to the associated stream.
write()	This function is used to write the binary data.
read()	This function is used to read the binary data.

Binary file functions:

- **read()**- read a block of binary data or reads a fixed number of bytes from the specified stream and store in a buffer.

Syntax : Stream_object.read((char *)& Object, sizeof(Object));

e.g file.read((char *)&s, sizeof(s));

- **write()** – write a block of binary data or writes fixed number of bytes from a specific memory location to the specified stream.

Syntax : Stream_object.write((char *)& Object, sizeof(Object));

e.g file.write((char *)&s, sizeof(s));

Note:

Both functions take two arguments.

- The first is the address of variable, and the second is the length of that variable in bytes. The address of variable must be type cast to type char*(pointer to character type)
- The data written to a file using write() can only be read accurately using read().

File Pointers

- File pointer is associated with two pointers,
 - 1. Input pointer** reads the content of a given file location.
 - 2. Output pointer** writes the content to a given file location.
- All Input/Output stream objects have at least one internal stream pointer.
- The ifstream has a get pointer which points to the element to read in the next input operation.
- The ofstream has a put pointer which points to the location where the next element has to be written.
- The fstream inherits both the get and put pointers from istream.

Following are the member function for manipulation of file pointer:

Function	Description
seekg()	It moves the get pointer (input) to a specified location.
seekp()	It moves the put pointer (output) to a specified location.
tellg()	It gives the current position of the get pointer.
tellp()	It gives the current position of the put pointer.

Steps To Create A File

- Declare an object of the desired file stream class(ifstream, ofstream, or fstream).
- Open the required file to be processed using constructor or open function.
- Process the file.
- Close the file stream using the object of file stream.

Example: Program demonstrating Read and Write mode of a file

Following program reads the information from the file and displays it onto the screen.

```
#include <fstream.h>
```

```
#include <iostream.h>

int main ()
{
    char name[50];

    ofstream ofile;           // open a file in write mode.
    ofile.open("abc.dat");

    cout << "Writing to the file" << endl;
    cout << "Enter your name: " << endl;
    cin.getline(name, 50);
    cout << endl;
    ofile << name << endl;      // write inputted data into the file.
    ofile.close();            // close the opened file.

    ifstream ifile;           // open a file in read mode.
    ifile.open("abc.dat");

    cout << "Reading from the file" << endl;
    ifile >> name;

    cout << name << endl;      // write the data at the screen.

    ifile.close();            // close the opened file.

    return 0;
}
```

Output:

Writing to the file
Enter your name:
EASY TECH CLASSES

Reading from the file
EASY TECH CLASSES

Command line arguments in C/C++

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.

- If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- argv[0] is the name of the program, After that till argv[argc-1] every element is command-line arguments.

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

Note : You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ".

// C program to illustrate

// command line arguments

#include<stdio.h>

```
int main(int argc, char* argv[])
```

```
{
    int counter;
    printf("Program Name Is: %s", argv[0]);
    if(argc==1)
        printf("\nNo Extra Command Line Argument Passed Other Than Program Name");
    if(argc>=2)
    {
        printf("\nNumber Of Arguments Passed: %d", argc);
        printf("\n---Following Are The Command Line Arguments Passed---");
        for(counter=0; counter<argc; counter++)
            printf("\nargv[%d]: %s", counter, argv[counter]);
    }
    return 0;
}
```

Output in different scenarios:

1. **Without argument:** When the above code is compiled and executed without passing any argument, it produces following output.

```
$ ./a.out
Program Name Is: ./a.out
No Extra Command Line Argument Passed Other Than Program Name
```

2. **Three arguments :** When the above code is compiled and executed with a three arguments, it produces the following output.

```
$ ./a.out First Second Third
Program Name Is: ./a.out
Number Of Arguments Passed: 4
---Following Are The Command Line Arguments Passed---
argv[0]: ./a.out
argv[1]: First
argv[2]: Second
argv[3]: Third
```

3. **Single Argument :** When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

```
$ ./a.out "First Second Third"
Program Name Is: ./a.out
Number Of Arguments Passed: 2
```

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First Second Third

4. **Single argument in quotes separated by space** : When the above code is compiled and executed with a single argument separated by space but inside single quotes, it produces the following output.

\$./a.out 'First Second Third'

Program Name Is: ./a.out

Number Of Arguments Passed: 2

----Following Are The Command Line Arguments Passed----

argv[0]: ./a.out

argv[1]: First Second Third

Practice programs:-

Chapter-5 **Exception Handling**

What is Exception?

- Exception is a problem that occurs during the execution of a program. It is a response to exceptional circumstances like runtime errors while a program is running. In other word , exception is an event or object which is thrown at runtime. All exceptions are derived from `std::exception` class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

What is Exception Handling?

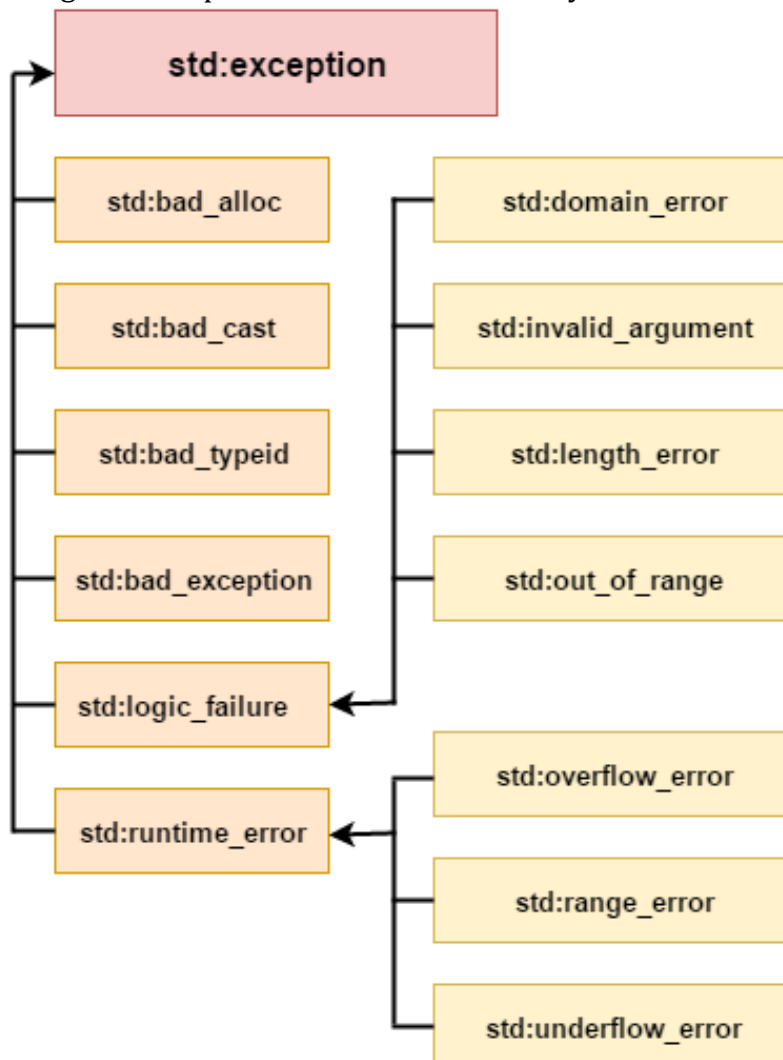
- Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.
- An exception transfers the control to special functions called Handlers.
- Exception handling makes it easy to separate the error handling code.
- It provides a way to transfer control from one part of a program to another.
- Exception handling makes code readable and maintainable. It separates the code to the normal flow.
- Function can handle or specify any exceptions they choose.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in `<exception>` class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from `std::exception` class. Let's see the list of C++ common exception classes.

Exception	Description
<code>std::exception</code>	It is an exception and parent class of all standard C++ exceptions.
<code>std::logic_failure</code>	It is an exception that can be detected by reading a code.
<code>std::runtime_error</code>	It is an exception that cannot be detected by reading a code.
<code>std::bad_exception</code>	It is used to handle the unexpected exceptions in a c++ program.
<code>std::bad_cast</code>	This exception is generally be thrown by dynamic_cast .
<code>std::bad_typeid</code>	This exception is generally be thrown by typeid .
<code>std::bad_alloc</code>	This exception is generally be thrown by new .

Following are the three specialized keywords where exception handling is built.

1. throw
2. try
3. catch

1. throw

Using throw statement, an exception can be thrown anywhere within a code block. It is used to list the exceptions that a function throws, but doesn't handle itself.

Following example shows throwing an exception when dividing by zero condition occurs.

```
float div(int x, int y)
{
    if(y==0)
    {
        throw "Divide by zero condition!!!";
    }
    return (x/y);
}
```

2. try

Try represents a block of code that can throw an exception. It identifies a block of code which particular exceptions will be activated. Try block followed by one or more catch blocks.

Syntax:

```
try
{
    //Code
}
catch(ExceptionName e1)
{
    //Catch block
}
catch(ExceptionName e2)
{
    //Catch block
}
catch (ExceptionName e3)
{
    //Catch block
}
```

3. Catch

Catch represents a block of code executed when a particular exception is thrown. It follows the try block which catches any exception.

Syntax:

```
try
{
    //code
}
catch(ExceptionName e)
{
    //code to handle exception
}
```

In the above syntax, code will catch an exception of ExceptionName type.

Example: Program demonstrating flow of execution of Exception

```
#include <iostream>
using namespace std;
int main()
{
    int a = -2;
    cout << "Before Try Block"<<endl;
    try
    {
        cout << "Inside Try Block"<<endl;
        if (a < 0)
        {
            throw a;
            cout << "After Throw Exception (Never executed)"<<endl;
        }
    }
    catch (int a )
    {
        cout << "Exception Caught \n";
    }
    cout << "Executed After Catch Exception"<<endl;
    return 0;
}
```

Output:

```
Before Try Block
Inside Try Block
Exception Caught
Executed After Catch Exception
```

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

You can define your own exceptions. Following example shows how to implement your own exception.

Example: Program demonstrating developer can create their own exceptions

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

struct MyException : public exception
{
```

```
const char *test () const throw ()
{
    return "New Exception";
};
int main()
{
    try
    {
        throw MyException();
    }
    catch(MyException &e)
    {
        cout<< "MyException caught"<<endl;
        cout<< e.test()<<endl;
    }
}
```

Output:

MyException caught
New Exception

In the above program, test() is a public method provided by exception class. It returns the cause of an exception.

Illustrate Rethrowing exceptions with an example.

- Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception. This causes current exception to be passed on to an outer try/catch sequence. An exception can only be rethrown from within a catch block. When an exception is rethrown, it is propagated outward to the next catch block.

- Consider the following code :

```
#include <iostream>
using namespace std;
void MyHandler()
{
    try
    {
        throw "hello";
    }
    catch (const char*)
    {
        cout <<"Caught exception inside MyHandler\n";
        throw; //rethrow char* out of function
    }
}
int main()
{
    cout<< "Main start";
    try
```

```
{
    MyHandler();
}
catch(const char*)
{
    cout << "Caught exception inside Main\n";
}
    cout << "Main end";
    return 0;
}
```

Output :

Main start

Caught exception inside MyHandler

Caught exception inside Main

Main end

- Thus, exception rethrown by the catch block inside MyHandler() is caught inside main();