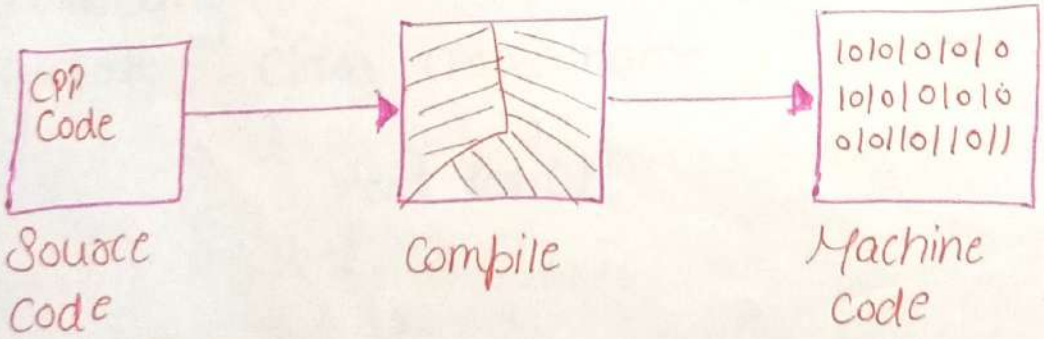
- C++ is a object oriented programming language and it is also known as structural programming language.
- It is developed by "Bjarne Stroustrup" in 1979 at "Bell Labs".
- C++ is a "general purpose", "Case Sensitive", "free-form" programming language that supports "object-oriented, procedural programming.
- C++ is a "middle-level" language as it encapsulates both high and low level language features.
- In earlier the name of C++ was "C with Classes"
- It is an imperative and a compiled language



Structures of C++

There are 4 sections.

1) include files
2) class declaration
3) Member funⁿ defination
4) Main funcⁿ program.

Copyrighted by Codeleap.io
All rights reserved.

```
# include <iostream>   ] → Header files
int main()  ] ← Main function return Type
{  → opening Brace
Cout << "Hello World";  ← Body of function
return 0;
}  → Closing Brace
```

Copyrighted by Codeleap.io
All rights reserved.

⇒ // → Comments (Single line)

⇒ /* */ → Multiline Comments

⇒ Output operator Cout <<

⇒ Input operator Cin >>

⇒ Cascading of operators :→ The multiple use of << and >> operator in one statement

C++ features

1) Simple :→ It is a simple language in the sense that programs can be broken down into logical unit and parts has a rich Library support.

2) Portable :→ C++ is a portable language and programs Made in it can be run on different Machines.

3) Mid-level Programming Language :→ C++ includes both low-level and High-level language so it is known as Mid-level programming Language. It is used to develop System application such as kernel, driver etc.

4) Structured programming language:→ C++ is a Structured program language. In this we divide the program into several parts using functions.

5) Rich library:→ C++ provide a lot of inbuilt functions that make the fast development.

- <iostream>
- <cmath>
- <stdlib>
- <fstream>

6) Pointer:→ C++ provides the features of pointers. We can use pointer for memory, structures, function, array etc.

7) Memory Management:→ C++ provides very efficient Management techniques. The various Memory Management operators help save the memory and improve the program efficiency.

8) Recursion:- In C++, we can call the function within the function. It provides code reusability for every function.

9. Compile-based:- C++ is a compiler-based program language, which means no C++ program can be executed without compilation.

- first compile the program then execute our program

Copyrighted by Codeleap.io
All rights reserved.

C++ Variable

- Variable refers to the memory location name where we stored values.
- It value can be changed and it can be reused many times.
- Syntax :→ type variable_list;

Ex :→
```
int x;
float y;
char z;
```

→ Here, x, y, z are variables
int, float, char are data types

→ Rules for defining Variables

- A variable can have alphabets, digits and underscore
- A variable name can start with alphabet and underscore only. It can't start with digit.
- No white space is allowed within variable name.

Valid Variable names

```
int a;
int _ab;
int a30;
```

Invalid variable names

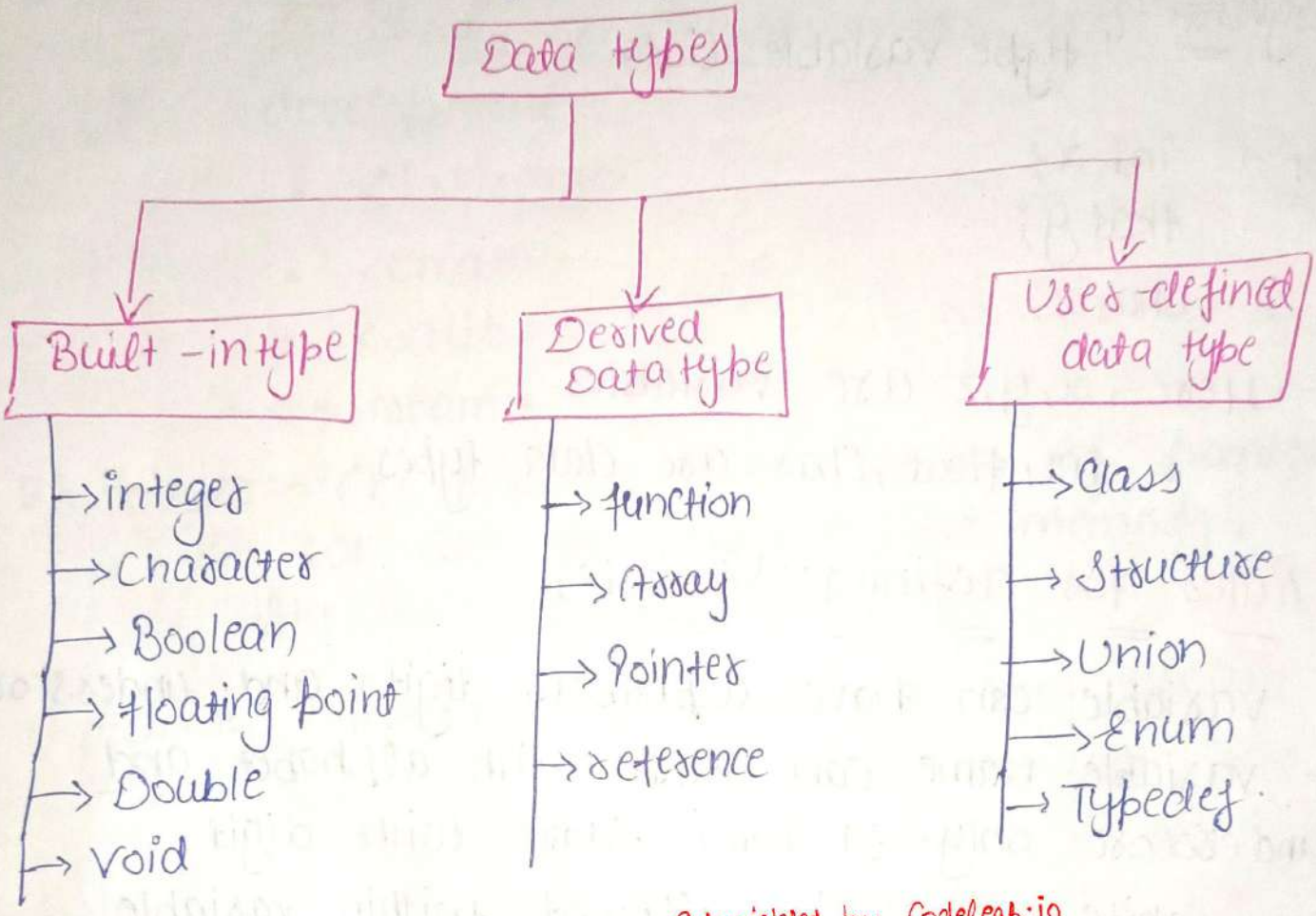
```
int 4;
int x y;
int double;
```

Copyrighted by Codeleap.io
All rights reserved.

C++ Data Types

Data types define the types of data a variable can hold.

- There are three type of data type in C++



Copyrighted by Codeleap.io
All rights reserved.

C++ keywords

A keyword is a reserved word which have fixed meaning and its meaning cannot be changed.

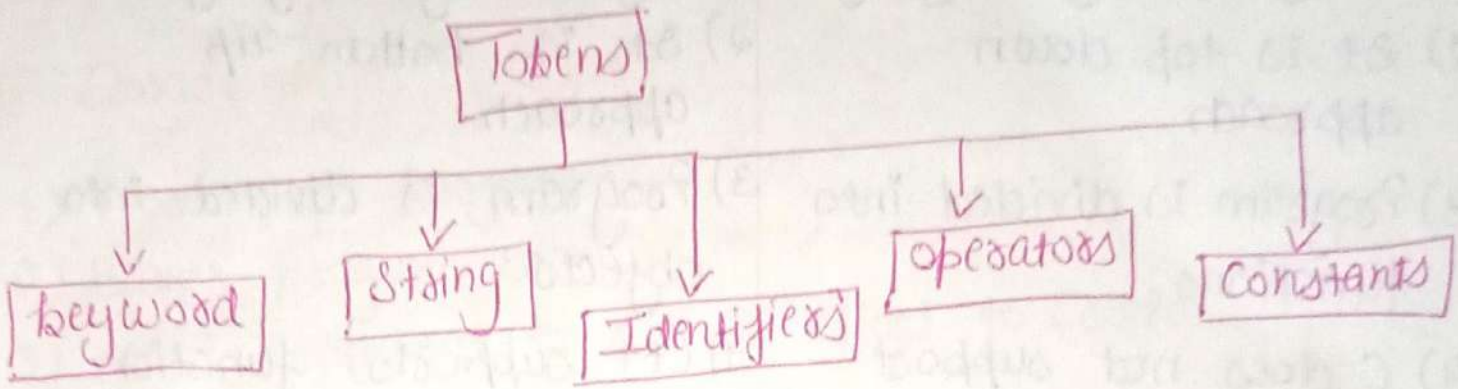
- The meaning and working of these keywords are already known to the compiler.
- There are 32 keywords in C++ language

auto	break	case	char	const
double	else	enum	extern	float
int	long	register	return	short
Struct	switch	typedef	union	unsigned etc.

C++ Tokens

A Token is the smallest element of a program that is meaningful to the compiler.

- Tokens act as building blocks of a program.



C++ Identifiers

Identifiers refers to the name of variables, function, arrays, class etc created by programmer.

- It can use both lowercase and uppercase letters

Rules

- only alphabetic characters, digits and underscores are allowed.
- The identifier cannot start with a digit i.e the first letter should be alphabetical. After the first letter, we can use letters, digits or underscores.
- A declared keyword cannot be used as a variable name.

valid identifiers

Result
Test2
_SUM

Invalid identifiers

SUM-1 // contain Character '-'.
2data // first letter is digit
break → // use of keyword.

[Identifiers cannot be used as the keywords.

Copyrighted by Codeleap.io
All rights reserved.

Difference b/w C and C++

C	C++
1) C is an Structural programming language	1) C++ is an object-oriented programming language.
2) It is top down approach	2) It is bottom up approach.
3) Program is divided into functions	3) Program is divided into objects.
4) C does not support function overloading.	4) C++ supports function overloading.
5) scanf() and printf() are mainly used for input/output.	5) Cin and cout to perform input and output operation.
6) C does not support the inheritance.	6) C++ supports the inheritance.
7) C does not provide the feature of name-space.	7) C++ supports the features of namespace.
8) It is less secure.	8) It is more secure.
9) Ex:-> C, Fortan	9) Eg:- C++, Java
10) It takes very less Memory	10) It takes More memory than Structural language.

Benefits of OOPS

1) The Reusability
2) Simplicity
3) Modifiability
4) Maintainability
5) Extensibility [add new]
6) Modularity [Separate]

Copyrighted by Codeleap.io
All rights reserved.

Advantages of oops

1) Re-usability
2) Data redundancy
3) Security
4) Problem solving
5) Code Maintenance
6) Design Benefits

Disadvantages of oops

1) More slow projects
2) Bigger program size
3) It is required to lot of effort to create.
4) Complexity

Object oriented programming

The word object - oriented is the combination of two words i.e object and oriented. The object meaning is an article or entity that exists in the real world

- The meaning of oriented is interested in a particular kind of thing or entity.
- The Major concepts / principle such as:

Abstraction
inheritance
polymorphism
Encapsulation

=> Object :-> Any entity that has state and behaviour is known as object.

Here State Means "data"
behaviour Means "functionality".

Copyrighted by Codeleap.io
All rights reserved.

- Object is a runtime entity, it is created at runtime.
- Object is an instance of a class.
- When a Class is defined, no memory is allocated but when it object is created, mememory is allocated.

Syntax: Classname objectname;

- It is a user-defined data type, which holds its own data members and members functions which can be accessed and used by creating an instance of that class.
- A Class is like a blueprint for an object.

Example Car ← Object

Data Member → speed limit etc.

Member function → apply brakes, increase speed

Access Data Members and Members functions

The data members and member function of class can be accessed using dot (.) operator with object.

- Syntax: objectName.memberdata;

objectName.memberfunction();

- The attributes of an object are represented by Data Members.
- The behaviour is represented by its Member function.
- Both are binded together as a single unit

Ex:-

Copyrighted by Codeleap.io
All rights reserved.

```
Class Student
{
  public:
     int rollno;      } Member data.
     String name;
     void show()      } member function
     (display)
     {
       Cout <<" your name = "<<name;
     }
};
int main()
{
   Student ob;
   ob.Rollno = 30;
   ob.name = "Tannu";
   ob.show();
   return 0;
}
```

Class

[Active Entity]

object

output
[yourname is Tannu]

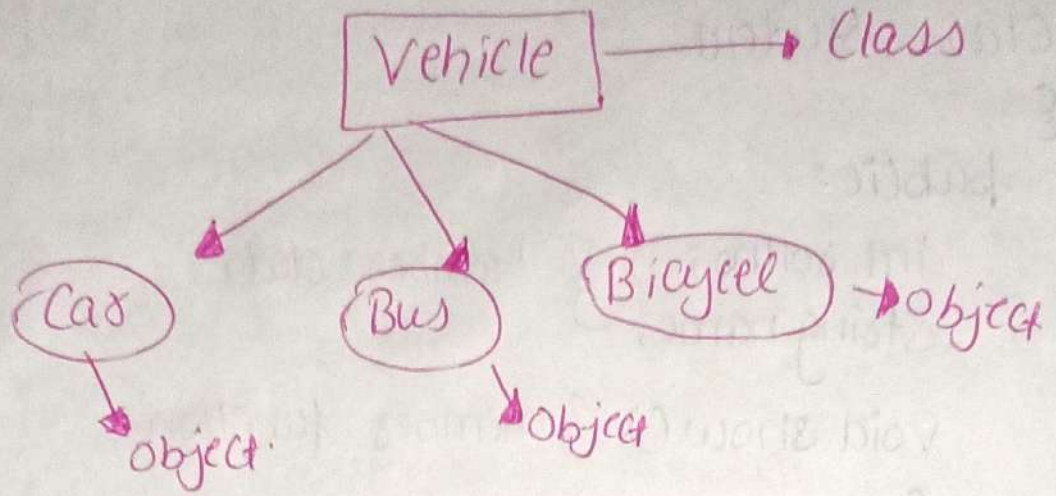
⇒ Class :- 1. Class is a Collection of object.
- Class is a "Passive entity".

- A Class is a User defined data type which has data Members and member function.

Syntax

```
Class Classname → User-defined name.
{
  access specifier: // Can be private, public
    Data Members; // variable to be used.
    Member function()
    {  body of function
    };
```

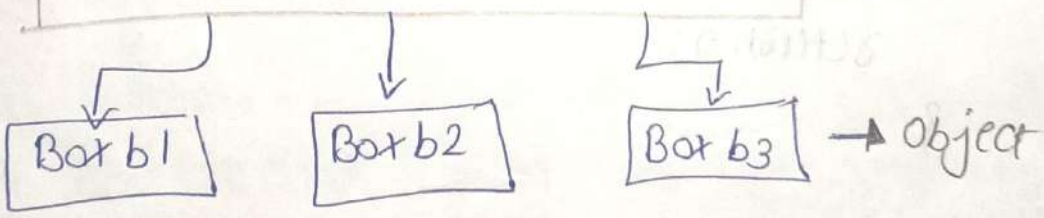
keyword



Ex:→ Class and object

```
Class Box
{
    private:
        data members
    public:
        Member function
};
```

← Class



⇒ Abstraction:→ Hiding internal details and showing functionality.

Eg:→ Washing Machine.

→ Abstraction using Class.

→ Abstraction using Header files

Copyrighted by Codeleap.io
All rights reserved.

In Object Oriented program, each object of Class may have internal and external part.

→ The internal part are represent by Private Members that can not be accessed from outside the side.

→ The external part are represent by Public Members that can be accessed from outside the Class.

- Both the class Members may be private or public

⇒ Encapsulation → Binding.

- Code and data together into a single unit

Ex:→ Capsule. It is wrapped with different Medicines

- It is the one of the most important features of oops.
- We can not access it directly, we can access the data through its member function.

Advantage

1) Data Hiding
2) Information Hiding

→ Syntax

```
ClassA
{
    private data
        +
    public methods
}
```

Copyrighted by Codeleap.io
All rights reserved.

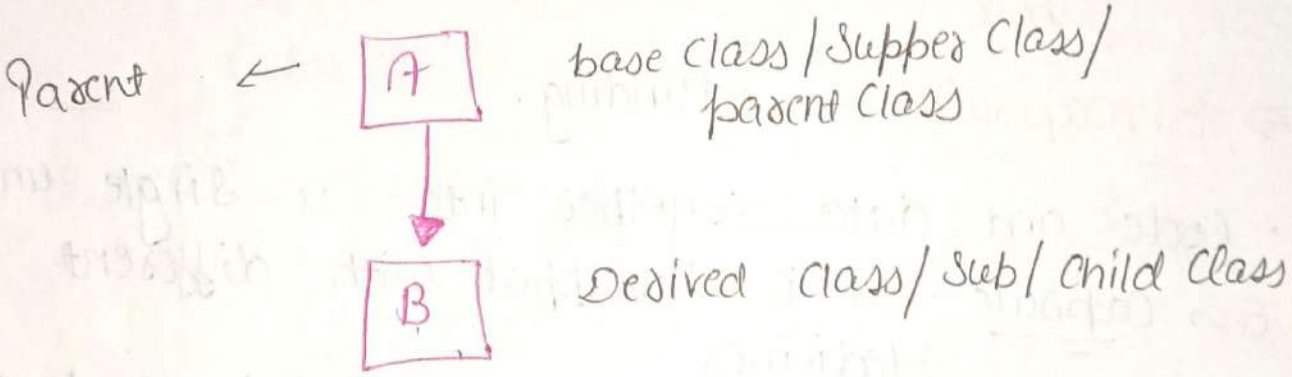
⇒ Inheritance

- The capability of class to drive properties and characteristics from another class is called inheritance.

or

When one class the property (data member and member function) of another class is called inheritance.

like :→ child inherits the properties of his parents



- The class which inherits the Members of another class is called Derived Class
- And the class whose members are inherited is called base Class

Ex :→

```
Class A
{
  public:
  void display()
  {
    ___ cout << "display Class";
  }
};
Class B : public A
{
  public:
```

Super Class

Copyrighted by Codeleap.io
All rights reserved.

```
void Show()
{
    show sub class
}
};
int main()
{
    B ob;
    ob.display();
    ob.Show();
    return 0;
}
```

Copyrighted by Codeleap.io
All rights reserved.

⇒ Polymorphism

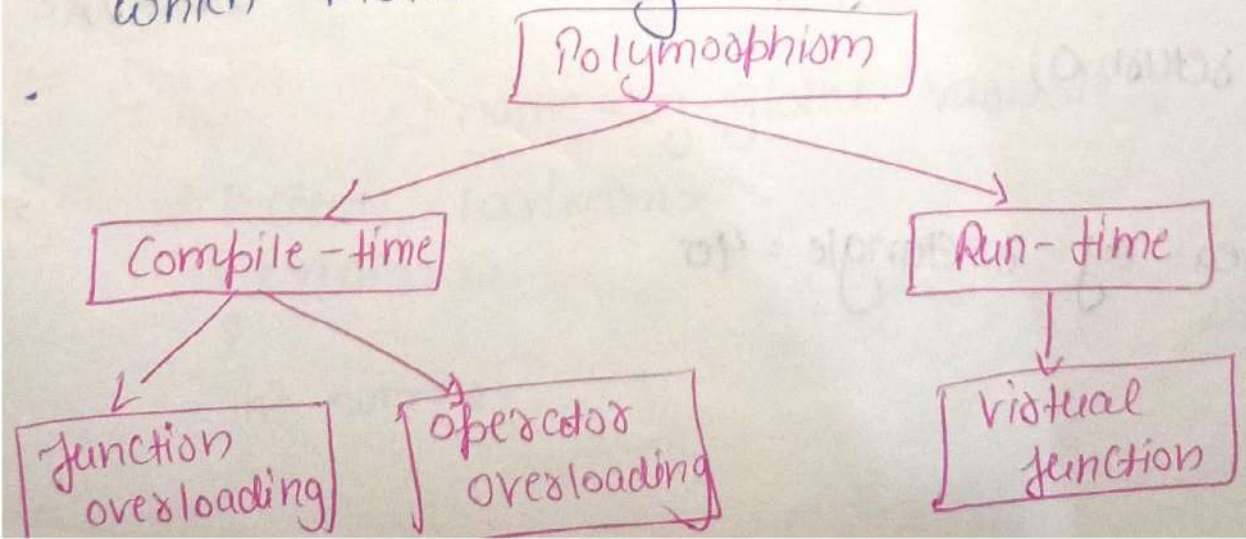
- When one task is performed by different ways are known as polymorphism.

for ex:→ A person at the same time can have different character.

like a father, a husband and Employee.

So the same person posses Different behaviour in different situations.

- Polymorphism is the combination of "Poly + morpho" which means many forms.



⇒ Structures

- In C++ classes and structs are blueprints that are used to create the instance of a class.
- C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

Syntax:

```
struct structure_name
{
    // method declaration.
}
```

Ex:→

```
# include <iostream>
using namespace std;
struct rectangle:
{
    int width, height;
};
int main()
{
    struct rectangle rec;
    rec.width=8;
    rec.height=5;
    cout << "area of rectangle = "<<(rec.height *rec.width);
    return 0;
}
```

→ area of rectangle = 40

Copyrighted by Codeleap.io
All rights reserved.

⇒ Application of oops

- Computer graphics application
- Object - oriented database.
- Client - Server System
- OAS
- Artifical intelligence system.
- User - interface design

⇒ Scope resolution operator

- Scope resolution operator in C++ programming Language is used to define a function outside a class or when we want to use a global variable but also has a local variable with the same name.

- [::]

- We use the scope resolution operator to access the hidden variable or function of a program.

Uses

1) Uses to hidden variable.
2) Define Member function outside the Class
3) Used to override function

Syntax:

:: name - of - global - variable

Ex:-

```
#include <iostream>
int main()
{
int num = 0;
```

Copyrighted by Codeleap.io
All rights reserved.

```
Std::cout<<"Enter the value of num:";
Std::cin>>num;
Std::cout<<"The value of num is:"<<num;
}
```

Output

```
Enter the value of num:50
The value of num is:50
```

Static Data Members

- Static Data Members are class Members that are declared using static keywords. Whenever we declare a data member as a static either inside or outside of a class called static data member.
- A Static Member has certain special characteristics
  1) There is only one copy of static data member even if there are many class object.
  2) It is shared Memory for all objects of the class.
  3) It is always initinlized with zero because it's default value is zero

Syntax: Static data-type data-member;

Static → keyword.

data-type → variable type in C++, int, float.

data-Member → name of static data

Copyrighted by Codeleap.io
All rights reserved.

Static Member function

- If we create a member function of a Class as a static called Static Member function.
- It is access only static data Members (Non-Static) Not follow
- you can also use a Static Member function to determine how many object of the Class have been created.

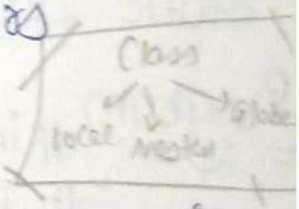
Syntax: Class_name :: function_name (parameters);

Name of Class

Static Member function

Argument pass

- A Static Member function shares the single copy of the member function to any number of the Class object
- It also not associated with object
- It cannot access non-Static data Members



Nested Class

A Class which contain inside another Class is called Nested Class

- The Class which contains thn nested Class is called a Enclosing Class

Syntax:
```
Class Class_name1
{
    public:
        Class Class_name2   // member variable
        {
            // Data Member
            public:
                // member function
        };
};
```

Copyrighted by Codeleap.io
All rights reserved.

Local Class in C++

- A class which is declared inside a function or block is called local class.

Syntax: return-type function name

```
{
    Class class-name
    {
        body of function
    };
}
main()
{
    function-name;
}
```

- A local class name can only be used locally. i.e inside the function and not outside it.
- A local class can have static functions but, not static data members.

Program

```
#include<iostream.h>
#include <conio.h>
void fun()
{
    Class A
    {
        private:
        int a,b;
        public:
        void show()
        {
            cout<<"enter two values:";
            cin >>a>>b;
```

Copyrighted by Codeleap.io
All rights reserved.

```
        cout<<a<<" "<b;
        }
    };
    A obj;
    obj.show();
}
void main()
{
    clrscr();
    fun();
    getch();
}
```

Output: enter two value:
78
72
78 72

Global Class

A Class which declared outside of all the function or block.

- It can be used and changed at any part of the program after its declaration.

Access Specifiers

- Access specifiers define how the members of a calls can be assessed.

→ Private	Public	Protected
• They are both friend function and Member function	• Member function and outside function	• Use in inheritance.

Copyrighted by Codeleap.io
All rights reserved.

Program for Static Data & function

```
#include <iostream>
Class A
{
    int a;                  } [Private Data Member
    Static int b;           } of class]
    public:
    A(int x, int y)         } [we pass two paramter x, y]
                              [x store to a , y store to b]
    {
      a=x; b=y;
    }
    void show()             } Member function
    {
      cout <<a <<" "<<b;
    }
    Static void disp()      } Static Member
                              function.
    {
      cout <<b;  // a is not accessible
                              [non-Static member function]
    }
};
int A::b=0;                 [b is Static Members]
void main()                 [a is non-Static
{
    A obj (10,20), obj2 (100,200);
    obj.show();             } call Member function   10 20
    obj2.show();                                     100 200
    A::disp();              [without object asses]
                              only access b.
    obj.show();             10 200
}
```

Copyrighted by Codelcap.io
All rights reserved.

Constructors

→ its name is the same as the class name.

- A Constructors is a special type of member func^n of a Class which initializes object of Class.
- In C++ Constructor is automatically called when an object of a Class is created.
- It is a special member function of a Class because it does not have any return type.

Syntax <class-name> (list-of-parameters);

- Constructors can be defined inside or outside the Class declaration.

→ The syntax for defining the constructor within the Class:

<class-name> (list-of-parameters) //Constructor defination

→ The syntax for defining the constructor within outside the Class:

<class-name>::<class-name> (list-of-parameters) { //constructor definition }

Eg:→

```
Class Student
{
  public:
    Student()
    {
      cout << "Constructor initiated";
    }
};
int main()
{ Student ob;
  return 0;
```

Copyrighted by Codeleap.io
All rights reserved.

- Why we use or purpose of constructor in C++.
  → Constructor is a special member function of class which is used to create and initalize the objects.

- How constructor are different from normal member function.

1) Constructor don't have return type.
2) Constructor has same name as the class itself.
3) A Constructor is automatically called when an object is created.
4) It must be placed in public section of class.
5) Default Constructors don't have input argument however, Copy and parameterized constructors have input arguments.
6) Constructors used to initalize the new object of data Members.

Characteristics of the Constructor

- Constructors can be overloaded
- Constructors can not be declared virtual.
- other characterstic same as normal member.

Can we have more than one constructor in a Class

We can have multiple constructor in a class under the same name but a different list of arguments. This concept is also known as constructor overloading.

Types of Constructor

There are three types of Constructor.

Copyrighted by Codeleap.io
All rights reserved.

1) Default (Class_name())
2) Parameterized (Class_name(parameters)
3) Copy (Class_name(const Class_name old_object)

1) Default Constructor :

It is the constructor which does not take any argument. It has no parameters.

Syntax:

```
Class Classname
{
  public:
     Classname()
     {
       member data;
       member function;
     }
};
```

- It is also known as zero-argument constructor
- It not have any return type, even void it is called automatically provide a default constructor.

Program

```
#include<iostream.h>
# include <string.h>
# include <conio.h>
  Class bca
  {
     int roll;
     char name [20];
     float marks;
     public:
          bca()
```

Copyrighted by Codeleap.io
All rights reserved.

```
{
    roll=0;
    Strcpy(name," ");
    marks =0;
}
void put_data()
{
    Cout<<"Name:"<<name<<endl;
    Cout<<"Marks:"<<marks<<endl;
    Cout<<"Roll:"<<roll<<endl;
}
};
void main()
{
    clrscr();
    bca ob;
    ob.put_data();
    getch();
}
```

Output:- Name:
Marks: 0
Roll: 0

2) Parameterized Constructors: → A Constructors which has parameters is called parameterized Constructors.

- Constructor is used to provide different value to distinct object

Uses: → 1) It is used to overload Constructors.
2) It is used to initialize the various data elements of different object with different values when they are created.

Copyrighted by Codeleap.io
All rights reserved.

Program:

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
Class bca
{
 int roll;
 char name[20];
 float marks;
 public:
    bca (int r, char n[], float m) // parametrised constructor
    strcpy{
    {
      roll=r;
      strcpy(name,n);
      marks=m;
    }
    bca (bca &ob)
    {
      roll=ob.roll;
      strcpy(name,ob.name);
      marks=ob.marks;
    }
    void put_data()
    {
      cout<<"Name:"<<name<<endl;
      cout<<"Marks:"<<marks<<endl;
      cout<<"Roll:"<<roll<<endl;
    }
};
void main()
{
  clrscr();
```

Copyrighted by Codeleap.io
All rights reserved.

```
bca ob(1,"tanu",500.3),ob2(ob1);
ob1.put_data();
ob2.put_data();
getch();
}
```

Output: Name: tanu
Marks: 500.29
Roll: 1
Name: tannu
Marks: 500.29
Roll: 1

3) Copy Constructor: ↳ A copy constructor is an overloaded constructor used to declare and initialize an object from another object

Syntax

Class_name (Const Class_name & old_obj);

Ex:-> Geeks for Geeks (Const Geeks for Geeks & old_obj);

Class name, Class name, object

-> Characteristics

1) The copy constructor is used to initialize the members of a newly created object by copying the members of an object.

2) The copy constructor can be defined explicitly by the programmer.

3) It takes a reference to an object of the same class as an argument

Copyrighted by Codeleap.io
All rights reserved.

Type of Copy Constructor

1) Default Constructor:- The compiler defines the default copy constructor.

- If the user defines no copy constructor, compiler supplies its constructor.

2) User Defined Constructor:- The programmer defines the user-defined constructor.

When is the copy constructor called?

- When an object of the class is returned by value.
- When an object of the class is passed by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object

Copy Elision

- The compiler prevents the making of extra copies which results in saving space and better the program complexity (both time and space)
- Hence making the code more optimized

Copyrighted by Codeleap.io
All rights reserved.

Program

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
Class bca
{
    int roll;
    char name[20];
    float marks;
    public:
        bca(int r, char n[], float m) //parametrised
                                        constructor
        {
            roll=r;
            strcpy(name,n);
            marks=m;
        }
        bca(bca &ob)  //copy constructor
        {
            roll=ob.roll;
            strcpy(name, ob.name);
            marks=ob.marks;
        }
        void put-data()
        {
            cout<<"Name:"<<name<<endl;
            cout<<"Marks:"<<marks<<endl;
            cout<<"Roll no:"<<roll<<endl;
        }
};
```

Copyrighted by Codeleap.io
All rights reserved.

```
void main()
{
    clrscr();
    bca obl(1,"Tanu",500.0), ob2(obl);
    obl.put-data();
    ob2.put-data();
    getch();
}
```

Constructor Overloading

- If a class contain multiple constructor where each type of constructors having different parameters list is known as constructor overloading
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

[] Means having more than one constructor with the same name].

Program

```
#include <iostream.h>
# include< string.h>
# include< conio.h>
class bca
{
  int roll;
  char name[20];
  float marks;
  public:
      bca()        // default constructor
      {
```

Copyrighted by Codeleap.io
All rights reserved.

```
        roll=0;
        strcpy(name," ");
        marks=0;
    }
    bca (int r, char n[], float m)  //parametrised
    {
        roll=r;
        strcpy(name,n);
        marks=m;
    }
    bca (bca &ob)   //copy Constructor
    {
        roll=ob.roll;
        strcpy(name,ob.name);
        marks=ob.marks;
    }
    void put-data()
    {
        cout<<"Name:"<<name<<endl;
        cout<<"Marks:"<<marks<<endl;
        cout<<"Roll:"<<roll<<endl;
    }
};
void main()
{
    clrscr();
    bca ob1;
    bca ob2(1,"tanu",500.6);
                        //calling default Constructor.
    bca ob3;
    ob1.put-data();     //calling parametrized
    ob2.put-data();
    ob3.put-data();     //Calling copy
    getch();
```

Copyrighted by Codecap.io
All rights reserved.

Default Value to Parameters (Arguments)

- you can use a default parameter value, by using the equal sign (=).
- If we call the function without an argument, it uses the default value.
- A Default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.
- During the calling of function, the values are copied from left to right

→ Advantages

- It helps in reducing the size of a program.
- It useful when we want to increase the capabilities of an existing function
- It provide a simple and effective programming approach.

→ Disadvantages

- It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.

→ All values that wille given default value will be on right

Ex:→ void function (int x, int y, int z = 0)

Explain:→ The above function is valid. Here z is the value that is predefined as a part of default argument.

Copyrighted by Codeleap.io
All rights reserved.

- void function (int x, int z=0, int y)
  Explain → The above function is invalid. Here z is the value defined in b/w and it is not accepted

Destructors

- A destructor works opposite to constructor. It destructs the object of class.
- Destructor is also a special member function like constructor. Destructor destroys the class object created by constructor.
- Destructor has the same name as their class name preceded by a tilde(~) symbol.
- The destructor does not have arguments.
- It cannot be declared static or const.
- It is not possible to define more than one destructor.
- Destructor should be declared in public section of the class.

Syntax:

```
Class Classname
{
  public:
    classname()    // Constructor
  { }
  ~Classname()    // Desctructor
  { }

};
```

Copyrighted by Codeleap.io
All rights reserved.

• When the destructor called?

A destructor function is called automatically when the object goes out of scope.

1) function ends
2) program ends.

Program

```
#include <iostream>
using namespace Std;
Class Distructor Test
{
  Public:
  Distructor Test()        //Constructor
  {
   Cout<<"Constructor Invoked"<<"\n";
  }
  ~Distructor Test()  // Distructor
   {
    Cout<<"Distructor involved";
   }
};
int main()
{
   Distruction Test ob;
   return 0;
}
```

Copyrighted by Codeleap.io
All rights reserved.

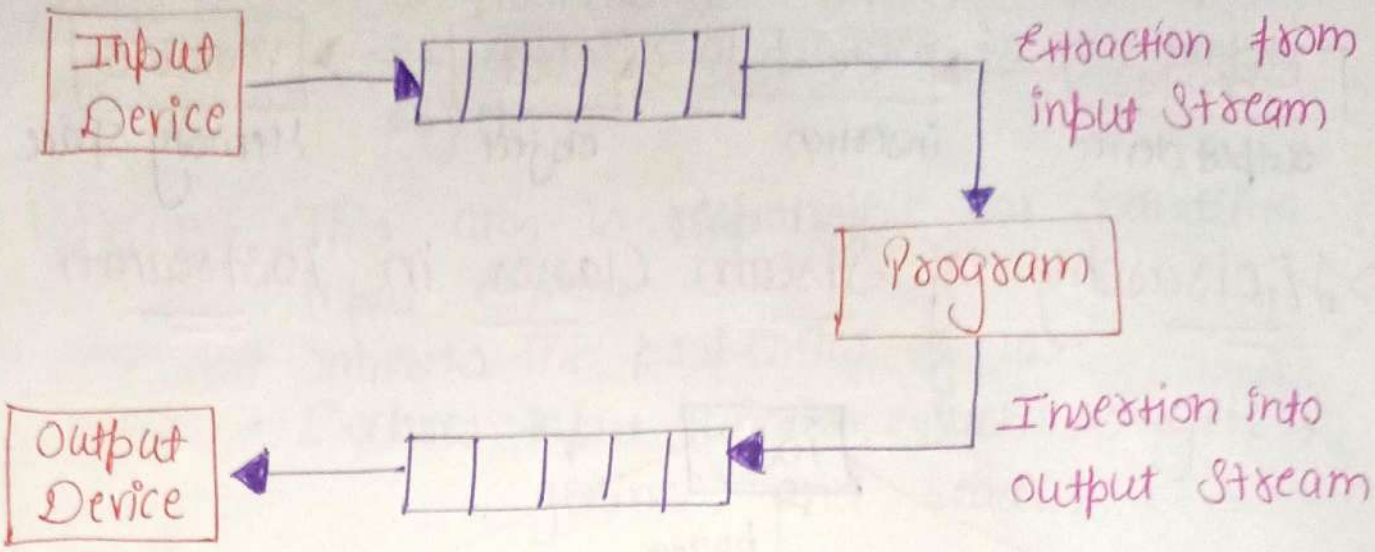
Constructor	Destructor
1) Constructor helps to initialize the object class.	1) whereas destructor is used to destroy the instances
2) It may or may not contain arguments	2) It cannot contain the arguments.
3) It can be inherited	3) It cannot be inherited.
4) It can be overloaded	4) It cannot be overloaded.
5) It has return type	5) It doesn't have any return type.
6) Constructor occupies memory	6) The Destructor releases Memory.
7) While declaring Constructor in C++, there is no requirement of the special symbol.	7) While declaring a destructor in C++, a particular symbol is required i.e tilde symbol.
8) We can use more than one constructor in our program.	8) We cannot use more than one destructor in the program.
9) Constructor has four types.	9) Destructors have no classes.
10) Syntax → Student(int a, int b)	10) Syntax → ~Student()

Hierarchy of Console Stream Classes

Streams :→ A Streams is a sequence of bytes

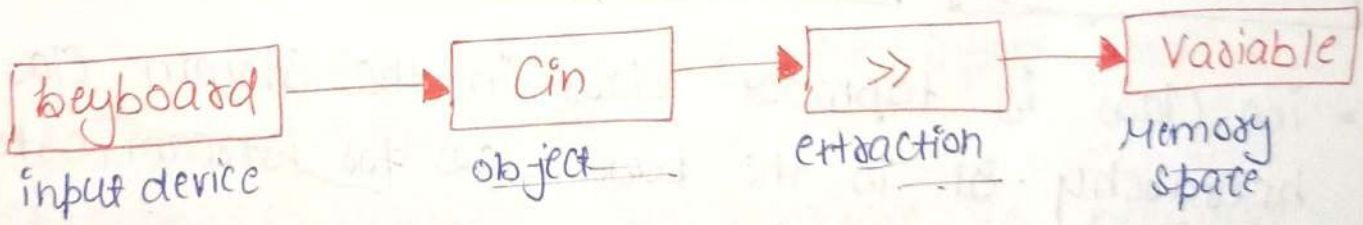
- Streams Classes are declared in header file "iostream.h".
- In C++ input and output are performed in the form of a sequences of bytes or more commonly known as streams.

Copyrighted by Codeleap.io
All rights reserved.



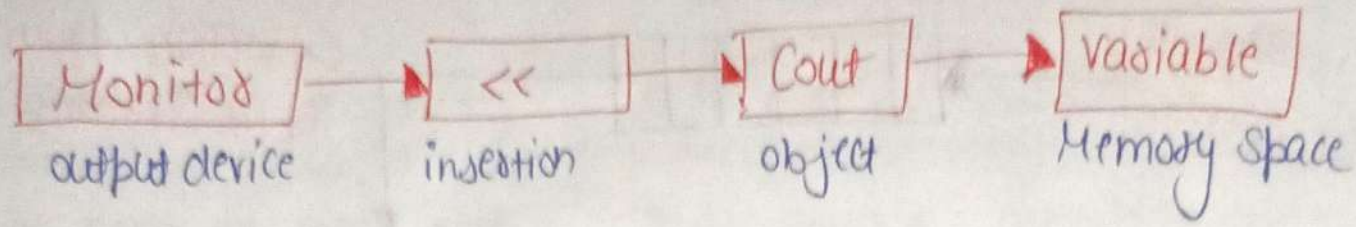
⇒ **Input Streams:** If the direction of flow of bytes is from the device (eg:→ keyboard) to the main memory then this process is called input streams.

- Cin Statement is the instance of the class istream and is used to read input from the input device
- (>>) extraction operator used for reading inputs

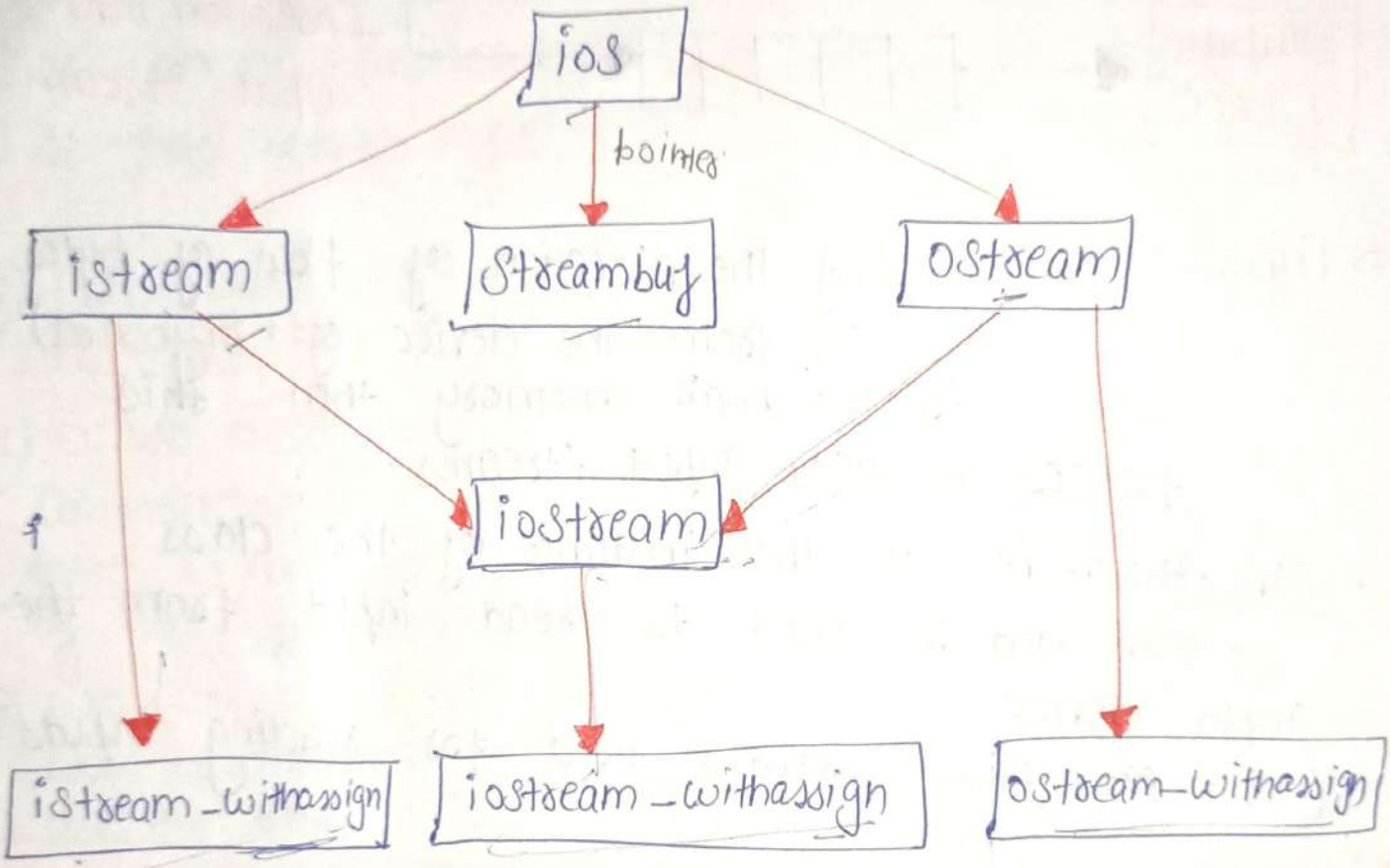


⇒ **Output Streams:** If the direction of flow of bytes is opposite. i.e from main memory to device (display screen) then this process is called output

- Cout Statement is the instance of the ostream class. It is used to produce output from the display screen.
- (<<) insertion operator used for produce output



⇒ Hierarchy of Stream Classes in iostream.h



- ios Class is topmost Class in the Stream Class hierarchy. It is the base Class for istream, ostream and streambuf Class.
- istream and ostream serves the base of iostream Class. The Class istream is used for input and ostream for the output
- The _withassign are provided with extra functionality for the assignment operations

Copyrighted by Codeleap.io
All rights reserved.

1) ios Class :→ • It provides the input & output facilities
• It contains a pointer to a buffer object.

2) istream :→ • This class is responsible for handling input stream.
• Inherits the properties of ios.
• Declares input function such as get(), getline() and read().
• Contains overloaded extraction operator (>>)

3) ostream :→ • This class is responsible for handling output stream
• Inherits the properties of ios.
• Declares input functions put() and write()
• Contains overloaded insertion operator (<<)

4) Iostream Class :→ This class is responsible for handling both istream and ostream class is inherited into it.
• also contains all the input and output functions.

5) Streambuf :→ • Provides an interface to physical devices through buffers
• Act as a base for filebuf class used ios files

6) istream_withassign :→ It is derived from istream class and used for input using cin object

7) ostream-withassign :→ It is derived from ostream class and is used while generating output using cout

8) iostream_withassign:- Combination of both istream_withassign and ostream_withas and it can also called bidirectional system.

formatted I/O in C++

The ios class contains several member function that are used to perform formatted I/O operations.

1) width()
2) Precision()
3) fill()

1) Width():- The width() function is used to define the width of the field for the output of an item.

- It is invoked with the help of cout object

Syntax:-> cout.width(w);

Example:-> cout.width(5);

cout << 253 << 14;

output -> | | | 2 | 5 | 3 | 1 | 4 |

2) Precision():- It is used to set the number of the decimal point to a float value

- It can be done using the precision() member function.

Syntax:-> cout.precision(d);

Example:-> cout.precision(2);

cout << 15.25 << endl;

cout << 5.40 << endl;

Copyrighted by Codeleap.io
All rights reserved.

3) fill() → It is used the blank spaces in output with given character.

- By default, unused positions are filled with whitespaces.

Syntax :- cout.fill(ch);

Example :- cout.fill('*');
cout.width(5);
cout << 25;

output :-

*	*	*	2	5

4) setf (format flags) :- Used to set various flags for formatting output like showbase, showpos etc.

5) unsetf(format flags) :- Used to clear the format flag setting.

format flag [iomanip.h] → header file.

1) endl → Used to move the cursor position to a newline.
2) ends → Used to point a blank space
3) dec → Used to set the decimal flag.
4) oct → Used to set the octal flag.
5) hex → Used to set the hexadecimal flag.
6) left → Used to set the left alignment flag
7) right → Used to set the right alignment flag
8) showbase → Used to set the showbase flag.
9) noshowbase → Used to set the noshowbase flag

Unformatted I/O

>> (extraction operator) and << (insertion operator)

overloaded in istream class

overloaded in ostream class.

• Syntax:

cin >> var1 >> var2 >> var3;

• Syntax

cout << var1 << var2 << var3;

→ put() and get() function

• The istream and ostream classes define two member function get() and put() respectively to handle the single character input / output operations

→ get() functions

• There are two type of get() functions.

• get (char*)
• get (void)

] input characters.

→ Ex: cinget(c);

→ put() function

• can be used to output a line of text, character by character.

• The put() function is used to help of the cout object

Ex:- Cout.put(c);

getline () functions

→ reads a whole line of text that ends with a newline character.

• cin. use.

Copyrighted by Codeleap.io
All rights reserved.

#Manipulators

- The unformatted I/o statements that it is impossible to display output in a required user format or input the value in the desired form. In certain situations, we may need to format the I/o as per user requirements.

  Eg:- 1) The square root of a number should be displayed upto 2 decimal numbers.

- To overcome the problem of unformatted I/o operations in C++, the concept of manipulators was introduced.
- Manipulators are helping function that can modify the input / output stream.
- It only modifies the I/o stream using insertion (<<) and extraction (>>) operators

  << → output form.

  >> → Input form

- Manipulators are operators that are used to ~~display~~ format the data display.
- iomanip.h The file should be included in the program
- In a single I/o statement, we can have more than one manipulator.

```
1 | cout << manip1 << var1 << manip2 << var2;
2 | cout << manip1 << manip2 << var1;
```

→ Types of Manipulators

1) Manipulators without arguments:

Iostream header file is used.

→ (oct, hex, end etc) are used in Manipulators without arguments.

- endl :→ '\n'. It is defined in ostream
  - It is used to enter a new line
  - iostream header file uses
  - It can also be used with all types of data.

- ends :→ • It is defined in ostream.
  - Add a null character into the output stream.

- dec, oct, hex :→ • Set the desired number system
  - iostream header file uses.
  - only used with integer data types

- flush :→ • It is defined in ostream.
  - it flushes the output stream.

2) Manipulators with arguments:

Iomanip.h header file is used.

<iomanip.h>

→ setw(w), setfill(n), setprecion(n), setbase(b) are used in Manipulator with arguments.

- setw(w) :→ • It is defined in ostream.
  - iomanip header file uses
  - also used in all type of data

Syntax • setw(int n)
  - Used to set the filed width

Ex:- Cout << setw(5) << x << setw(5) << y;

2) Setfill (c) : ↳ · fill the whitchspace with character c
· iomanip.h header files uses
· also be used with all type of data

Syntax · Setfill (Char ch)

3) Setprecision (n) : ↳ Set floating point precision to n
· iomanip header file uses.
· only used in float data type.

4) Setbase(val) : ↳ It is used to set the numeric base value for numeric values

Purpose

· It is Mainly used to make up the program Structure
· To carry out the operation of <iomanip.h>
· Specially designed to insertion (<<) and extraction (>>) operators.
· Manipulators are used to changing the format of parameters on Streams

friend function

· friend function is a function which is not the member of the Class, then the it can access the private and protected member of class

Se
· It is declared in the Class with friend keyword.
· It can be declared either in private or public part.

Copyrighted by Codeleap.io
All rights reserved.

Syntax:

```
Class Class_name
{
  friend data-type function-name (argument);

};
```

- When friend function is called neither name of object nor dot operation is used.
- A friend function ~~f~~ can be:
  a) A Member of another Class
  b) A global function.

Program

Copyrighted by Codeleap.io
All rights reserved.

friend Class

- If a Class beame a friend Class then it can access all the function i.e private and protected return of that class.
- Same friend keyword is used

Syntat:

```
Class A
{
  public:
    member function()
    {
      =
    }
    friend Class B;
};
Class B
{
  =
  public:
    Member function (A ob)
    {
      =
    }
};
```

program

Copyrighted by Codeleap.io
All rights reserved.

Operators in C++

- An operator is a symbol that operates on a value to perform specific mathematical or logical computation

Ex:→ · operands → Define variable → {C, A, B}

· operators → Perform some value → {=, +}

$$\boxed{C = A + B}$$

- C++ provides the following Types of operators

→ Arithmetic operators
→ Relational operators
→ Logical operators
→ Assignment operators
→ Conditional operators
→ Increment / Decrement operators
→ Bitwise operators.

Copyrighted by Codeleap.io
All rights reserved.

1) Arithmetic operators :→ These operators are used to perform arithmetic / Mathmati operations on operands

Ex:→ +, −, *, /, %.

Copyrighted by Codeleap.io
All rights reserved.

operator	operation	Example
+	addition	x+y
−	Substraction	x−y
*	Multiplication	x*y
/	Division	x/y
%	Modulo	x%y

Program

```
#include <iostream>
using namespace std;
int main() {
a = 7;
b = 2;
 //printing the sum of a and b
  cout << "a+b =" << (a+b) << endl;
 // printing the difference of a and b
  cout << "a-b =" << (a-b) << endl;
 // printing the product of a and b
  cout << "a*b =" << (a*b) << endl;
 // printing the division of a by b
  cout << "a/b =" << (a/b) << endl;
 // printing the Modulo of a by b
  cout << "a%b = " << (a%b) << endl;
return 0;
}
```

Output :-> a+b = 9
a−b = 5
a*b = 14
a/b = 3
a%b = 1

Copyrighted by Codeleap.io
All rights reserved.

2) Relational operators :→ These are used for the comparison of the value of two operands.

Ex:→ ==, >=, <=, >, <

```
int a=3;
int b=5;
a<b; // operator to check if a is smaller than b
```

- also check the Boolean value, i.e True or false

Here, $\boxed{0}$ denotes false and $\boxed{1}$ denotes true

operator	operation	Example

• ==	is equal to	int a=3, b=6;
		a==b;
		// returns false

• >	greater than	int a=3, b=6;
		a>b;
		// returns false

• >=	greater than or equal to	int a=3, b=6;
		a>=b;
		//returns false.

• <	less than	int a=3, b=6;
		a<b;
		// returns True.

• <=	less than or equal to	int a=3, b=6;
		a<=b;
		// returns true.

!=	Not equal to	int a=3, b=6; a!=6; → true.

Program

Copyrighted by Codeleap.io
All rights reserved.

```
#include <iostream>
using namespace std;
int main()
{
  int a=6, b=4;
   // Equal to operator
    cout << "a =b is" << (a = =b) << endl;
   // Greater than operator
    cout << "a >b is" << (a >b) << endl;
   // Greater than or equal to operator
    Cout << " a >=b is" << (a >=b) << endl;
   // Less than operator
    Cout << "a <b is" << (a<b) << endl;
   // Less than or equal to operator
    Cout << "a <=b is" << (a <=b) << endl;
   // true
    cout << "a != b is" << (a != b) << endl;
    return 0;
}
```

Output:

```
a = =b is 0
a >b is 1
a >=b is 1
a <b is 0
a <=b is 0
a != b is 1
```

3) Logical operators: } Logical operators are used to combining two or more condⁿ.

- The result of the operation of a logical operator is a Boolean value either True or false.

operator	operation	Example
&&	Logical AND	a=3, b=6; [a&&b;] // returns true.
||	Logical OR	int a=3, b=6; [a||b;] // return true
!	Logical NOT	int a=3; [!a;] // returns false.

program

Copyrighted by Codeleap.io
All rights reserved.

```
# include <iostream>
using namespace std;
int main()
{
    int a=6, b=4;
  // logical AND operator
    cout << "a&&b is" << (a&&b) << endl;
  // logical OR operator
    cout << "a || b is" << (a>b) << endl;
  // logical NOT operator
    cout << "!b is" << (!b) << endl;
  return 0;
```

Output:
a&&b is 1
a || b is 1
!b is 0

Ex: i) $a > b \&\& a > c$

→ if both condⁿ are true then give true. otherwise not.

ii) $a > b \| a > c$

→ if one condⁿ is true then our result is true.

4) Assignment operators: These operator are used to assign value to a variable.

- The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value.

Ex: a = 10;

Assignment operator

a → variable

10 → value.

→ a += b; = a = a+b;

→ a -= b; = a = a-b;

Operator	Operation	Example.
=	Assignment operator	a = b; , a = b;
+=	Add and assignment	a += b; , a = +b;
-=	Subtract and assignment	a -= b; , a = a-b;
*=	Multiply and assignment	a *= b; , a = a * b;
/=	Divide and assignment	a /= b; , a = a / b;
%=	Modulo and assignment	a %= b; , a = a % b;

Program

Copyrighted by Codeleap.io
All rights reserved.

```
#include <iostream>
using namespace std;
int main()
{
    int a=6, b=4;

    // assignment operator
    cout << "a= " << a << endl;
    // add and assignment operator
    cout << "a+=b is" << (a+=b) << endl;
    // Substract and assignment operator
    cout << "a -=b is" << (a-=b) << endl;
    // Multiply and assignment operator
    cout << "a*=b is" << (a*=b) << endl;
    // Divide and assignment operator
    cout << "a /=b is" << (a/=b) << endl;
    return 0;
}
```

Output:

```
a =6
a+=b is 10
a -=b is 6
a*=b is 24
a /=b is 6
```

5) Conditional operator: The conditional operator is kind of similar to the if-else statement as it does follow the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statement

in the shortest way possible.
· It is also known as Ternary operator
· Symbol → (?. :)
· Syntax →

Variable = Expression ?. true Statement : false Statem;

Ex: int a = (b > c) ?. b : c;

Program

Copyrighted by Codeleap.io
All rights reserved.

```
# include <iostream>
using namespace std;
int main()
{
    int a=3, b=4;
    //Conditional operator
    int result = (a<b) ? b : a;
    cout << "The greatest number is" << result << endl;
    return 0;
}
```

Output
The greatest number is 4

5) Increment / Decrement operator → increment operator (++) is used to increase the value of the operand by 1. whereas the decrement operator (--) is used to decrease the value of operand by 1.

Ex: x++; // Same as x = x+1
x--; // Same as x = x-1

6) Bitwise operators: The Bitwise operators are used to perform bit level operations on the operands.

- The operators are first converted to bit-level and then the calculation is performed on the operands.

Operator	Description
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise XOR operator
~	Bitwise one complement operator
<<	Left Shift
>>	Right Shift

→ Only Char and int data types can be used with Bitwise operators

Precedence and Associativity Rules

→ Precedence: Determines which operation is performed first in an expression with more than one operators with different precedence.

- also determine the evaluation of operators in absence of brackets.

Ex: $10 + 20 * 30$

addition operator ← lower precedence

higher precedence → Multiplication operator

$10 + 20 * 30$

Multiply will happen first has higher precedence.

$10 + 600$

Now addition with lower precedence

- $10 + 20 * 30 \rightarrow 10 + (20 * 30)$ ✓
  $\rightarrow (10 + 20) * 30$ ✗

- Operators Associativity: ↳ It is used when two operators of same precedence appear in an expression.
- Associativity can be → Left to Right
  → Right to Left

Ex: '*' and '/' have same precedence and their associativity is Left to Right. So the expression "100/10 * 10" is treated as "(100/10) * 10"

→ "100/10 * 10" ↳ Right Method.
"(100/10) * 10"

Copyrighted by Codeleap.io
All rights reserved.

Division operator ← Higher associativity ↓ Lower associativity ↓ → Multiplication operator

100 / 10 % 10

Divide will first higher associativity

10 % 10

Now multiply with lower associativity

100

Copyrighted by Codeleap.io
All rights reserved.

Operators precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expression in absence of brackets.

Ex:→ 100 + 200 / 10 − 3 % 10

→ Associativity is only used when there are two or more operators of same precedence.

→ All operators with the same precedence have same associativity

→ Precedence and associativity of postfix (++) and prefix (++) are different.

→ Postfix (++) is more than Prefix (++) in $\boxed{\text{Precedence}}$

→ Postfix (++) is left to Right
→ Prefix (++) is Right to left) > associativity

Category	Operator	Associativity
Postfix	() [] . ++ --	Left to right
Unary	+ - ! ~ ++ --	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /=	Right to left
Comma	,	Left to right

Copyrighted by Codeleap.io
All rights reserved.

Category	Operator	Associativity
Postfix	() [] . ++ --	Left to right
Unary	+ - ! ~ ++ --	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /=	Right to left
Comma	,	Left to right

Copyrighted by Codeleap.io
All rights reserved.

Operator overloading

Overloading

- If we create two or more member having the same name but different in number or type of parameter, it is known as overloading.

→ In C++, we can overload:

- Method
- Constructors
- indexed properties

- It is because these members have parameters only.

→ Type of overloading in C++

1) function overloading
2) operator overloading

⇒ Operator Overloading

- Operator overloading is a compile - time polymorphis in which the operator is overloaded to provide the special Meaning to the user- defined datatype.
- operator overloading is used to overload or redefines most of the operators available in C++.
- Operator overloading is used to perform the operation on the user - define datatype.
- The advantage of operator overloading is to perform different operation on the same operand.

Copyrighted by Codeleap.io
All rights reserved.

- C++ operator that cannot be overloaded.
  - → Scope resolution operator (::)
  - → Size of operator
  - → Member Selection (.)
  - → Member pointer operator (*)
  - → ternary operator (? :)
- Syntax of operator overloading

```
return_type Class_name :: operator op (argument_list)
{
    // body of function
}
```

→ return-type: is the type of value return by 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

1) Existing operators can only be overloaded, but the new operators cannot be overloaded.
2) The overloaded operator Must have the atleast one operand of user defined type.
3) It follows syntax rules of original operator. This Means we can't change the basic Memory of operator.

4) We cannot use friend function to overloaded certain operators. However, the member function can be used to overload those operators.

5) When unary operators are overloaded through a Member function takes no explict arguments, but if they are overloaded by a friend function, take one argument

6) When binary operators are overloaded through a member function takes one explict argument and if they are overloaded through a friend function takes two arguments.

- operators can be overloaded
  - Unary operators
  - Binary operators.

Unary operators:

A operator which contain only one operand is called unary operators overloading.

- They are pre-increment and post increment.
- These are some of examples, unary operators:
  1) increment operator (++)
  2) Decrement operator (--)
  3) Logical not operator (!)
- In unary operator function, no arguments should be passed it works only with one Class objects

Copyrighted by Codeleap.io
All rights reserved.

Syntax: return type operator op ( )
{
body;
}
← Class function

- friend return type operator op (arg list);
→ friend function.

→ operator is a keyboard.
→ op, we perform any arithmetic operation
→ In friend function, we need to pass argument list because friend function code is use outside the class [then we use [i]].
→ implement

Program

```
#include <iostream>
using namespace std;
class Space
{
int x;
int y;
int z;
public:
    void getdata (int a, int b, int c);
    void display (void);
    void operator-();  // operator overload declare.
};
void Space::getdata (int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
```

Copyrighted by Codeleap.io
All rights reserved.

```
}
void space::display(void)
  {
   cout << x << "\n";
   cout << y << "\n";
   cout << z << "\n";
  }
void space::operator() // Define overloaded function.
  {
   x = -x;
   y = -y;
   z = -z;
  }
int main()
 {
 space s;
 s.getdata(10, -20, +30);
 cout << "s : ";
 s.display();
 -s;          // operator overloading
 cout << "s : ";
 s.display();
 return 0;
}
```

output:

```
s :
x = 10
y = -20
z = 30
s :
x = -10
y = 20
z = -30
```

Unary Minus (-)

The minus operator changes the sign of its argument. A positive number becomes Negative, and a Negative no. becomes positive.

```
int a = 10;
int b = -a;  // b = -10
```

2) Binary operator overloading :- A operator which contain two operands is called Binary operator overloading.

- It is a polymorphic compile technique where a single operator can perform various functionalities by taking two operands.
- There are multiple binary operators like +, −, ∗, / etc. than can directly manipulate or overload the object of a class

Ex:- Suppose we have two number 5 and 6 and overload the binary (+) operator.
So, the binary (+) operator add the no 5 and 6 returns 11.

→ 5 + 6 → 11

5 → Number

+ → operator Binary

Syntax: return type operator op (arg list)
{
body;
}

← Class function

// we can pass only single argument in Class function.

- friend return type operator op (Arg list)

// we can pass only two argument.

Copyrighted by Codeleap.io
All rights reserved.

Program

```
# include <iostream>
using namespace std;
class complex
{
  float x;
  float y;
public:
  complex()
  Complex(float real, float img)
  { x = real ; y = img ; }
  Complex operator + (Complex);
  void display (void);
};
Complex Complex :: operator + (Complex C)
{
  Complex temp;
  temp.x = x + C.x;
  temp.y = y + C.y;
  return (temp);
}
void complex :: display (void)
{
  cout << x << "+j" << y << \n";
}
int main()
{
  Complex C1, C2, C3;
  C1 = Complex (2.5, 3.5);
  C2 = Complex (1.6, 2.7);
  C3 = C1 + C2;
```

[a+ib]
→ x+jy

Copyrighted by Codeleap.io
All rights reserved.

```
Cout << "C1 = "; C1.display();
Cout << "C2 = "; C2.display();
Cout << "C3 = "; C3.display();
return 0;
}
```

Output
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.6

⇒ function overloading

- When there are multiple function with same name but different parameters then these function are said to be overloaded.
- function can be overloaded by change in no of arguments or/and change in type of arguments.

Syntax:

```
class class_name
{ public:
  void add ()
  {
  }
  void add (int a)
  {
  }
};
```

Copyrighted by Codeleap.io
All rights reserved.

- In function overloading "function" name should be same and the "argument" should be different

- Parameters should have a different type

```
add (int a, int b)
add (double a, double b)
```

- Parameters should have a different number

```
add (int a, int b)
add (int a, int b, int c)
```

- Parameters should have a different sequence of parameters.

```
add (int a, double b)
add (double a, int b)
```

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

- Rules

1) The function must have the same name.
2) The function must have different type of parameters.
3) Different set of parameter.
4) sequence of parameter are differents

Example

```
#include <iostream>
using namespace std;
void SUMNUM (int A, int B);
void SUMNUM (int A, int B, int C);
void SUMNUM (int A, int B, int C, int D);
int main()
{
```

```
    SUMNUM (1,2);
    SUMNUM (1,2,3);
    SUMNUM (1,2,3,4);
    return 0;
}
void sumNUM(int A, int B)
{
  cout << endl <<"SUMNUM is: "<< A+B;
}
  void SUMNUM (int A, int B, int C)
  {
    Cout << endl << "SUMNUM is: " << A+B+C;
  }
  void SUMNUM (int A, int B, int C, int D)
  {
    Cout << endl << "SUMNUM is: "<< A+B+C+D;
  }
```

Output

```
SUMNUM is 3
SUMNUM is 6
SUMNUM is 10
```

Inline function

- Inline is a request to the compiler (not a Command) to make a function as an inline function to reduce the overload (overhead) of function calling.

Copyrighted by Codeleap.io
All rights reserved.

- If Compiler treat a function as a inline function it substitute the code of function in a single line.
- Inline function should contain a single instruction

Syntax: inline return-type function name()

keyboard

```
{
    _____ // single instruction
}
```

Program:

```
# include<iostream.h>
# include< conio.h>
inline int fun()
{
    cout << "Hello World";
}
void main()
{
    clrscr();
    fun();
    getch();
}
```

Copyrighted by Codeleap.io
All rights reserved.

Output: Hello World

- The main use of the inline function in C++ is to save memory space.
- It can be used over the macros.
- We can use the inline function outside the class so that we can hide the internal implementation of the function

Advantage of inline function

1) We do not need to call a function, so it does not cause any overhead.
2) It does not require any stack.
3) An inline function is mainly benefical for the embedded system

Disadvantage of inline function

1) If we use too many inline function then the size of the binary file will be large.
2) Inline function may increase compile time.
3) Inline function might cause thrashing.

---

function overriding

- It is used to achieve run-time polymorphism
- Whenever we writing function in base and derived Class in such a way that function name parameter must be same called function overriding.

Syntax:

```
Class Class-name          → base Class
{
    void fun()
    {
    }
};
Class class-name : public base-Class
{                                      derived Class
    void fun()
    {
    }
}
```

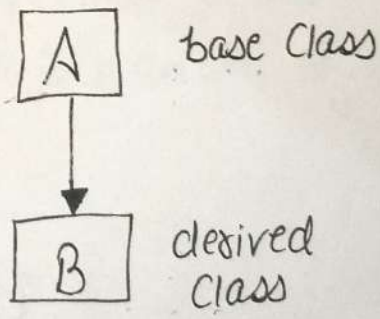
Inheritance :- Inheritance is a mechanism in which one Class inherits the property of other Class is known as inheritance

- To inherit from a Class, use the [:] symbol.
- Inheritance is one of the most important features of object-oriented Programming.

Inheritance

Derived Class | Base Class

- Derived Class :→ The Class which inherits the members of another Class
  subclass / child
- Base Class :→ The Class whose members are inherited.
  super / parent
- When we say derived Class inherits the base Class, it means, the derived Class inherits all the properties of the base Class, without changing the properties of base Class and may add new features.



- These new features in the derived Class will not affect the base Class.
- The derived Class is the specialized class for the base Class.

→ Syntax:

Class derived_Class_name : visibility_mode base_Class_name

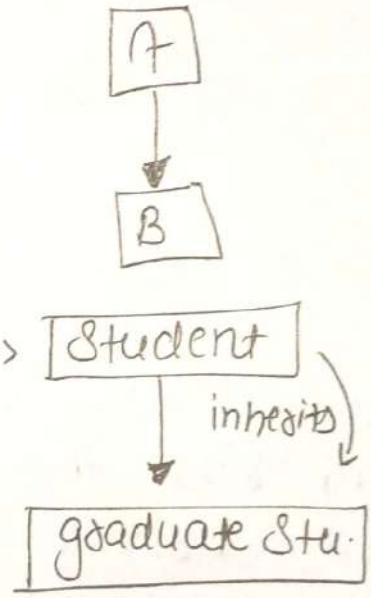
{
// body of the derived Class
}

Where

- Class :→ keyword to create a new Class
- Derived_Class_name :→ name of the new Class, which will inherit the base Class.
- visibility_mode :→ The visibility mode specifies wheather the features of the base Class. It can be public, private, protected.

Example :→

```
Class A
{
    public:
    void display()
    {
        - - -
    }
};
Class B : public A
{
    public:
    void show()
    {
        - - -
    }
};
```



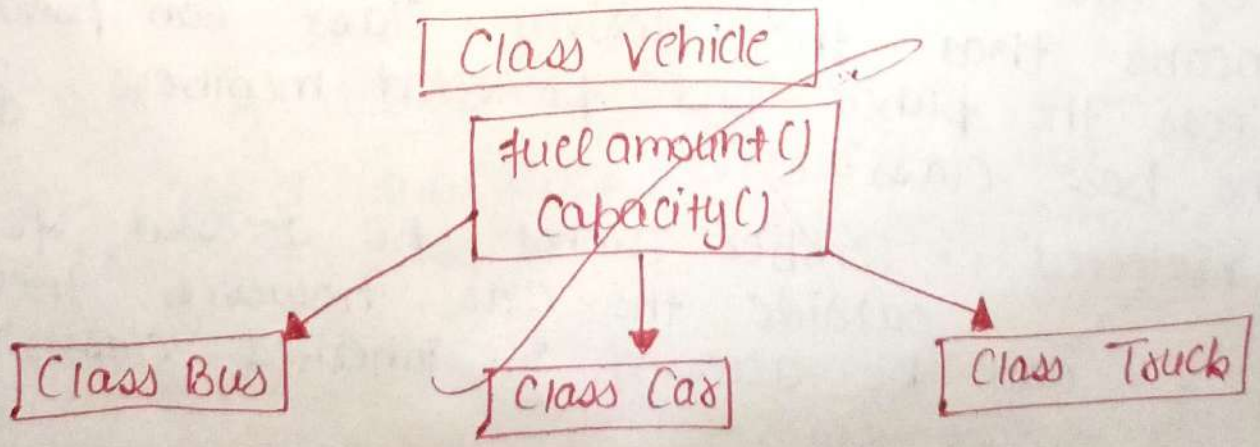
→ Advantage / features of Inheritance

1) Code reusability: one of the main reasons to use inheritance is that you can reuse the code.

eg:→ Conside a group of vehicles. you need to create classes for Bus, Car, Truck.

Class Bus	Class Car	Class Truck
fuel amount()	fuel amount()	fuel amount()
capacity()	capacity()	capacity()

- The Methods will be same for all three classes.
- Above 3 classes results in duplication of the same code 3 times. This increase the chances of error and data redundancy.
- To avoid this type of situation, inheritance is used.
- If we create a Class vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase reusability.



2) Save Time and effort.
3) Transitive nature
4) faster development, easier maintenance and easy to extend.

Modes of Inheritance

These are three Modes of inheritance.

1) Public
2) Private
3) Protected

1) Public :-> Members are accessible from outside the Class.
- Inside Class - Outside Class

- If the visibility modes is public, it means that the derived Class can access the public and protected members of the base Class, but not the private members of the base Class.

2) Private :-> Member cannot be accessed or view from outside the Class
- Member of Class
- friend of Class

- If the visibility Mode is private, that means that the derived Class can privately access the public and protected members of the base Class.

3) Protected :-> - Member cannot be accessed from outside the Class, however, they can be accessed in inherited classes
- Base Class
- Derived Class

- If the visibility mode is protected, that means the derived class can access the public and protected member of the base class protectively.

Access Control

- Access Modifiers / Specifier are same
- Access specifier define how the members of a class can be accessed.
- Access Modifiers are used to implement an important aspect of object-oriented progr. known as Data Hiding.

Syntax:
```
Class Classname
{
    public:
        Data member;
        Member function;

    private:
        Data member;
        Member function;

    protected:
        Data member;
        Member function;

};
```

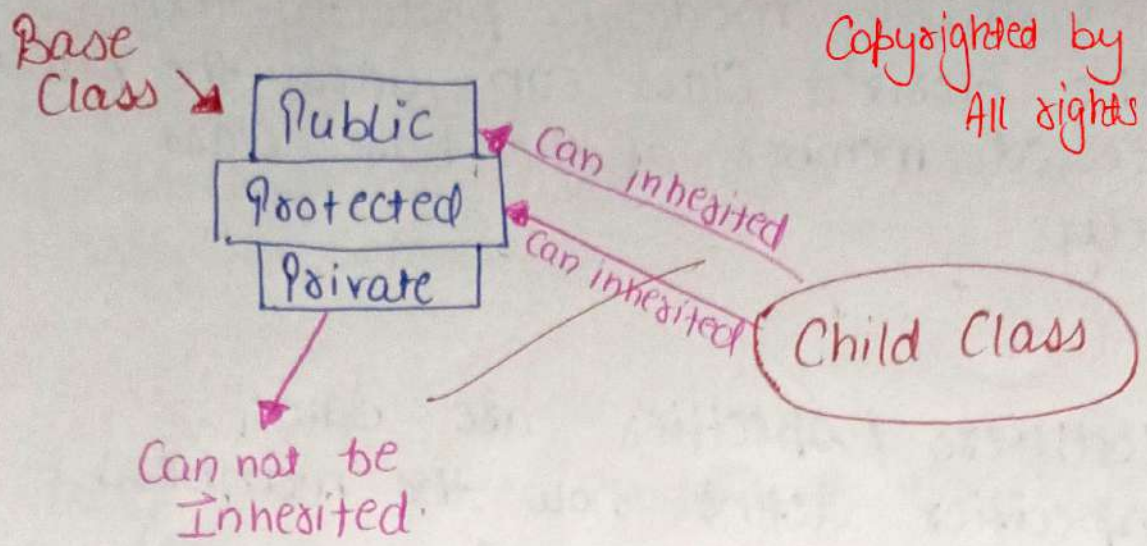
```
Class Base
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
Class Public Deri : public Base
{
    // x is public
    // y is protected
    // z not accesible
};
class protected Deri : pro Ba
{
    // x is proted
    // y is proted
    // z not acc
```

Role of access Specifiers

only affect wheather outsiders and derived class can access those inherited Members.

- private → Most Secure.
- protected → used only with inheritance
- final keyword → proted class from inheritance.

Copyrighted by Codeleap.io
All rights reserved.



Visibility of Inherited Members

Base Class Visibility	Derived Class Visibility		
	public derivation	private derivation	protected derivation.
private	Not inherited	Not inherited	Not inherited
protected	protected	private	protected
public	public	private	protected

#

Access Specifier	Class	derive	friend function
private	Accessible	Not	Accessible
protected	"	Accessible	"
public	"	Accessible	"

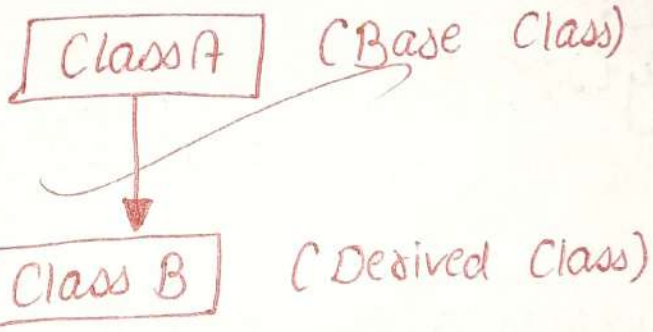
Type of Inheritance

1) Single inheritance 2) Multiple inheritance
3) Multilevel inheritance 4) Hierarchical inheritance
5) Hybrid inheritance.

1) Single inheritance :-> Single inheritance is the one where you have a single base class and single derived class

or

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



# Syntax: Class Subclass_name : access_mode base_class
{
//body of Subclass
};

Example:-> # include <iostream.h>
include conio.h>
ClassA
{
public:
void display()
{
Cout << "super class display function\n";
}

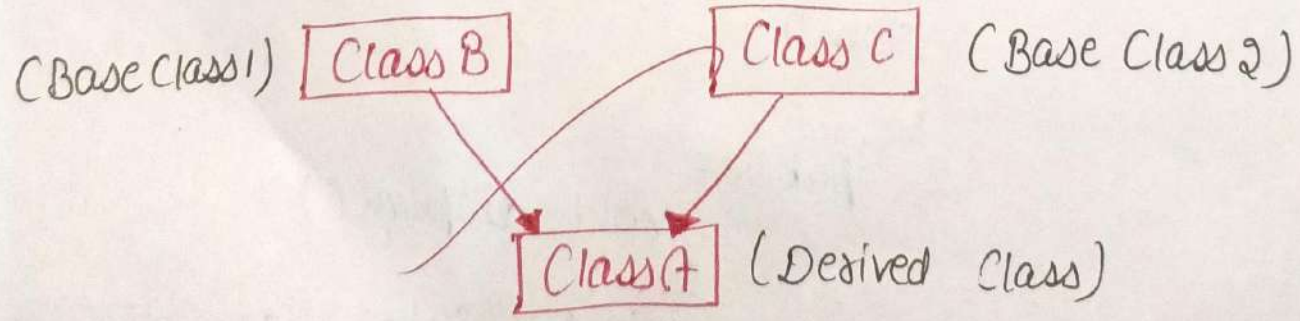
```
};
Class B : public A
{
 public:
     void show()
     {
        cout <<"Subclass show function\n";
     }
};
int main()
{
  B ob;
  ob.show();
  ob.display();
  getch();
}
```

Copyrighted by Codeleap.io
All rights reserved.

Output: Subclass show function
Super Class display function.

---

2) Multiple inheritance: → Multiple inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class



Syntax:

```
class subclass-name : access-mode base-class1,
                      access-mode base-class2, ...
{
    // body of subclass
};
```

or

```
class B
{ ......
};
class C
{ .....
};
class A : public B, public C
{
.......
};
```

{ Here, the no. of base classes will be separated by comma (',') . }

In Constructor

class C : public A, virtual A2

class C → derived; public A → base cons; virtual A2 → base cons

American Student Class Inherit → Student Class, American Class

Use of Multiple Inheritance

- When a subclass needs to combine multiple contracts and inherit some implementation.

or

allow a child class to inherit properties or behaviour from multiple base classes.

Advantage of Multiple Inheritance

- a class can inherit the functionality of more than one base class, flexibility
- when you want to save time by not writing the same code for diff. class

Disadvantage

- It can lead to a lot of confusion when two base classes implement a method with the same name., Order of inheritance, incr complexity

Example:

```
#include <iostream.h>
#include <conio.h>
 Class A                                  // SuperClass 1
 {
   public:
        void display ()
         {
           Cout << " super class A \n";
         }
 };
 Class B                                  // SuperClass 2.
  {
    public:
         void show()
          {
            Cout << " Super Class B \n";
          }
  };
  Class C : public A, public B            // Subclass C
  {                                        inherit A and B
   public:                                 Subclass properties
         void display1 ()
          {  cout << " Subclass C";
```

```
    }
};
int main()
{
    C ob;
    ob.display();
    ob.show();
    ob.display1();
    getch();
}
```

Copyrighted by Codeleap.io
All rights reserved.

Output: Super Class A
Super Class B
Sub Class C

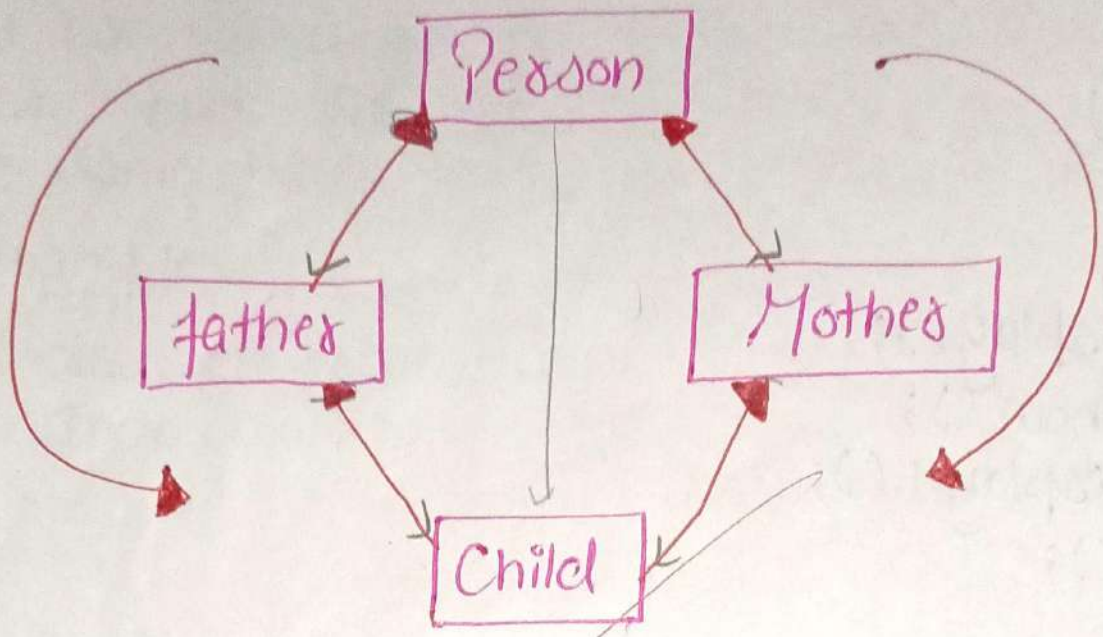
Problem with Multiple inheritance

Diamond problem

The Diamond problem is an ambiguity (i.e. duplicay of program or data) that aries in multiple inheritance when a function with the same name occurs in more than one base class

or

- When a child class inherits from two parent class who both share a common grandparent class.
- Here, we have a Child Class inheriting from classes father and Mother. These two classes, in turn, inherit the class person because both father and Mother are Person



- in the above figure, Class Child inherits of Class person twice - once from father and again from Mother. C++ Compiler is confused in selecting the Member function of a Class for the execution of a program. This scenario give rise to a diamond shaped inheritance.

Example:

```
#include <iostream.h>
#include <conio.h>
Class A
{
  public:
      void printmessage()
      {
        cout<<"Class A is printing"<<"\n";
      }
  };
Class B
  {
```

Copyrighted by Codeleap.io
All rights reserved.

```
public: void printmessage()
    {
      cout << "Class B is printing" << "\n";
    }
  };
Class C: public A, public B
 {
  public: void printmessage()
    {
      cout << "Class C is printing" << "\n";
    }
  };
 void main()
 {
  C m;
 m. printmessage();
 m. printmessage();
 m. printmessage();
 getch();
 }
```

output: C is printing
C is printing
C is printing

· In above program we printing, A and B also but in output only C is printing in three time. To create the duplicay in problem. for avoid this types of problem we use 2 ways

1) Scope resolution operator.
2) Use virtual base Class.

Program by using Scope resolution operator

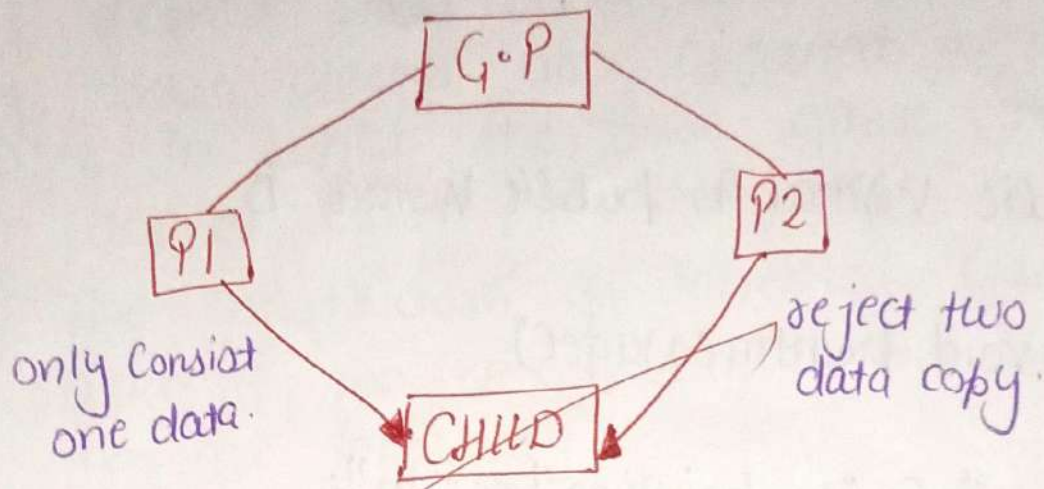
```
#include <iostream.h>
#include <conio.h>
Class A
{
 public:
        void printmessage()
        {
         cout << "Class A is printing" << "\n";
        }
  };
Class B
 {
  public: void printmessage()
     {
      cout << "Class B is printing" << "\n";
     }
 };
Class C: public A, public B
 {
  public:
     void printmessage()
     {
      cout << "Class C is printing" << "\n";
     }
 };
 void main()
 {
  C m;
clrscr();
m.A :: printmessage();
m.B :: printmessage();
m.C :: printmessage();
getch();
```

output:

Class A is printing
Class B is printing
Class C is printing

Virtual Base Class

To provide permanent solution of ambiguity problem.



[Single data copy only]

- Virtual can be written before or after the public. Now only one copy of data.
- member will be copied to Class C and Class B.

Class A becomes the Virtual Base Class

Syntax

```
Class B: virtual public A
{
 };
Class C: public virtual A
{
 };
```

Program

```
#include<iostream.h>
# include <conio.h>
Class A
{
  public: void Show_name()
   {
     cout <<"Class A is printing"<<"\n";
   }
```

Copyrighted by Codeleap.io
All rights reserved.

```
Class B : Virtual public A
 {
   public : void printmessage()
     {
       Cout << "Class B is printing" << "\n";
     }
 };
Class C : public virtual A, public ~~virtual~~ B
 {
   public : void printmessage()
     {
       Cout << " C is printing" << "\n";
     }
 };
void main()
{
  C m;
  Class C ();
 m.A :: Show_name ();
 m.B :: printmessage();
 m.C :: printmessage();
 getch();
}
```

Why we don't create base Class?

Ans→ C++ have doesn't a base Class because, having parametric polymorphism through templates, it doesn't need to

→ Programming language.

Which class Cannot be a base Class?

Constructor & friend function

↳ Not be inherited.

derived Class call Constructor.

Constructors and Destructors in M.I

- The constructors of the base class are executed first and then the constructor of derived class is executed. The constructors of base classes are executed in the same order in which the base classe are specified in derived class (i.e from left to right)

  [The constructors of inherited classes are called in the exact same order in which they are inherited]

- The destructors are executed in reverse order i.e derived class constructor is executed first and then constructor of the base classes are executed (from right to left)

  [destructors are called in the reverse order of constructors]

- If the constructors of the base classes have no parameters, then the syntax to create relationship b/w derived class and base classes is very simple.

Some facts

- In Multiple inheritance, the constructors of the base class and constructor of the derived class are automatically executed when an object of the derived class is created.

Copyrighted by Codeleap.io
All rights reserved.

Why the constructor of base Class is called first to initialize all the inherited members.

→ The data and member function of base class come automatically in derived Class based on the access specifier but the derivation of these member exist in base class only.

→ So when we create an object of derived Class, all of the members of derived Class must be initialized but the inherited member in derived Class can only be initialized by the base class's constructor as the defination of these members exists in base class only.

Order of execution of Constructor and destructor in inheritance

- C++ Constructor call order will be from top to down that is from base Class to derived Class and C++ destructor call order will be in reverse order.

The order is:

1) Base Constructor
2) Derived Constructor
3) Derived Destructor
4) Base Destructor

Class C (Base Class
↓
Class B (Base Class)
↓
Class A (Derived Class
↓

C() Cons	~(A) Des
B() Con	~(B) B
A() Con	~(C) B

Example:
```
#include <iostream.h>
#include <conio.h>
class A
{
public:
```

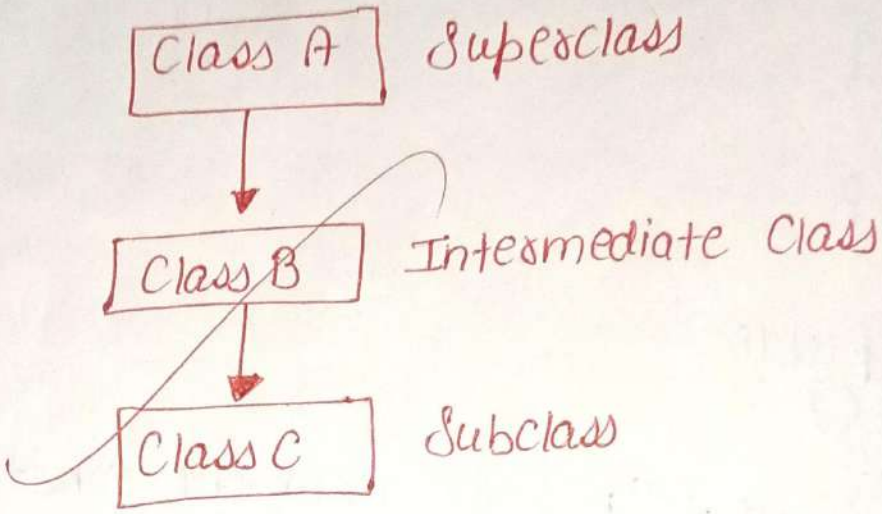
```
A()
{
  cout << "\n zero argument constructor of base class A";
}
~A()
{
  cout << "\n destructor of base class A";
}
};
Class B
{
  public:
  B()
  {
    cout << "\n zero argument constructor of base class B";
  }
~B()
  {
    cout << "\n destructor of base class B";
  }
};
Class C : public A, B
{
 public:
 C()
  {
   cout << "\n zero argument constructor of desired class C";
  }
~C()
 {
   cout << "\n destructor of class C";
  }
};  void main()
  {  cobj
```

output zero arg. const of b C a
" " " " " " b
" " " " desired c
destructor of Class C
" " b C B
" b C A

3) Multilevel Inheritance :-

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

- When one class inherits another class which is further inherited by another class. It is known as Multi-level inheritance in C++



Syntax :
```
Class A
{ .....
};
Class B : public A
{ .....
};
Class C : public B
{ .....
};
```

Copyrighted by Codeleap.io
All rights reserved.

Advantage : → Code reusability.

Program

```
# include <iostream.h>
# include <conio.h>
 Class A
 {
  public :
   void display()
   {
    Cout <<" Super Class display function \n";
   }
 };
 Class B : public A
 {
  public:
   void show()
   {
    Cout <<"intermediate show function \n";
   }
 };
 class C : public B
 {
  public:
  void display1()
  {
   Cout <<" subclass display1 function";
  }
 };
  void main()
  {
   classr();
   Cob;
```

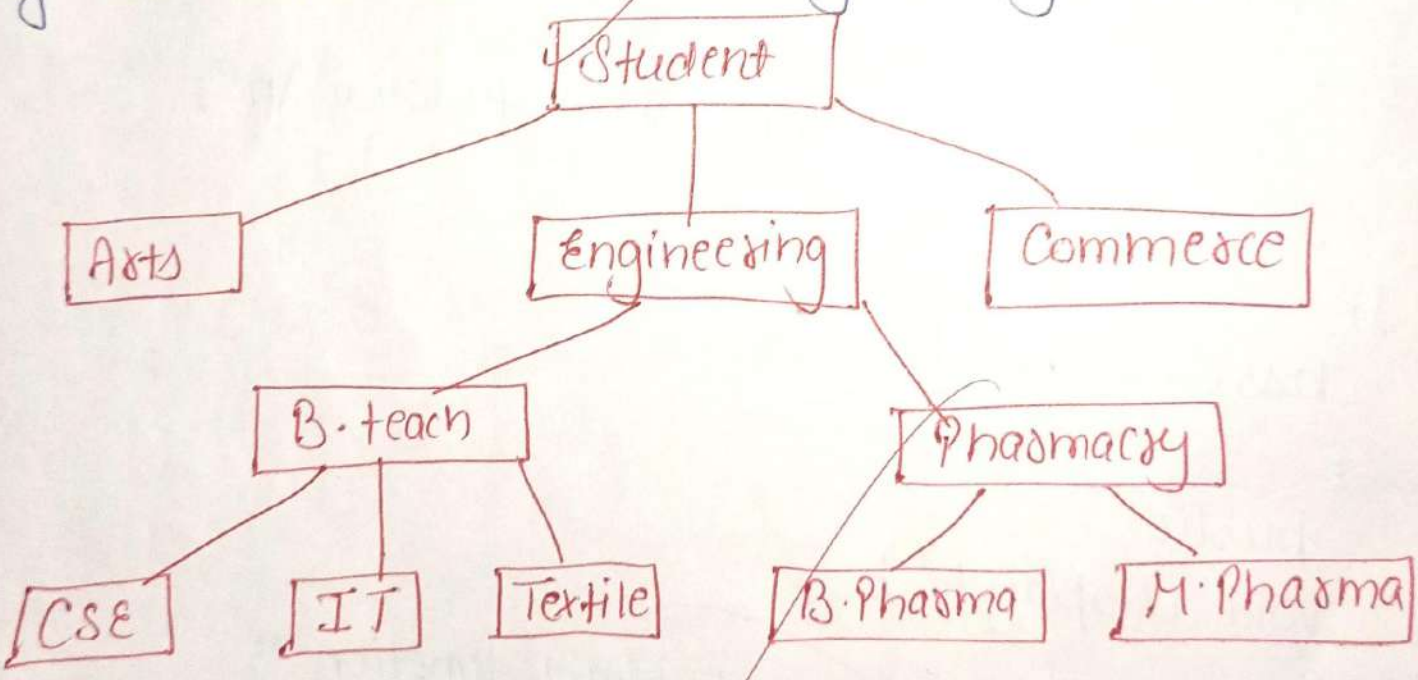
Copyrighted by Codeleap.io
All rights reserved.

```
    ob.display();
    ob.Show();
    ob.display1();
  getch();
}
```

Output: Super Class display function
intermediate Show function
Subclass display1 function

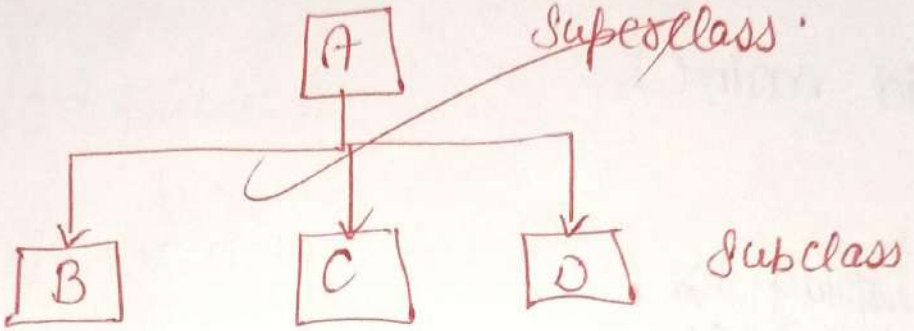
---

4) Hierarchical Inheritance: More than one Sub-class is inherited from a single base Class.
- It can be used where programming problems can be cast into a hierarchy certain features of one level are shared by many other level.



Syntax:
```
Class A
{
  ....
};
Class B : public A        // derived Class from A
{ ....
};
```

```
Class C: public A          // derived Class from A
 {
  ....
 };
Class D: public A          // derived Class from A
 {
  ....
 };
```



Program.

Copyrighted by Codeleap.io
All rights reserved.

```
#include <iostream.h>
#include <conio.h>
Class A
{
  public:
    void display()
    {
      Cout <<"Superclass display function \n";
    }
};
Class B: public A
{
  public:
    void Show()
    {
      Cout <<"Subclass Show Method \n";
    }
};
```

```
Class C : public A
{
  public:
     void display1 ()
     {
       Cout <<"Subclass display1 method \n";
     }
  };
    void main()
  {
   C ob;
   ob.display();
   ob.display1();
  getch();
  }
```

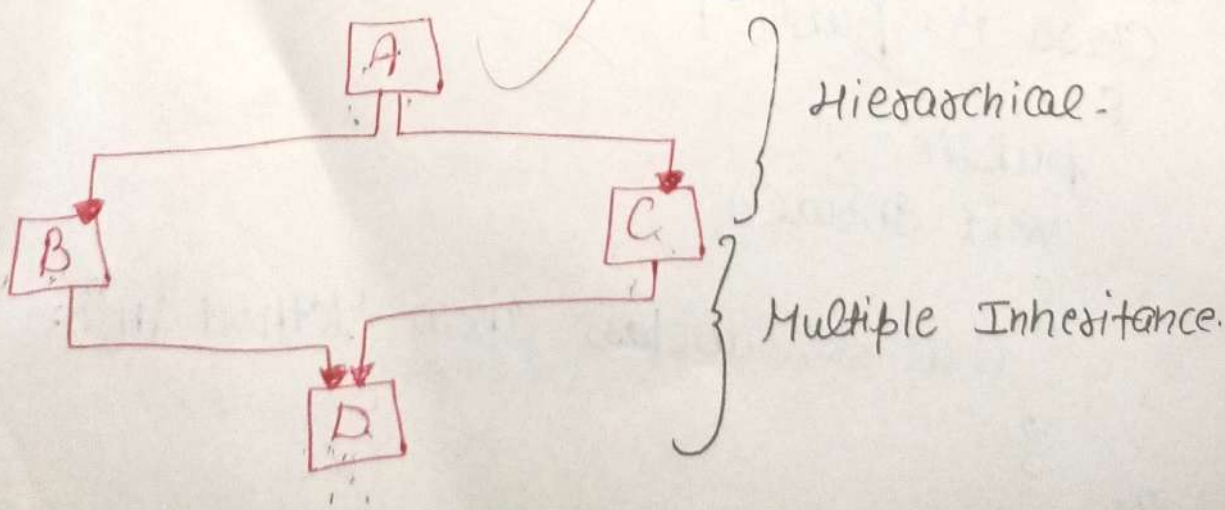
we create ob, C and A but no call ob.B
if you create B object then
B ob1;
ob1.Show();

Output: Super Class display function
Subclass display1 fun Method

5) Multipath or hybrid inheritance:-

The process of Combining more than one type of inheritance together while deriving Subclasses in a program.

- Multiple and Single inheritance and hierarchical inheritances are Combined together



Syntax :→ Class A

```
{
  Statement;
};
Class B : public A
{
  Statement;
};
Class C : public A
{
  Statement;
};
Class D : public B, public C
{
  Statement;
}
```

Copyrighted by Codeleap.io
All rights reserved.

Program.

```
# include <iostream.h>
# include <conio.h>
Class A
{
  public:
        void display ()
        {
            cout << "Super class display function\n";
        }
};
Class B : public A
{
  public:
```

```
  void show()
  {
    Cout <<"Subclass show method\n";
  }
};
Class C: public A
{
  public:
      void display1()
      {
        Cout<<"Subclass display1 function\n";
      }
};
```

// Hierachical inheritance
[A inherits B
A inherits C]

```
Class D: public B, public C
{
  public:
      void show1()
      {
        Cout <<" Sub-Subclass show1 method\n";
      }
};
```

[// create multiple
D access B and C]

```
 void main()
 {
   D ob;
   ob.show();          // B function
   ob.display1();      // A function.
   ob.show1();         // D function.
```

, member implies is

output: Subclass show Method.
Subclass display1 Method.
Sub-SubClass show1 Method

if we call display(); [// A function] then ambiguity problem create.

→ C and B access the property of A

→ Through D, we call B and C also call D through A but D through A, B

then we solve the problem to create b object or C object.

```
# B ob1;
  ob1.display();
}
```

} Solve ambiguity problem

→ output: → add super class display function

Advantage and Disadvantage of Inheritance

→ Code reusability

→ organization

→ Polymorphism

→ Overriding

→ Abstraction.

Disadvantage

→ Complexity

→ Rigidity

→ Dependency

→ Performance.

Copyrighted by Codeleap.io
All rights reserved.

Role of Constructors

- Constructors is used to initialize the object of the Class.
- A Constructors is a particular member function having the same name as the class name. It calls automatically whenever the object of the Class is created.
- Constructors is called by the compiler, it allocates the memory to the object and initializes class data members by default values or value passed by the user while creating an object whenever the object of the Class is created.
- Constructors don't have any return type because their work is to just create and initialize an object
- We can use Constructors in derived Class in C++
- If the base Class constructor does not have any arguments, there is no need for any constructor in the derived Class.
- But if there are one or more arguments in the base Class constructor, derived Class need to pass argument to the base Class constructor
- If both base and derived Classes have constructor, base Class constructor is executed first and then the constructor in the derived Class executed.

=> Constructors in Multiple inheritance

- In Multiple inheritance, base classes are constructed in the order in which they appear in the class declaration.

Ex:-> three Class A, B, C.

- C inherit Class A and B
- If Class A is write before B then the constructor of Class A will be executed first
- Similarly, in Class "B".

Role of Destructor

- Destructor is called by the compiler when the object is destroyed and its main function
- is to deallocate the memory of the object.
- Destructor is just the opposite function of the constructor.
- The destructor is called automatically by the compiler when the object goes out of scope.
- Destructor has the same as of the class with prefix tilde (~) operator.
- It cannot be overloaded as the constructor
- Destructors takes no argument and have no return type and return value

Genericity in C++

- Generics is the idea to allow type (Integer, String .. etc) user-defined types] to be a parameter to methods, classes and interfaces.

for ex:→ Class like an array, map which can be used using generics very efficiently. We can use them for any type.

- The Method of generic programming is implemen to increase the efficiency of the code. Generic programming enables the programmer to write a general algorithm which will work with all data types.
- It eliminates the need to create different algorithms if the data type is an integer, string or a character.
- Generic programming is an approach where generic types are used as parameters in algorithms so they work for a variety of data types.

· Generic in C++ allow you to write code that can work with a variety of data types. achieved using templates

⇒ Advantage

1) Code reusability, Maintainability
2) Avoid function overloading
3) Once written it can be used for Multiple times and Cases

⇒ How it is implemented?

- Generic programming is implemented using the templates in C++

Copyrighted by Codeleap.io
All rights reserved.

- Generics can be implement in Templates.
- Template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.

Example: a software company may need sort() for different data types. Rather than writing and maintaining the Multiple codes, we can write one sort() and pass data type as a parameter.

=> Generic function using Template

We write a generic function that can be used for different data type.

```
#include <iostream.h>
 template <Class T>
 T mymax ( T x, T y)
 {
   return (x>y)? x:y;
 }
 void main()
 {
  Clrscr();
  cout << mymax (3,7)<<endl;
  cout << mymax (4.3, 8.6)<<endl;
  cout << mymax ('a', 'b') <<endl;
  getch();
 }
```

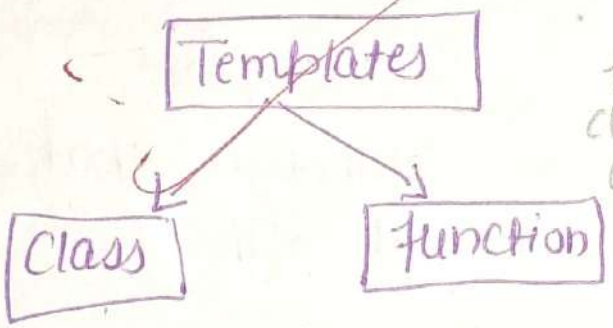
```
int mymax (int x, int y)
{
  return (x>y)? x:y;
}

Char mymax (Char x, y)
{ return (x>y)? x:y;
}
```

Output:
7
8.6
b

Templates in C++

- Template is the frame which define its actual meaning in C++. We can draw any logic using templates.
- At the time of execution, it will create appropriate code.
- Templates are powerfull features of C++ which allows you to write generic programs.
- keyword → template.
- Templates can be represented in two ways.



- Template is a blue print for a generic class or function that can be used with different type of data.

function Templates

- It is also known as generic function.
- We write a generic function that can be used for different data types. Example: of function templates:→ sort(), max(), min(), printarray()
- A generic function is created by using the keyword template

⇒ Syntax:

```
Template <ClassType>
return type  function name (parametrized_list)
{
    // body of function.
}
```

→ Where Type: It is a placeholder name for a data type used by the function. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

→ Class: A Class keyword is used to specify a generic type in a template declaration.

- Generic function define a set of operation that can be applied to the various types of data.

⇒ Advantages

1) Templates are type-safe.
2) Templates avoid some common errors found in code. use of function-like Macros.
3) increase efficiency.

⇒ Disadvantages

1) Template error are only detected by the compiler.
2) Maintenance of the template framework.

Uses

- It allows us to use the functionality of More than one type or Class without repeating the entire code for each type.

Difference

function template when you want to create a function than can operate on many different types.

Class Templates → when you want to create a Class that is parameterised by type.

- It is also known as generic Class.
- Class templates like functions templates, Class templates are useful when a Class defines something that is independent of the data type. Can be useful for Classes like linked list, Binary Tree, Array etc.

Syntax:

```
template <Class Ttype>
Class Class_name
{
  :
}
```

Program

```
#include <iostream.h>
#include <conio.h>
template <Class T>   [ <Class T1, Class T2> ]
Class Show
{
 T a,b;   [T1 a, T2 b;]
public:
      Show(T p, T q)   [ Show (T1 p, T2 q)
      {
        a=p; b=q;   [ a=p; b=q;
      }
      void Show()
      {
        Cout << "a=" << a << endl << "b=" << b << endl;
                 value of a and b are: " << a << ", << b << endl;
      }
```

```
};
void main()
{
 Show<float> ob(10,20);
 clrscr();
 ob.Show();
 getch();
}
```

```
Show <int, float> ob(5,6.5);
ob.Show();
getch()
```

Copyrighted by Codeleap.io
All rights reserved.

output:- a = 10
b = 20

Fnctions in C++ using Bubble Sort

```
#include <iostream>
 template < class T> void bubblesort (T a[], int n)
 {
     for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (a[j]<a[j-1])
                swap(a[j], a[j-1]);
 }
int main()
{
   int a[5] = {10, 50, 30, 40, 20};
   int n = sizeof(a) /sizeof(a[0]);
   bubble sort <int> (a,n);
  cout <<" Sorted array:";
   for (int i =0; i<n; i++)
      cout << a[i] << " ";
   cout << endl;
  return 0;
}
```

output:

Sorted array: 10 20 30 40 50

Syntax of Class Template with multiple Parameters

We can use more than one generic data type in a class template, and each generic data type is separated by comma.

→ Syntax

```
template <Class T1, Class T2>
Class Class_name
{
    // Body of Class
}
```

How do templates work in C++

Templates in C++ work in such a way that it gets expanded at compiler time, just like macros and allows a function or class to work on different data types without being rewritten.

or

The idea is simple, source code contains only function/Class, but compiled code may contain multiple copies of the same function/Class.

Difference b/w Class Template and function Template

- Class template when you want to create a class that is parameterised by a type.
- function template when you want to create a function that can operate on many different types.
- when instantiated, a class template becomes a class
- a function template becomes a function

Disadvantages

- Difficult to implement.
- Template error can only be detected by the compiler when the template is instantiated.
- Slow the compile time, when template are large and complicated.

Advantages

- remove code duplication
- help us to make generic callback.
- improve performance for code

Copyrighted by Codeleap.io
All rights reserved.

Exception handling

- Exception handling is a process to handle runtime errors. We perform exception handling so, the normal flow of program can be maintained even after runtime errors. [to separate the error handling code from the normal code] ← aim.

Syntax:

```
try
{
    // Code
    throw exception;
}
Catch (type arg) // Catches exception.
{
    // Code to handle exception
}
```

} block of statements which detects & throw exception.

} block of statement that handles the exception.

- Exception handling keywords

1) try :-> It is represent block of code that can throw an exception.

2) Catch :-> It is represent block of code that is executed when a particular exception is thrown. [if any error occurs in try block]

3) Throwh: It is used to throw an exception when a problem is detected, which lets us create a custom error.

for eg: User divides a no. of zero, this will x/0 Compiles successfully, but an exception on runtime error will occur due to which our application will be crashed
→ You can use the try-catch block to handle exception
→ In order to avoid this we'll introduce exception handling, teach in our code.

Exception handling mechanism

1) finding the problem [Hit the exception]
2) Informing that error has occured [Throw the except]
3) Receiving the error information [Catch exception]
4) Take proper action [Handle the exception]

Advantage of exception handling

- Separating error-handling Code from regular code
- A logical grouping of error types Exception can be used to group together errors that are related. [It maints the normal flow of application
→ It make prog readable and maintainable

Exceptions

Exceptions are the unexpected situations that arise during the execution of a program.

Why we need exception handling

1) Separation of error handling Code from Normal Code.
2) functions /Method can handle only the exceptions they choose.
3) Grouping of error type. [one of the most common run time exception can be 0 division error.

Errors can be broadly into two types

1) Compile Time errors
2) Run Time errors.

Error are the problem that occurs in program due to an illegal operation performed by user

1) Compile Time errors :→ These are the error that stop the program at compile time itself. - That are caught during compile time.
syntax

- Compile Time errors include library reference, syntax error or incorrect Class import

eg :→ Using string without including string library in the program headers

2) Run Time errors :→ These are called exceptions that can handle using try, catch and throw keywords
logic

eg :→ Dividing by 0

- these errors that cannot be caught during compile time.
- As we cannot check these errors during compile time, we name them Exception.

Types of Exceptions

1) Synchronous :→ Problem are created by users
2) Asynchronous: Problem are created by System.
   eg :→ Disk failure

Program

```
#include <iostream.h>
#include <conio.h>
void main()
{
    int age;
    cout << " enter your age:";
    cin >> age;
```

Copyrighted by Codeleap.io
All rights reserved.

```
try
  {
     if (age > 18)
     {
        throw age;
     }
  }
Catch (int age)
 {
   cout << "age less than 18" << endl;
 }
return 0;
}
```

output: enter your age : 21
age less than 18

Program :> Division by zero exception

```
# include <iostream.h>
  void main()
  {
     int a, b;
     cout << "Enter the value of a and b" << endl;
     cin >> a >> b;
     try {
          if (b! = 0)
               cout << a/b;
          else
               throw b;
     }
     Catch (int i)
```

```
        {
            cout<<" Division by zero:"<<i<<endl;
        }
    getch();
}
```

output: Enter the value of a and b

2

0

Division by zero : 0

[Catch Exception] #(2)

Try Block → Try Block contain the code that might thrown an exception (1)

It is followed by one or more catch block

```
try
{
    // try block code
}
```

Catch Block

Catch block contains the code that handles the exception.

```
Catch (exception_name i)
{
    // Catch block
}
```

have one or more catch block associated with try block

. Both the try and Catch blocks works together

Exception handling is the process of handling errors and exceptions such that the normal execution of the system is not halted

Copyrighted by Codeleap.io
All rights reserved.

Syntax

```
Try
  {
  // Block of Code to try
      throw parameter;
  }
Catch ( expection ex)
  {  // code to handle exception.
  }
```

[whenever some exception is found in try block, it is immediately thrown to catch block

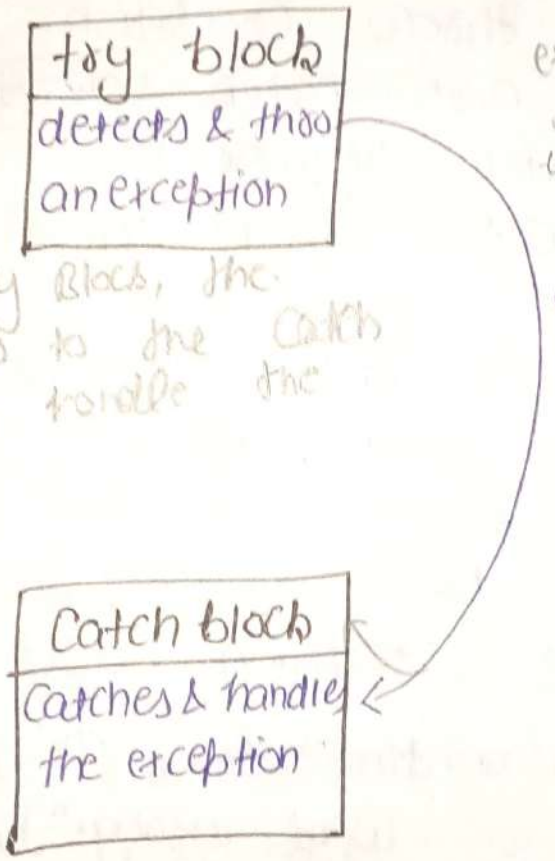
Throw

A program thrown an exception when a problem show up. This is done using a throw keyword. Exception can be thrown anywhere within a code block using throw statement

Exception object

[The block throwing exception]

#



. If an exception is thrown in try Block, the program jums to the catch block to handle the exception.

- One of the most common run time exception can be 0 division error. When we try to divide a number by 0, then the program will get executed successfully but during compile time, we will face an error causing an application crash.
- We use the exception handling in C++ is to separate the error handling code from the normal code.

How is exception handling in C++?

Exception handling in C++ is a mechanism that allows you to handle errors that occur during program execution in a structured way.

- When an error occurs, an exception is thrown, which can be caught and handled by the program.
- The basic idea behind exception handling is to separate the code that detects an error from the code that handles the error.
- In C++, you can throw exception using the 'throw' keyword and catch exceptions using the 'try' and 'catch' keywords.

How we can catch and throw an exception in C++

- you can catch and throw exception using the 'try', 'catch' and 'throw' keywords.
- To <u>throw</u> an exception, you use the 'throw' keyword followed by an expression that evaluates to an exception object.

eg:
```
throw std::runtime_error("Something went wrong.");
```

- To <u>catch</u> an exception, you use the 'try' keyword followed by a block of code in which you expect an exception to be thrown. You then use the 'catch' keyword followed by a parameter that specifies the type of exception you want to catch.

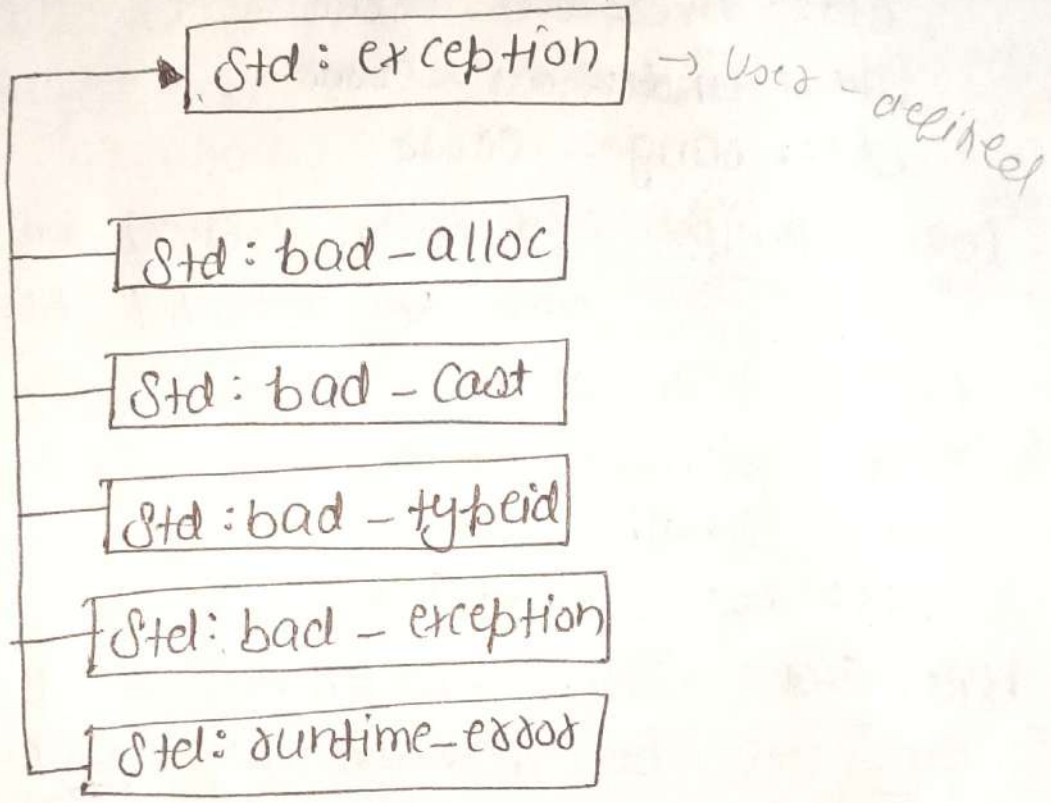
eg:- Syntax

Standard exceptions

- C++ provide a list of Standard exception defined in <exception> which we can use in our programs.
- C++ compilers can handle the all three types of exceptions:

  Std::bad_alloc
  Std::exception
  Std::runtime_error

#



- Std::exception: It is an base class for all Standard exceptions.
- Std::bad_alloc: This exception is thrown when a program fails to allocate memory using the 'new' operator.
  - This can occur when the system runs out of memory, or when the requested amount of Memory is too large.

3) Std:: runtime-error : This exception is thrown when a program encounter a runtime error that cannot be handled.

or

It is an exception that canot be detected by reading a code

- The 'Std:: runtime-error' class has several derived class

Std:: overflow-error
Std:: underflow-error
Std:: range-error

4) Std:: logic-failure : It is an exception that can be detected by reading a code.

Std:: domain-error
Std:: length-error
Std:: out-of-range

5) Std:: bad-cast : This exception is generally be thrown by dynamic-cast

6) Std:: bad-typeid : by typeid

File I/O in C++

- A file is collection of related records, a record is composed of several fields and field is a group of character.

Text file

- .txt files
- Text files are human-readable files that contain characters ~~that~~ ~~conta~~ that can be displayed on a screen, such as letters, no and symbols.
- Text files are typically used to store data that is meant to be read by humans.
- It stores data using ASCII format.
- Each line is terminated by special char called EOL
- eg → 13 number
  i.e for every symbol, digit or character there is some ASCII code available
  - The ASCII code of 1 is 49
  - The ASCC code of 3 is 51.
    - ASCII codes for digit 1 and 3.
- To read and write data to a text file in C++, we can use the 'fstream' library. The 'fstream' library provides classes for reading and writing files such as
  'ifstream' for reading input from a file.
  'ofstream' for writing output to a files

and 'fstream for both reading and writing.
eg: example.txt ["data.txt"]

→ Text file function

- get [read a single character]
- put [writing a single characters]
- getline

Binary file

- bin files
- Binary files are files that store data in a machine-readable format.
- each pieach of data is stored as a sequence of bytes, and there is no delimiter b/w the data (i.e eol)
- Binary files are commonly used to store data that needs to be read or written quickly by a program, such as image files, audio files and executable files

eg:- 13 is an integer value.
The binary form of 13 is '1101'.

- Stores information in the form of 0s and 1s
- eg:→ ["dada.bin"]

Copyrighted by Codeleap.io
All rights reserved.
Copyrighted by Codeleap.io
All rights reserved.

Text file	Binary files
• Stores information in ASCII Characters.	• Stores information in the form of 0s and 1s
• Can Store only plain text.	• can Store different type i.e audio text, image, in a Single file.
• Used to Store data Usere-friendly in text files	• Used to Store custom and compact data.

limitation of text file and it Sol

- Text files can only Store textual data, not binary data
- Text files can be easily corrupted if not handled properly.
- Text files have a limited capacity to Store data.

-> To overcome these limitations, you can use binary files in C++

- Binary files can Store both textual and binary data and are less prone to corruptions.
- Binary file also have a large capacity to Store data than text files.

Copyrighted by Codeleap.io
All rights reserved.

file pointers

- Each file has two associated pointer known as file pointers
- One is the input pointer → get pointer
- and other is output pointer → put pointer
- Pointers are used to move through the files while reading or writing.
- Input pointer: is used for reading the contents of a file.
- Output pointer: is used for writing to a file.

→ function of file pointers

All the action in the file pointers take place automatically

- If we want to provide control on our own, we can do that by using function.

→ Seekg() ⇒ move get pointer to specified location

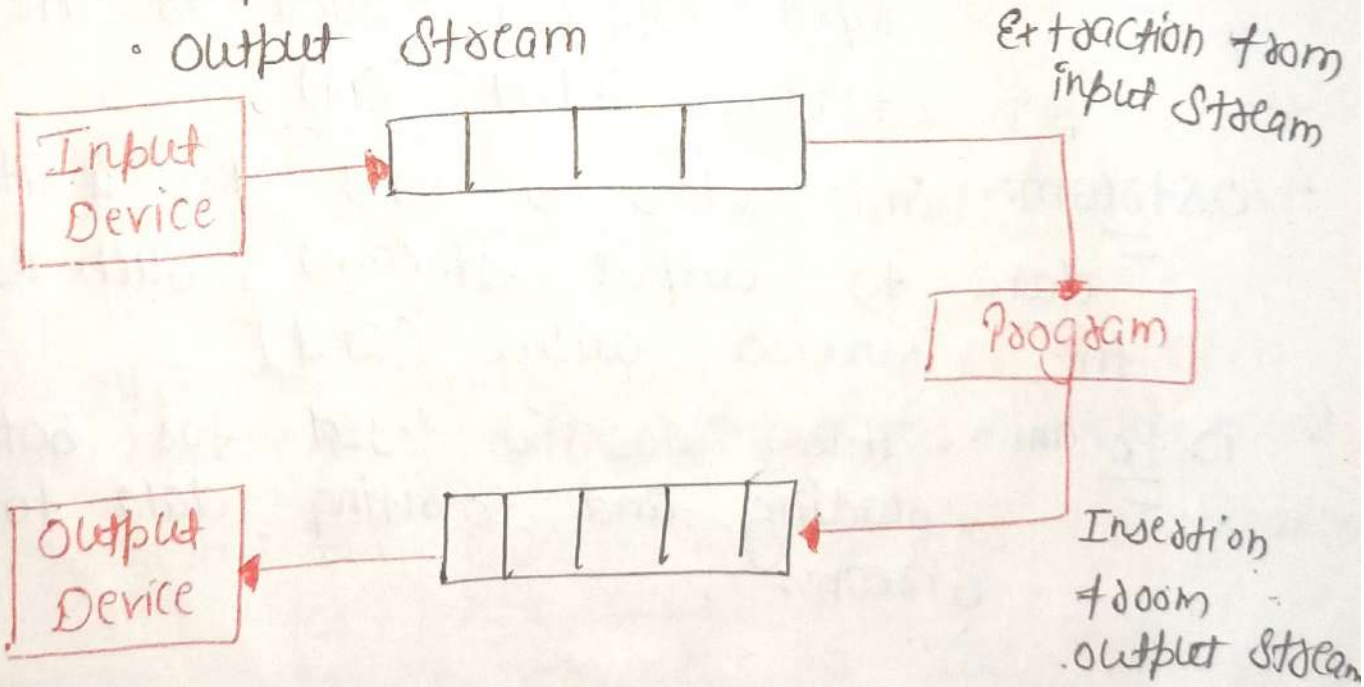
→ Seekp() → move put pointer

→ tellg() → file.tellg()
- gives the current position of input pointer.

→ Tellp() → gives the current position of put pointer.

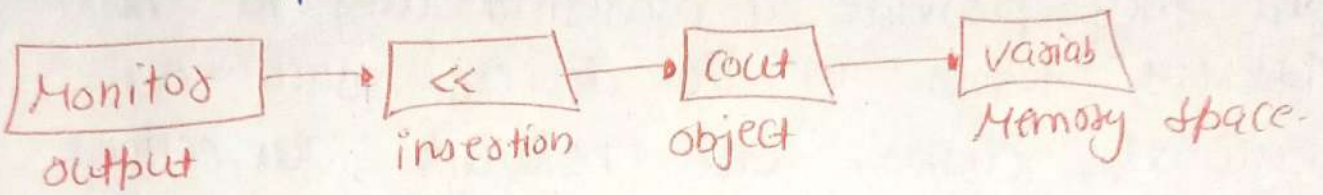
Stream

- Stream is a sequence of bytes
- a Stream is an abstraction that represents a sequence of characters that can be read or write.
- Stream are used for I/O operation in C++, and they provide a consistent way to handle different types of I/O devices, such as keywords, disks, and network connections
- header file <"iostream.h">
- There are two type of Stream in C++
  - Input Stream
  - output Stream



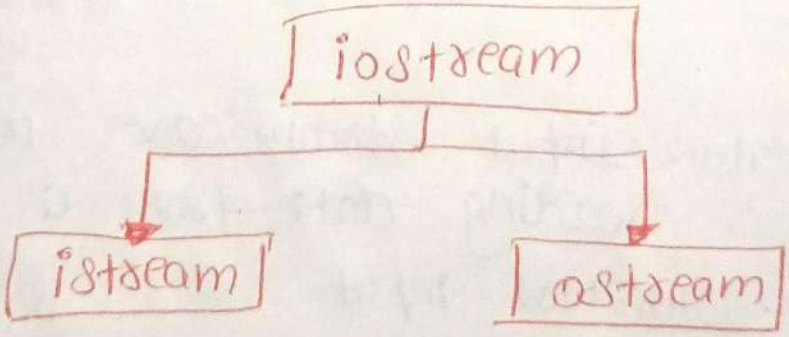
- Input Stream: Input Stream are used for reading data from a source,
  - cin ← Standard input
  - (>>) extraction operator used for reading inputs.

- Output Streams :→ output stream are used for writing data to a destination.
- Cout → Standard output [ostream]
- (<<) → insertion operator used for produce output.



Stream Class in C++

- istream :→ used for reading data from input Streams such as the standard input (Cin)
- ostream :→ This Class is used for writing data to output Streams, such as the Standard output (Cout)
- iostream :→ This Class is used for both reading and writing data to Streams.



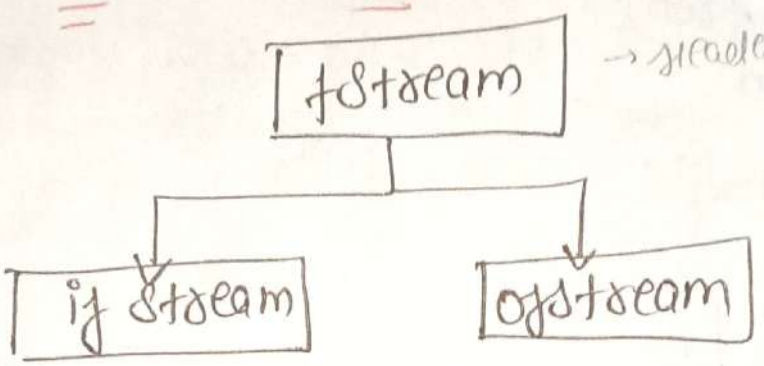
- Example:

```
# include <iostream>
# include <String>
int main()
```

Copyrighted by Codeleap.io
All rights reserved.

```
{
    std::string name;
    std::cout << " enter your name: ";
    std::cin >> name;
    std::cout << "Hello ," << name << "!" << std::endl;
    return 0;
}
```

Corresponding Method or various class available for file operation / file I/O in C++



- fstream: This call provides files I/O operation
  - It can be used to read from and write to files.
  - fstream objects can be used to open a file in either input or output mode.

→ ifstream: This class provides files I/O operation for input only.
  - It can be used to read from files.
  - ifstream object can be used to open a file in input mode.

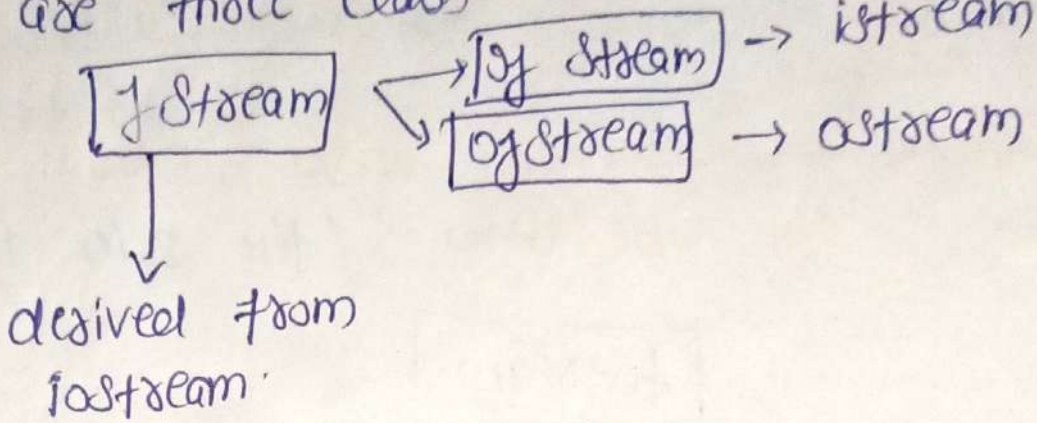
(>>)

→ ofstream: This class provides files I/O operation for output only. It can be used to write to files.
  - ofstream object can be used to open a file in output mode

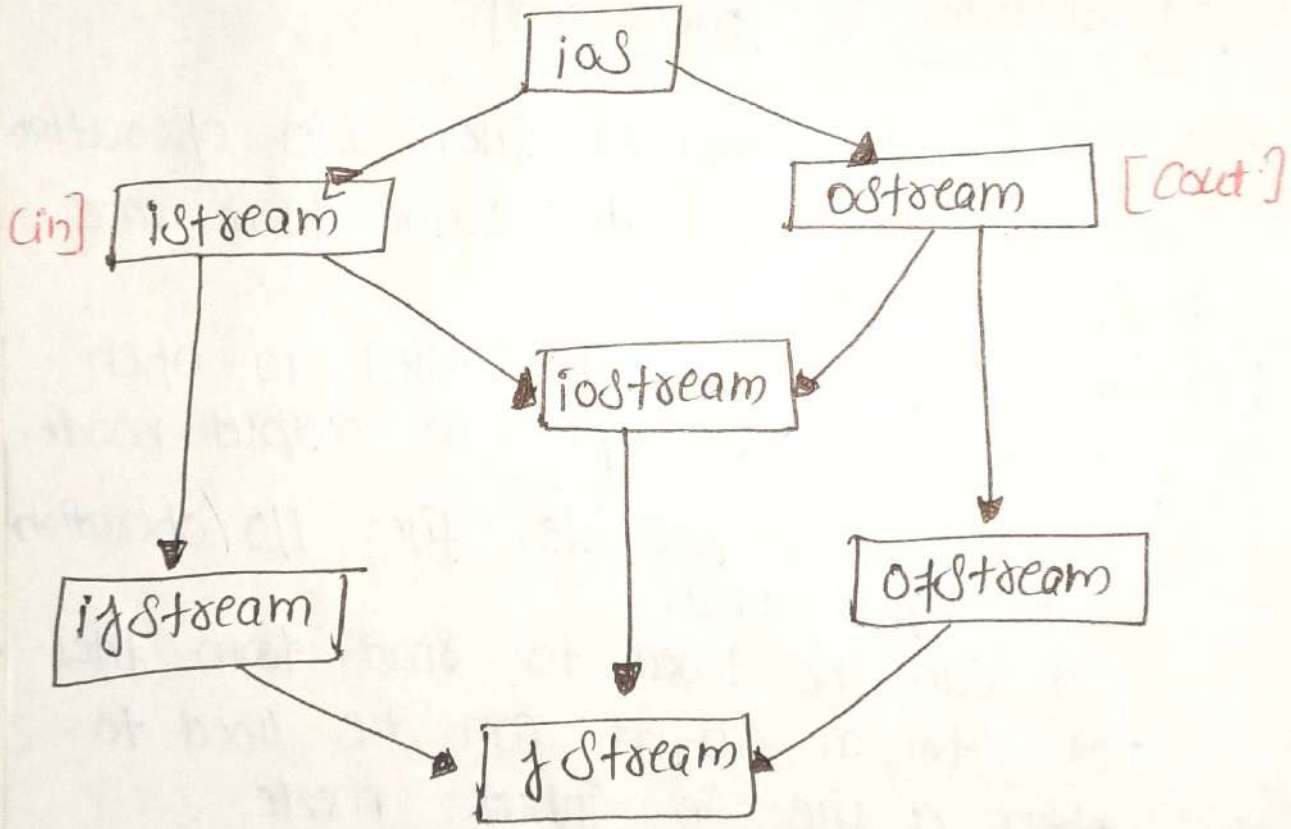
(<<)

files I/O in C++

- files I/O is the process of reading data from and writing data to files.
- There are three Class



Class hierarchy for files



Ios: This is the base Class for all I/O Stream. It provides basic function for reading and writing data.

2) istream: This Class is derived from ios and is used for input operation.
- It provide function for reading data from input streams.

3) ostream: This class is derived from ios and is used for output operation
- It provide function for writing data to output streams

4) iostream: This Class is derived from both istream and ostream and is used for both input and output operations.

5) ifstream: This class is derived from istream and is used for reading data from files.

6) ofstream: This Class is derived from ostream and is used for writing data to files

7) fstream: This call is derived from iostream and is used for both reading and writing data to files

- The ios Class provides basic function for reading and writing data. i.e istream and ostream. [Input and output Stream]

- The ifstream, ofstream, f stream Classes are used for file I/O operation

file operation [are performed using file stream]

1) opening file in C++

To open a file, you create an object of the appropriate stream class (either ifstream or ofstream) and call its open() method, which take two argument : the name of file : mode in which you want to open the file.

- The mode can be one of the following:

1) 'ios::in' : open the file for reading in input mode

2) 'ios::out' : open the file for writing in output mode

3) 'ios::app' : open the file for writing at the end of the file in append mode.

4) 'ios::ate' : open the file and seek to the end of the file

5) 'ios::binary' : open the file in binary mode

[open (filename, mode);

↓

filename.open("fileName", ios::out);

Default open Modes

ifstream	ios::in
ofstream	ios::out
fstream	ios::in / ios::out

Copyrighted by Codeleap.io
All rights reserved.

- you can also combines these mode using the bitwise OR operators (|).

eg:→ both reading and writing

i.e ios::in	ios:out mode

2) Closing a file : To close a file, you call the Close () method of the Stream Class object

- Closing a file, when we done reading from or writing to the file, you need to Close the file using 'Close()' function.

eg: fileName.Close();

3) Reading data from a file:→ To read the content of a file in C++, you can use the 'ifstream' Class, which stands for 'input file Stream".

- The ifstream Class provides several methods for reading data from a file such as 'getline ()' and '>>' operator

eg: fileName >> variable;

```
filename.open ("filename.txt", ios::in);
  -   -   -
  -   -   -
c  fine name >> x;
```

4) Writing data to file :- To write content to a file (create)

in C++, you can use the 'ofstream' Class, which stands for "output file stream".

- The 'ofstream' Class provides several methods for writing data to a file, such as "<<" operator.
- The output stream operator is used to write data of a specific type to the file (like integer, string)

  - filename << " Insert the text";

eg:-

```
# include <iostream>
# include <fstream>
int main()
{
    fstream fileName;
    filename.open("fileName.txt", ios::out);
    if ( !fileName)
    {
        cout << " error while creating file";
    }
else
    {
        cout << " file created";
        filename << "This a blog posted";
        filename.close();
    }
return 0;
}
```

[read] ifstream
[write] ostream

read
[ios::in]

+ Not use in opening a file

Exception Example [Dividing by zero]

```
# include <iostream.h>
# include <conio.h>
int main()
{
    int a,b,c;
    cout <<" enter value of a and b";
    cin >>a>>b;
    int c = a/b;
    cout <<"\n" t = "<<x;
    getch();
  return 0;
};
```

Copyrighted by Codeleap.io
All rights reserved.

```
# include <iostream.h>
void main()
{
    int a,b,c;
    clrscr();
    cout<<" enter value of a,b"<<endl;
    cin >>a>>b;
    try
    {
        if (b! =0)
        {
            c = a/b;
            cout<<" the value of c is "<<c;
        }
    else
        { throw (b);
        }
}
```

```
catch (int b)
{
    cout << " the value of b is " << b;
}
getch();
}
```

Multiple Catch Statement

```
{
    int x;
    cout << " Enter value: ";
    cin >> x;
    try
    {   if (x = = 0)
            throw (x);
        if (x = = 100):
            throw ('x');
        if (x = = 1000)
            throw (5.5);
    }
catch ( int x)
{
    cout << " value of x is zero ";
}
catch (char x)
{
    cout << " value of x is char " << x;
}
catch (float x)
{
    cout << " value of x is " << x;
}
getch();
}
```

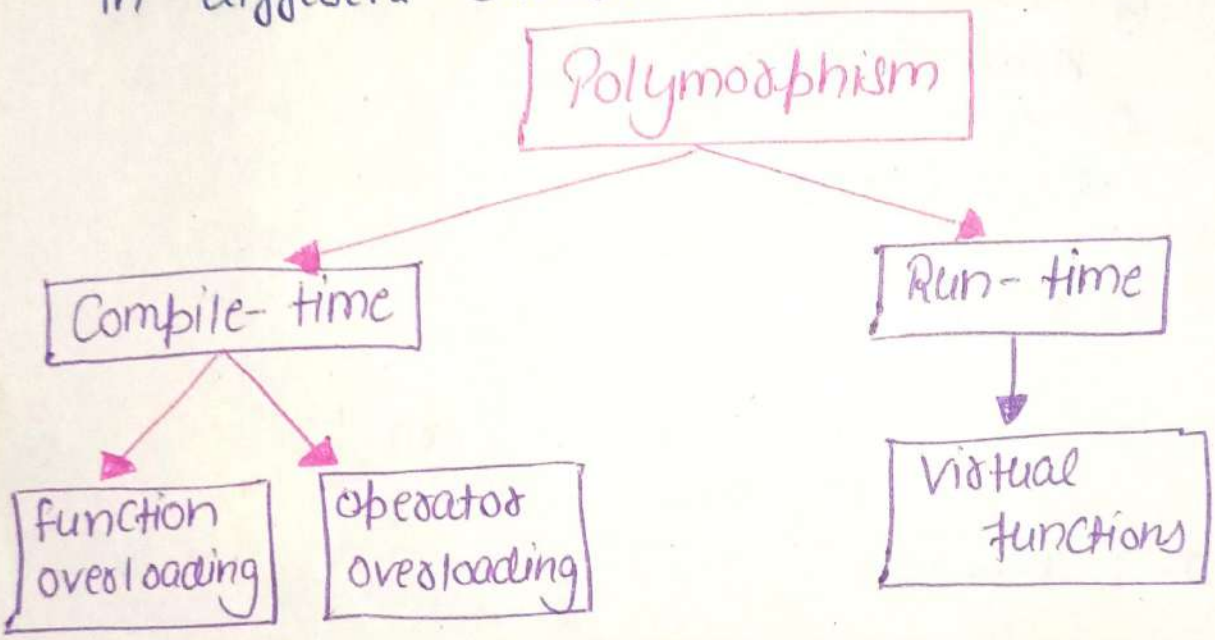
[One try block can have multiple catch statement]

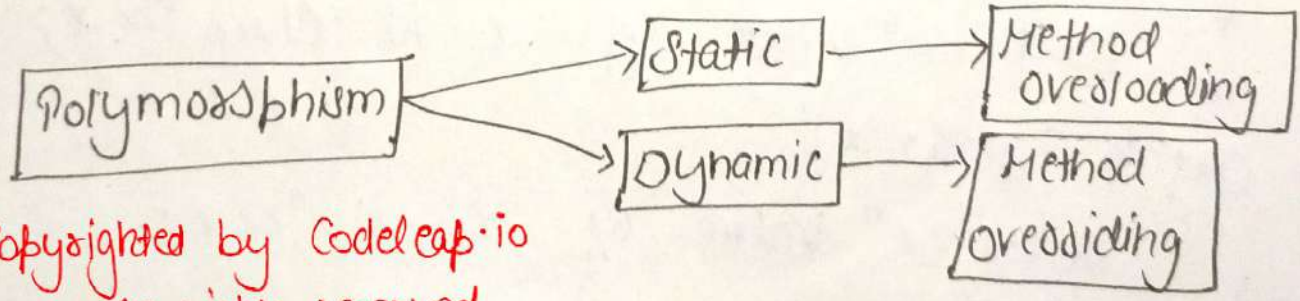
Polymorphism

- The word 'Polymorphism' means having many forms.
  (Poly = many, morph = form)
- When one task is performed by different ways are known as Polymorphism.

→ for ex: A person who at the same time can have different characteristics.
i.e A man at the same time is a father, a husband, and an employee.

- So the same person exhibits different behaviour in different situations.





Copyrighted by Codeleap.io
All rights reserved.

Advantage

- It helps the programmer to reuse the codes
- Easy to debug the codes.
- Single variable can be used to store multiple data types. [float, double etc]

Disadvantage

- developers find the it diffiault to implement polymorphism in codes
- Polymorphism reduces the readability of the program.

Compile-time polymorphism

- also known a Static polymorphism
- It is achieved by function overloading and operator overloading which is also known as Static binding or early binding
- It can be defined as a process in which a function call with an object resolve at compile time by the compiler.
  - function overloading
  - operator overloading
- function overloading: When there are multiple function with same name but different parameters then these function are said to be overloaded.
- function can be overloaded by chance in no. of arguments or/and change in type of arguments.

Syntax Class Class_name

[Type Conversion]
[function with default argument]
↑ causes

```
{
  public:
    void add()
    {
    }
    void add(int a)
    {
    }
};
```

argument
↓ fun
int add (int x, int y)
{
return x+y;
}
double add (do x, y)
{
return x+y;
}

→ Main advantage of function overload is that it allows you to write code that can work with different types of arguments as long those argument have same means.

- It increases the redability of program because you don't need to use different names for the same action. [without having to write separate function for each type]

- operator overloading
  - the operator is overloaded to provide the special meaning to the user-defined datatype.
  - advantage of operator overloading is to perform different operation on the same operand. [Uses operator keyword]

Syntax:

return-type Class-name :: operator op (arg list)
{
// body of function
}

::
Size of
member selector (.)
↑ Cannot be overload

Run-time polymorphism

- also known as Dynamic polymorphism
- It is achieved by method overriding which is also known as dynamic binding or late binding
- It can be defined as a process in which a function call with an object resolve at runtime.

- Run time polymorphism is useful because it allows for more dynamic and flexible code.
↓
allow to write code that can work with objects of differ classes.

Syntax:

```
Class A
{
 public: void show()
 {

 }
};
Class B : public A
{
 public: void show()
 {
 }
};
```

Example:

→ Run time polymorphism can be achieved by data members [without virtual keyword]

```
# include <iostream.h>
# include <conio.h>
Class Animal
 {
  public:
      void eat()
      {
       cout << " Eating ";
```

Copyrighted by Codeleap.io
All rights reserved.

```
        }
    };
Class Dog : public Animal
    {
      public:
          void eat()
              {
                cout << " Eating bread ";
              }
          };
  void main()
    {
      Dog d = Dog();
      d.eat();
  getch();
```

output: eating bread.

- function Overriding

It is used to achiere run-time polymorphism.

- Whenever we writing function in base and derived class in such a way that function name parameter must be same called. function overriding.

- It allow you to write code that can work with different type of objects

Syntax:

```
~~public~~ Class baseclass {
    access-modifier:
        // overriding function
    return-type functionname () { }
    };
```

```
public class derived Class : public baseClass {
    access_modifier:
        //overriding function
    return_type functionname() { }
};
```

It redefines the function of base Class in the derived Class, so the return-type, function-name, parameter should be same to achieve function overriding

Example:-

```
#include <iostream.h>
class baseclass
{
 public:
  virtual void print()
    {
      cout << "The Base function" << endl;
    }
};
class derived Class : public baseClass
  {
   public: // function overriding
      void print()
    {
      cout << "The Derived function" << endl;
    }
};
void main()
{
  Derived d1;
  d1.print();
getch();
}
```

d1 → only call derived Class

d1.print(); → If we use Class B obj से Class A की function चलाना है तो → Scope resolution operator → pointer through

```
B obj;
obj.A::print();
```

If we use virtual

```
A *ptr; B obj;
ptr = A obj;
ptr -> print();
return
```

Output:

Derived function

- Here, the same function print is in both Base and Derived Classes.
- So, when we call print from the Derived object d1, the print from Derived is executed by overriding the function in Base.

The function in derived class overrides the function in base Class

Advantages

- Helps in saving memory space.
- Maintain consistency and readability of our code
- Help in making code re-usability easy.

Disadvantages

- The static Method cannot be overridden
- function overriding cannot be done within a same class. So, we need to do implement inheritance
- Additionally the function signature must be the same in both the base and derived Class
- If the signature of the function in the derived class is different from the signature in the base class, it is not considered to be an override.

Topic:- take report case study

Digital forensic case study of take it investigation

function

- A functions is a block of code that perform a specific task which only runs when it is called.
- you can pass data, known as parameters, into a function.
- functions are used to perform certain actions, and they are important for reusing code

Syntax:

```
return-type function_name (parameter list)
{
    body of the function
}
```

- **Virtual function** → virtual function is called during run time

- A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class.
- It is declared by using the virtual keyword
- They are mainly used to achieve Runtime polymorphism.
  [Runtime polymorphism can be achieved only through a pointer of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class)
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.

Rules for Virtual function

- Virtual functions cannot be Static
- Virtual functions can be a friend function of another Class
- Virtual function should be accessed using a pointer
- A Class may have a virtual destructor but it cannot have a virtual constructor

limitations

1) Slower
2) Difficult to Debug

Copyrighted by Codeleap.io
All rights reserved.

Syntax:

```
virtual return-type function-name (arguments)
{
    // body of function
}
```

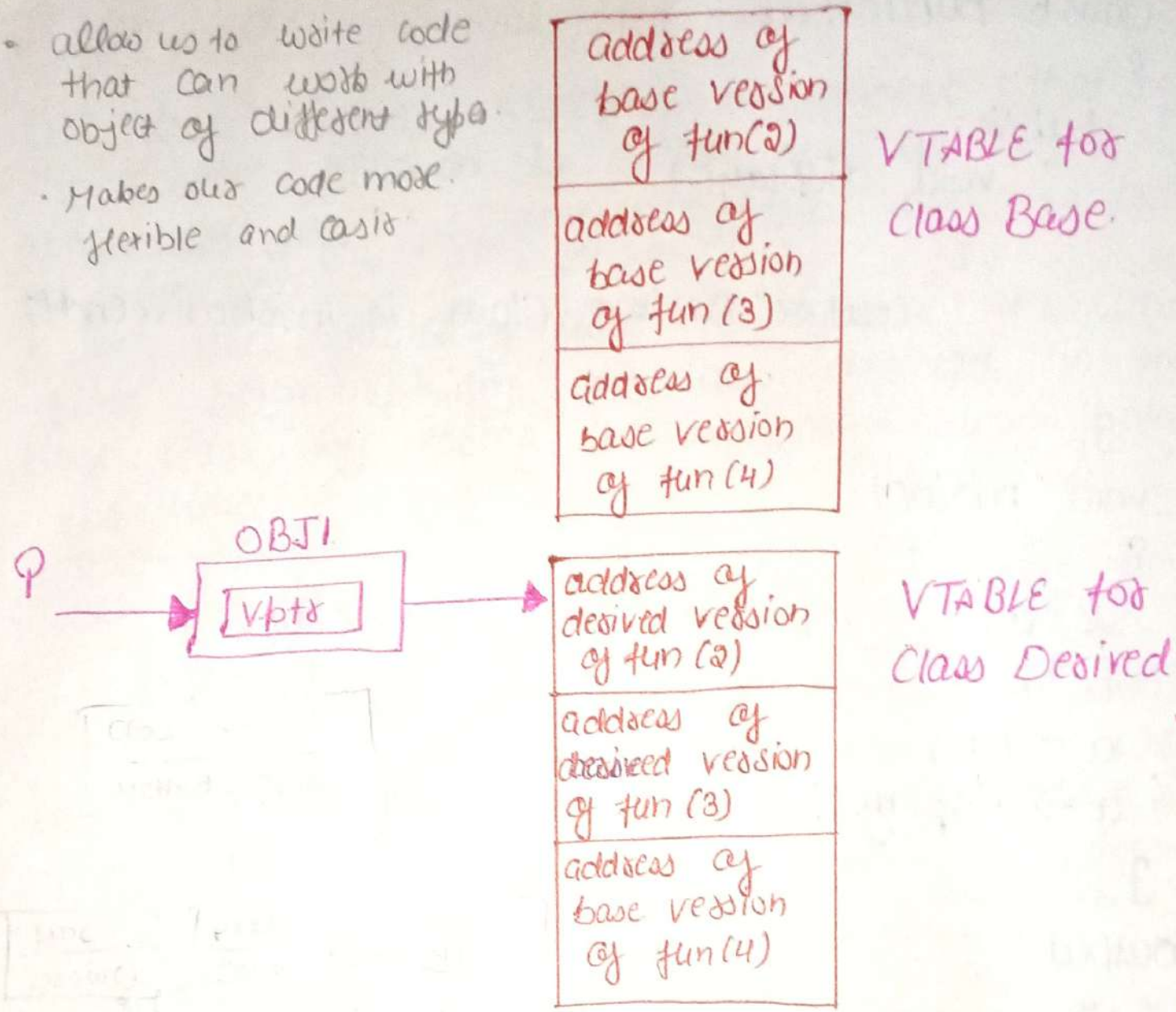
Virtual Table

A Class contains a static array of member function pointer known as vtable. which store the address of each virtual member function present in that class.

Virtual pointer

when an object of a Class is created, a virtual pointer is inserted to point to the virtual table of that Class.

- allow us to write code that can work with object of different type.
- Makes our code more flexible and easier



Example:

[first without use of virtual keyword]

```
#include <iostream.h>
# include <conio.h>
Class A
{   int x=4;
  public:
  virtual void display()   // virtual function
  {
     cout << "Base Class is invoked" << endl;
  }
};
```

```
Class B: public A
{
  public:
      void display()
      {
          cout << " Derived Class is invoked" << endl;
      }
};
void 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

- "a" is the base Class pointer. The pointer can only access the base Class members but not the member of the derived Class.
- C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the member of derived Class.
- Therefore, there is a need for virtual function which allow the base pointer to access the member of derived Class

Copyrighted by Codeleap.io
All rights reserved.

Why use virtual functions

- We use virtual function to ensure that the correct function is called for an object, regardless of the reference type used to call the function.
- They are basically used to achieve the run-time polymorphism and are declared in the base class by using the virtual keyword before the function.

Pure Virtual function → We can't change the signature of pure virtual function

- A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation, we only declare it.
- A pure virtual function is declared by assigning 0 in the declaration. [It means declared in Base class but no implementation.
- These functions can't be global or static

Syntax virtual return_type function_name (argument) = 0;

→ virtual void display () = 0;

- It can be considered as an empty function means that the pure virtual function does not have any definition relative to the base class.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class
- They start with virtual keyword and ends with equal to zero

Virtual function	Pure Virtual function
1) A virtual function is a member function in a base class that can be defined in a derived class. • Virtual function can have body.	1) A pure virtual function is a member function in a base class whose declaration is provided in a base class and implemented in a derived class. [Not have body]
2) Not abstract classes.	2) are abstract classes.
3) function is provided in the base class.	3) function is not provided in the base class

Abstract Class

any class that contains a pure virtual function is consider an abstract class.

- A class which have at least one pure virtual functions. A pure virtual fun specified by "=0".
- An abstract class cannot be instantiated. It means we cannot create object of the class
- We can create pointers of abstract class type
- Abstract class mainly used for upcasting but can only be used as a base class for other classes. Main purpose to provide an appropriate base class from which other classes can inherit

Syntax :

```
Class A
{
    public:
        virtual void Show() = 0;
};
```

- Abstract class can have normal func & variable along with a pure virtual function.

Example:

```
#include<iostream.h>
# include < conio.h>
 Class A        //Abstract Class.
 {
     public:
     virtual void show () = 0; //Pure virtual function
     void display ()  // Normal Member function
      {
          Cout << " Hi  i am base Class";
      }
};
Class B : public A
 {
     public:
      void show ()
      {
          Cout << " Hi  i am derive Class";
      }
};
void main ()
 {
    Class a ();
    A *p; B ob;
    p = & ob;
    p -> show ();
    getch ();
}
```

- By declaring a pure virtual function in the abstract base class, we force each derived class to provide its own implementation of that function.

output:

Hi i am derive Class

Copyrighted by Codeleap.io
All rights reserved.

int x1 = 6;

- 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 class containing the pure virtual function cannot be used to declare the object of its own, such classes are known as abstract base classes

Example: Pure virtual function

```
#include <iostream.h>
Class A
{
Public: virtual void disp()=0;
};
Class B : Public A
{
public:
    void disp()
    {
      cout<<"\n I am from Derived Class B";
    }
};
Class C: Public A
{ Public:
    void disp()
    {
      cout<<"\n I am from Derived Class C";
    }
};
int main() {
  B obj b;
  C obj c;
  obj b.disp(); , obj c.disp(); }
```

Derived Class is derived from the base Class

```
int main()
{ Base *bptr; //Base b,
  Derid d;
  bptr = &d;
  bptr -> disp();
```

to create base pointer used for runtime achieving

return

[the base Class contains the pure virtual function. the base Class is an abstract base Class we cannot create the object of base Class

Virtual Destructors

- Destructors is a member function of a class used to free the space occupied by or delete an object of the class that goes out of scope.
- A destructor has the same name as the name of the constructor function in a class, but the destructor uses a tilde (~) sign before its function name.
- It automatically called.
- Destructor destroy the class objects created by constructor.

- A destructor is a special member function in C++, that gets called automatically when an object is destroyed.

→ virtual destructor is special type of destructor that is declared with the 'virtual' keyword in C++ class.

Virtual Destructor

- Destructor can be declared as virtual, but constructor can't be declared as virtual.
- Virtual Destructor are implemented just like normal virtual function.
- Virtual Destructor in C++ is used to free up the memory allocat resources used by the derived class object when it is deleted from the memory.
- A base or parent class destructor use the virtual keyword that ensures both base and derived class destructor will be called run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Copyrighted by Codeleap.io
All rights reserved.

Example

```
# include <iostream.h>
Class Base
{
  public:
      Base()  // Constructor member function
      {
        Cout <<"Base Class Constructor"<< endl;
      }
    virtual ~Base() // To call destructor Derived function
    {
      Cout <<" Base Class destructor "<<endl;
    }
};  // inheritance Concept
Class derived: public Base
  {
    public:
        Derived()  // Constructor function
    {
      Cout <<" Derived Class Constructor "<<endl;
    }
    ~ Derived ()  // Destructor function
  {
    Cout <<" Derived Class destructor "<<endl;
  }
};
int main()
{
  Base * bptr ;
  bptr = new derived; // Create a base Class ptr object
```

```
    delete bptr;
return 0;
}
```

that both destructor are called when the object is deleted through a pointer to the base class.

Output: Base Class ~~c~~constructor
Derived Class constructor
Derived Class ~~co~~destructor
Base Class destructor.

→ destructor of the derived class is called first, followed by the destructor of base class.

- If the base class destructor does not use a virtual keyword, only the base class destructors will be called or deleted its occupies space because the ptr obj is pointing to base class. So it does not call the derived class destructor to free the memory used by the derived class, which leads to memory leak for the derived class.

→ Constructor base class.
Constructor Derived class
Destructor base class.

- We have used a virtual destructor inside the base class that calls the derived class destructor before calling the derived class destructor ~~be~~ and release the spaces or resolve the memory leak issues in the program.

Copyrighted by Codeleap.io
All rights reserved.

Compile Time

- It can be defined as a process in which a function call with an object resolve at compile time by the compiler.
- also known as overloading, early binding and static binding.
- It is achieved by function overloading and operator overloading.
- less flexible
- It provide fast execution at compile time.

void area(int a);
void area(int a, b);

Run - Time Polymorphism

- It can be defined as a process in which a function call with an object resolve at runtime.
- also known as overriding, Dynamic binding and late binding.
- Pointer and virtual functions
- More flexible
- It prove slow execution at run-time

function Overloading

- It falls under compile-Time polymorphism
- A function can be overloaded multiple time as it is resolved at compile time.
  - Destructor can't be overloaded
- They are in the same scope
  - does not use inheritance
- function signatures should be different
- No keyword is used during function overloading

function Overriding

- It falls under Runtime Polymorphism.
- A function cannot be overriden multiple time as it is resolved at run time.
  - can be overriden
- They are of different scopes.
  - achieved with help of inheritance
- function signatures should be same.
- When the function is defined, it is decled by 'virtual' keyword

Virtual Derivation

Virtual Derivation is a Technique used in C++ to resolve the ambiguity that when a derived class inherits from multiple base classes that share a common base class.

- It is specifically used to avoid problems related to multiple inheritance and "diamond Problem".
- The diamond problem occurs when a Class inherits from two or more classes that have a common base class. In cases compiler confuse and it display error
- To solve this problem, C++ introduces the concept of virtual inheritance. When a base class is virtually inherited, it ensures that only one instance of the common base class is shared among all the derived classes. This way, the diamond problem is eliminated.

= Syntax:

```
Class B : Virtual public A
{
    3;
Class C : public Virtual A
{
    3;
```

- 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.
- avoding duplicate copies and the ambiguity caused by the diamond problem.

Type Conversion

- Converting data from one type into another type is called type conversion.
- Type conversion is the process that converts the predefined data type of one variable into an appropriate data type.

eg: we are adding two number, where one variable is of int type and another of float type.

x and y
↓ ↓
int float

- we need to convert the int variable into a float to make them both float data type to add them

[x = y]

int x = 20; float y = 5;

float z = x + y;

Type Conversion

- Implicit
- Explicit

Copyrighted by Codeleap.io
All rights reserved.

1) Implicit Type Conversion: also known as 'automatic type conversion'.

- The implicit type conversion is the type of conversion done automatically by the compiler without any human effort.
- It means an implicit conversion automatic convert one data type into another type based on some predefined rules of the C++ compiler. Hence it is also known as automatic type conversion

eg:

```
int x = 20;   short int y = 5;
int z = x+y;
```

- C++ compiler automatically converts the lower rank data type (short int) value into higher type (int).
- It avoids the data loss, overflow, or sign loss in implicit type conversion of C++.

→ Order of the typecast in implicit conversion [lower rank to higher rank]

- bool → char → short int → int → unsigned int → long → unsigned → long long → float → double → long double

- C++ provides mechanism to perform automatic type conversion if all variable are of basic type.

Program

Copyrighted by Codeleap.io
All rights reserved.

```
#include<iostream.h>
# include<conio.h>
void main()
{
int num1 = 25;      // assign integer value
float num2;         // declare a float variable
num2 = num1;        // Convert int into float variable.
cout<<" The value of num1 is:"<<num1<<endl;
cout<<" The value of num2 is:"<<num2<<endl;
return 0;
};
```

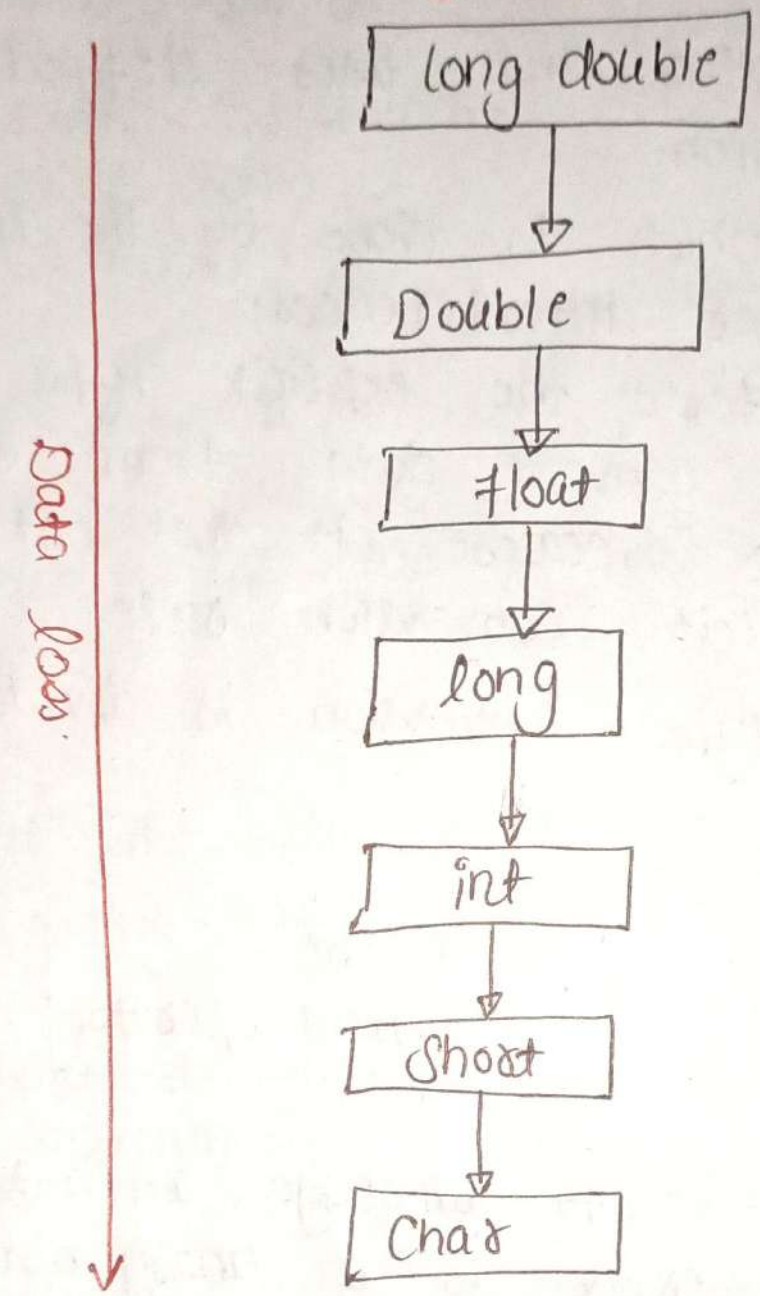
output: The value of num1 is : 25
" " " num2 is : 25

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned) and overflow can occur (when long long is implicity converted to float)

Data loss During Conversion (Narrowing Conversion)

- Conversion from one data type to another is prone to data loss.
- This happens when data of a larger data type is converted to data of a smaller data type

Higher Data Type



Lower Data type.

- Possible Data loss During Type Conversion

Advantages of Type Conversion

- Solve mathmatical and logical expressions easily without any data loss.
- prevent the error.

Copyrighted by Codeleap.io
All rights reserved.

2) Explicit Type Conversion: This process is also called type casting and user-defined type conversion.

This conversion, which is done by the user or requires user interferences.

- Generally, we force the explicit type conversion to convert data from one type to another because it does not follow the implicit conversion rule.

∴ The explicit type conversion is divided into 2 type.

- Using the Cast operator
- Using the assignment operator

- Cast operator :- In C++ language, a Cast operator is a unary operator who forcefully converts one type into another type.

which is a special operator that tells the compiler to convert the value to a spec. dT.

- C++ support four types of Casting:

1) Static_Cast [static_cast <data type> (expres);
2) Dynamic_Cast ["]
3) Const_Cast ["]
4) Reinterpret_Cast ["]

Type Conversion operator

Example: Convert float data type into int type using the cast operator.

```
# include <iostream.h>
# include <conio.h>
void main()
{
 float f2 = 6.7;
 int x = static_cast<int>(f2);
 cout << " The value of x is : " << x;
 getch();
};
```

Use cast operator.

output: The value of x is : 6

- Converting by assignment: This is done by explicity defining the required type in front of the expression in parenthesis.
- This can be also considered as forceful casting.
- Syntax: (type) expression;
  - → where type indicates the data type to which the final result is converted.
  - → C-Style type Casting: Cast notation.
    - (datatype) expression;
  - → function Style Casting :→ old c++ Style type Casting.
    - datatype (expression);

Copyrighted by Codeleap.io
All rights reserved.

Program [one data type into another]

```
# include <iostream.h>
# include <conio.h>
void main()
{
  float num2;      // declare a float variable.
  int num1 = 25;   // initialize an int variable.
num2 = (float) num1;  // Convert data from int
                        to float
cout <<" The value of int num1 is:" << num1<< endl;
cout << " The value of float num2 is:" << num2 << endl;
getch();
}
```

output: The value of int num1 is: 25
The value of float num2 is: 25.0

User - defined data type

- for user defined data type programmers have to convert it by using constructor or by using casting operator
- A user - defined data type are designed by the user to suit their requirements, the compiler does not support automatic type conversions for such data type.
- the user needs to design the conversion routines by themselves if required.

Copyrighted by Codeleap.io
All rights reserved.

· Three type of Situation arise in user-defined data type Conversion.

1) Basic type to Class type (Constructor)
2) Class type to Basic type (Casting operator)
3) Class type to Class type (Constructor & Casting operator)

1) Basic type to Basic type → Conversion a one basic type to another basic data type

Implicit [Automatic]
Explicit [By programmer]

2) Basic type to Class type

· In C++ conversion of basic type to Class type can be done using constructor.
· By creating a Constructor the basic data type integer, float can be converted in to object of Class type.
· Conversion is automatically done by compiler, by applying type casting.

Ex:-
```
#include <iostream.h>
# include <conio.h>
Class time
{
  int hours;
  int minutes;
public:
        time (int t)
        {
            hours = t/60;
            minutes = t % 60;
        }
```

· basic types are built in data type such as int, float, char etc
· Class type on other hand, are user defined data type that are creating using Class
· A Constructor is a special member function that is called when an object of a Class is created. It is used to initialize the object data members.

```
    void display()
    {
      cout << " hours (s) = " << hours << endl;
      cout << " minutes = " << minutes << endl;
    }
};
void main()
{
  Class c();
  int duration = 85;
  time t = duration;
  t.display();
  getch();
}
```

Copyrighted by Codeleap.io
All rights reserved.

Output:
hours (s) = 1
minutes = 25

2) Class to basic type Conversion

- The compiler does not have any knowledge about user-defined data type built using classes. [converting an object of a class type to a value of basic type]
- In this type of conversion, the programmer, need to explicity tell the compiler how to perform conversion from class to basic type. These instructions are written in member function. Such type of conversion is also known as overloading of type cast operators. which is also known as conversion function.
- With the help of this conversion function a class type can be converted to basic types like integer, char, float.
- The Class type to Basic type Conversion is done using casting operator function.

→ The compiler does not support automatic type conversion for the user-defined data type.

- We can use casting operator function to achieve this. The casting operator function should satisfy the following condn:
  1) It must be a class member.
  2) It must not specify a return type
  3) It must not have any arguments

Syntax:

```
operator destination type ()
{
    ....
    return
}
```

[A conversion operator is a special member function of a class that allows you to convert an object of that class to a basic type

Example:

```
#include <iostream.h>
class hours        class time
{
    int hrs;
    float f;       int min;
public:
    hours (int t) (int m, int h)
    {
        hrs = t/60;      →   void display()
        hrs = h;             {
        min = m;             cout << " the hours are " << hrs << ...
    }                        cout << " the min are " << min << endl;
                             }
    operator float () // type conversion from
                         class to basic.
    {
        return float (hrs)/2;
                         // type conversion int to float
    }                    [basic to basic]
```

```
void show()
{
  cout << hrs << "hours" << endl;
}
};
void main()
{
  float f;
  hours t1 = 185;
  t1.show();
  f = t1;
  cout << "time =" << f << endl;
  return 0;
}
```

```
operator int()
{ return hrs * 60 + minutes;
}
```

```
int duration;
time t(3,20);
t.display();
duration = t;
```

output

3 hours
time = 1.5

// type conversion from class ob 't1' to float 'f'.

```
cout << "duration" << duration << endl;
```

[n = 3
m = 20
du = 200]

3) One Class type to Other Class type

- When an object of one class is assigned to object of another class, it is necessary to instruct the compilier about how to make conversion b/w these two - user defined data types.
- There are two ways to convert object data type from one class to another class.
  1) Define a conversion operator function in source class. [Using constructor]
  2) Define one argument constructor in a destination class [Using conversion function].
- Consider the statement : destinat Class obj t = obj Sourc

  obj X = obj Y;
- The class Y type data is converted to the class X type data and the

Converted value is assigned to the obj x.

- The Class Y is known as source class and Class X is known as destination Class.

Example: Using Constructor

```
# include <iostream.h>
Class minutes   // source Class
{
  int m;
 public:
        minutes (int ms)
        {
           m = ms;
        }
  void show()
     {
       cout << "Minutes =" << m << endl;
     }
  int get data()
   {
     return m;
   }
};
Class hours {   // destination Class.
   int h;
  public:
  hours ()  {
   h=0;
   }
 void show () {
        cout << "Hours =" << endl;
        }
```

Copyrighted by Codeleap.io
All rights reserved.

```
}
hours (minutes mm)
    { h = mm.getdata() / 60;
    }
};
```

The conversion take place in the source class and the result is given to the destination class object.

Conversion	Source Class	Destination
Basic to Class	Not applicable	Constructor
Class to Basic	Casting operator	Not applicable
Class to Class	Casting operator	Constructor

Class in C++

- It is the building block that leads to object - oriented programming. It is a user defined data type, which holds its own data members and member function, which can be accessed and used by creating an instance of that class.
- A C++ class is like a blueprint for an object.
- In other words, a class is a collection of object of the same kind.
- for ex: facebook, Instagram, Twitter and Snap all come under social media class.

- Declaration of Class in C++

→ you can define classes using the keyword 'Class' followed by the name

```
Syntax: Class Class-name   ← User-defined type
        (keyword)
        {
            access-specifier: // can be private, public
            data-variable;    // variable to be used
            member-function   // Methods to access data member
            {
                body
            }
        };   // Class name ends with a semicolon.
```

- Here, inside the class, there are access-modifier, data variables and member function

- Access Modifiers

These are the specifiers which provide or grant access for the members. They are of three types:

Private: only members of the same class have access to private members

Public: you can access the public members from within, as well as from outside of the class.

Protected: you can access the protected member from the same class member and member of the derived class. It is also accessible from outside the class but with the help of friend function.

Object in C++

- An object is an instance of a Class.
- When a class is defined, no memory is allocated but when it is instantiated (i.e an object is created) memory is allocated to hold the object data members.

Syntax: Classname Objectname;

- you can create multiple objects from the same Class.
- each object is independent and maintains its own state.
- accessing data members and member functions
- The data members and member function of the Class can be accessed using the dot ('.') operator with the object (->)

eg:→ If the name of the object is obj and you want to access the member function with the name printvame() then you will give to write obj.printvame().

→ Accessing Data members

The public data members are also proccesed in the smae way. however the private data members are not allowed to be accessed directly by the object.

- Accessing a data member depends only on the access control of that data member.

- This access control is given by Access modifiers in C++ :
  - → public access from outside the class
  - private only accessed within the Class
  - protected. within the Class and its derived Class

→ Member function in Classes

There are 2 ways to define a member function:

- Inside Class definition.
- outside Class definition

- To define a member function outside the Class definition we have to use the Scope resolution :: operator along with the Class name and function name.

Example:

```
#include <iostream.h>
Class geeks
{
    //Access Specifier
    public:
        String geekname; //Data member
        void printname //Member function.
        {
          cout << "Geekname :" << geekname <<
        }
};int main()
{
    Geek obj 1;
    ob1.geekname = "Tannu"; // accessing DM.
    obj1.printname(); // accessing MF
getch();
```

- Member function is a function that is associated with class and can access the data members and other member functi

Copyrighted by Codeleap.io
All rights reserved.

Rules of Derivation

- The rules of derivation in C++ refer to how a derived class can inherit members from a base class.
- When a class is derived from another class in C++, it can inherit all the member of the base class except for it constructors, destructors and overloaded operators.
- The derived class can access the public and protected members of the base class, but not the private members because private members are only accsible within the scope of the base class.

This is known as rule of derivation in C++.

- Public Derivation : When the access specifier for the inherited base class is public, all public member of the base class become public member of the derived class.
- All protected members of the base class become protected members of the derived class.
- All private members of the base class remain private to it and are inaccessible by the derived class.

Accessibility	Private	Protected
Base Class	Yes	Yes
derived Class	No	Yes

Same in Protected

Example:

```
# include <iostream.h>
Class Base
{
private:
```

```
    int pvt = 1;
protected:
        int prot = 2;
public:
        int pub = 3;
    int getPVT ()   { return pvt; }
};
Class PublicDerived: public Base {
  public:
        int getPROT () { return prot; }
};  int main()
  {   PublicDerived object1;
```

cout << private="<<obj1.getPVT<<

- **Private Derivation:** If the access specifier is private, all public members of base Class becomes private member of the derived Class.
- All protected member of the base Class also become private member of the derived class.
- All the private member of the base Class are remain it and are inaccessible by the Derived Class.

Example:

```
#include <iostream.h>
Class Base
{
  private:
        int pvt = 1;

  protected:
        int prot = 2;

  public:
        int pub = 3;
```

Copyrighted by Codeleap.io
All rights reserved.

```
// function to access private member
int getPVT ()
{
    return pvt ;
}
};
Protected                    Protected
Class Private Derived : private Base.
public                       public
{
public:
    int getProt () // function to access
                      ~~private~~ protected. from
                                  Base.
    {
        return prot;
    }
};  in case of public
    int getPub () // function to access
                      private member
    {
        return pub;
    }
};
int main ()
{ public
Private Derived object1;
cout << "Private cannot be accessed" << endl;   object1.getPVT() << endl; [Public]
cout << " Protected = " << object1.getProt () << endl;
cout << "Public =" << object1. getPub () << endl;
                                  → Public → pub
return 0;
}
```

Output: Private cannot be accessed.
Protected = 2.
Public = 3

3) Protected Derivation

A base class can also be inherited with the access specifier protected by a derived class then the public and protected member of the base class become protected member of the derived class.

- All private members of the base class remains private to it and are inaccessible by the derived class.

Example: Same as private Derivation.

Modular programming is an approach to programming that involves breaking a program down into smaller, more manageable parts called "Modules".

- In C++ Modular programming can be achieved using function and classes.
- functions are self-contained block of code that perform a specific task and can be called from other parts of the program.
- Classes are similar to function, but are more complex and can have multiple function and data members.

Access	Public Members	Protected Mec	Private.
Base Class	yes	yes	yes
Derived Class	yes	yes	No

Copyrighted by Codeleap.io
All rights reserved.

```
Class A
{
  public:
      {
        };
Class B : public A
  {
    public:
        {
          }
Class C : public A
  {
    public:
        {
          }
Class D: public B, public C
  {
    public:
        {
          };
```

- Class B and C both inherit from Class A.
- When Class D inherits from classes B and C, it has two copies of Class A.
- Lead of ambiguity.

```
Class A
  {
    public:
      int a;
  };
Class B: virtual public A
  {
    public:
      int b;
  };
Class C: virtual public A
  {
    public:
      int c;
  };
Class D: public B, public C
  {
    public:
      int d;
  };
```

→ Class B and C both inherit from Class A virtually

→ When Class D inherit from classes B and C, it has one instance of Class A.

Unit-1
Topic

Copyrighted by Codeleap.io
All rights reserved.

Conversion function

- Conversion function is a member function of a Class that is used to convert an object of that Class to another data type.
- A conversion function is defined using the 'operator' keyword.

eg:-> Class to basic type

Conversion b/w objects of different Classes

- Conversion b/w objects of different Classes is achieved using conversion constructors or conversion operators or function

→ Conversion Constructor is a special constructor that take a single argument of a different type and is used to convert that type to an object of Class.

Syntax: Class-name (type argument);

→ Conversion operator is a member function of a Class that is used to convert an object of that Class to a different type.

Syntax: 'operator type().

- To convert a basic type to an object of Class, we use Conversion constructor
- To convert a object of a Class to a basic type, we can use Conversion operator

Class to Class conversion using construct in destination Class

- A Source class can be converted to a destination class by using a constructor in destination class which take reference to source class.

→ Destination Class (Source-Class &object)

```
{
    //assignment of members of
        destination class
    // with members of source class
    using source class object
}
```

Example: # include <iostream.h>

```
Class Class_type_one //Source Class
{
    String a = " Tannu";
public:  Member function which  return String type
            String get_String()  data data
            {
                return (a);
            }
        void display()
        {
            Cout << a << endl;
        }
};
//Destination Class
```

Copyrighted by Codeleap.io
All rights reserved.

```
Class Class-type-two
    {
        String b;
public:
        Class-type-two () // Default Constructor
        {

        }
        Class-type-two (Class-type-one&a)
        {
            b = a.get-String();
        }
    void display ()
    {
        Cout << b << end l;
    }
};
void main()
{
    Class-type-one a; //creating object of Class type one
    Class-type-two b; // creating object of two
    b = a;  // Class type Conversion using operator overloading
    a.display ();
    b.display ();
    return 0;
}
```

output: Tanny
Tanny

Copyrighted by Codeleap.io
All rights reserved.

Class to Class Conversion using Casting operator

- Class to Class Conversion is also possible using Casting operator in Source Class
- operator type will be that of destination Class.
- A Casting operator will be member function of source Class and will return object of destination Class.

```
# operator destination_class ()
  {
      destination_Class obj;
      // assign of members of destin Cls with
      return obj          members of source Cls
  }
# include <iostream.h>
Class Class_type_two; // destination Class.
Class Class_type_one  // Source Class
  {
      String a = "tenna";
public:
      String get_String ()
      {
        return (a);
      }
      void display()
      {
```

Copyrighted by Codeleap.io
All rights reserved.

```
        cout <<a << endl;
    }
    // declaration of casting
    operator class-type-two ();
};
    class class-type-two
    {
        public:
                string b;
                void display ()
                {
                    cout << b << endl;
                }
};
    // define our casting operator
class-type-one::operator class-type-two()
 {
    class-type-two obj;
    obj.b = a;
    return obj;
 }
void main()
{
    class-type-one a;
    class-type-two b;
    b = a;
    a.display();
    b.display();
 getch();
 }
```

Copyrighted by Codeleap.io
All rights reserved.