Date 15/09/2020

C++

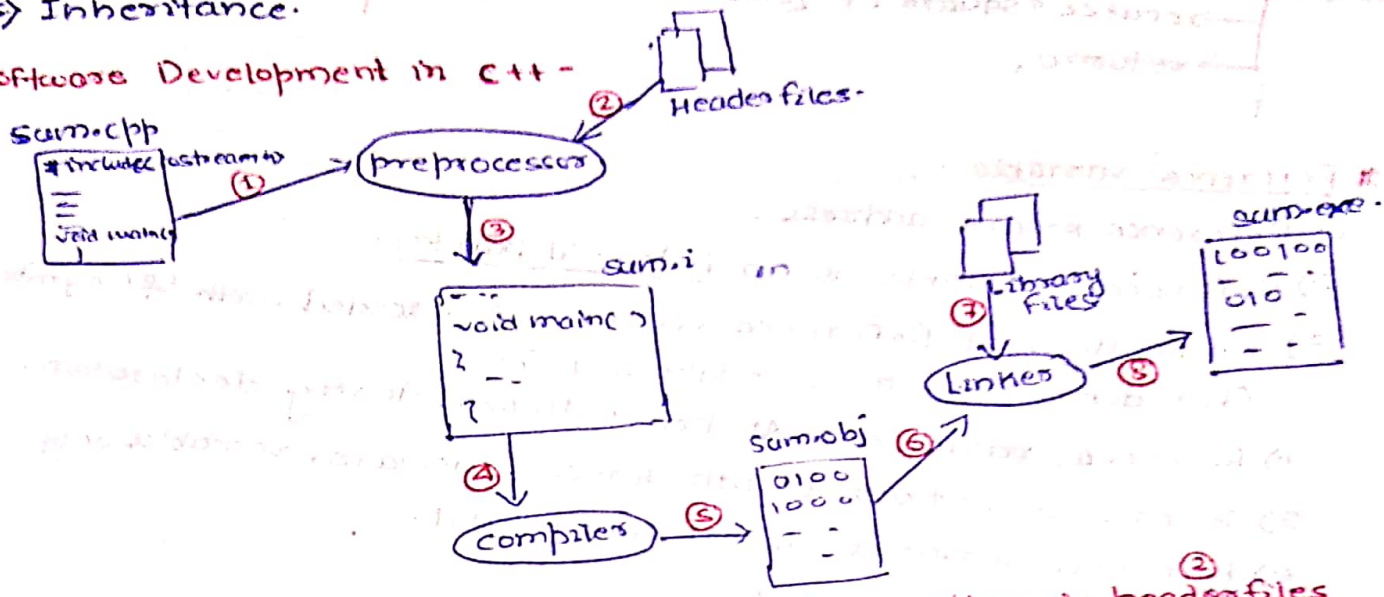
- Saurabh Sukla Sir.

* OOPs pillar -
1) Encapsulation.
2) Data hiding.
3) Abstraction.
4) Polymorphism.
5) Inheritance.

- develop by (Bjarne Stroustrup) 1979 (C with classes.)
- 1993 (C++)
- middle level language (low + high)
- implement → procedural
  → object oriented
  → generic programming using templates.

• class - Blueprint of object.
• object - runtime entity.

* Software Development in C++ -



① #include< > are replaced by the code written in header files (i.e. declaration statements) using preprocessor. ③

③ After preprocessing 'sum.cpp' converted into sum.i, which is a temporary file stored in RAM.

④ After that compiler ⑤ convert 'sum.i' into 'sum.obj', but still OS does not understand it.

⑤ After that few files from library files ⑦ (contains code of predefined object and function) included using linker and convert the code into 'sum.exe', which is now understable to OS.

Note-
- header files contains only declaration of predefined object or functions
- library file contains the actual code of all the predefined object or function.
- Eg- #include <stdio.h> → stdio.h → contains only declaration of functions like printf and scanf
  ↓
  code is present in library files.

Eg. cin, cout declaration is present in iostream.h but code is present in library files.

Eg- square.cpp.

```
#include <iostream.h>   <- Declaration of cin, cout

int main()
{
    int x;   <- Declaration.
    cout << "Enter a number" << endl;
    cin >> x;
    int s = x * x;   <- Dynamic initialization.
    cout << "square of " << x << " is " << s;
    return 0;
}
```

Action statements

Note -> In C++, it is necessary to first declare predefined Object before using them. i.e. must include header files

Reference variable -

1.) Reference means address.
2.) Reference variable is an internal pointer.
3.) Declaration of Reference variable is preceded with '&' symbol (but don't read it as 'address of').
4.) Reference variable must be initialized during declaration.
5.) It can be initialized with already declared variables only.
6.) Reference variables cannot be updated.

```
int x = 6;
int &y = x;

y++;
cout << x;   // 7
```

x & y

7
100

Date 29/09/2020

inline function -

- Every times a function is called it takes some overhead time to initialize its variable and to return its value.
- To prevent this time consumption, inline functions concept came.
- It works only for small function where there is less things to do.
- Inline is a request, not command. It depends upon the compiler whether it treats function as inline or normal function.
- Syntax - inline int square(int a){

```
    retar a*a;        // function definition & declaration.
}
```

Structure in C++ -

1.) In C, only variables i.e. properties of a particular thing is defined in a structure.

2.) But in C++, properties along with their methods are defined in C++ structure. (encapsulation)

3.)
```
struct book
{   private:
    int bookid;
    char title[20];
    float price;
    public:
    void input()
    {   cout<<"Enter bookid, title, and price";
        cin>> bookid>>title>> price;
    }
    void display()
    { cout << bookid <<" " << title << " " << price;
    }
};
int main()
{
    book b1;  // in C++, 'struct' keyword is not necessary
              before book as required in 'C' to declare.
              a variable of book type.

    b1.input();
    b1.display();
    return 0;
}
```

4.) Structure member are now be more secure by providing access specifier like private, public.

5.) By default, all membess are public.

Note:- The only difference blw class and structure in C++ is that by default, all members of class are private whereas by default, all members of structures are public.

▶ Defining member function outside of class -

```
class complex
{   int a, b;
    public:
    void set_data(int, int);  // function must be declared inside class.
};
void complex :: set_data(int x, int y)   (:: membership operator.)
{   a=x; b=y;
}
```

#Implement complex numbers addition using class.

Soln-

```
#include<iostream>
using namespace std;
class Complex
{   private:
      int a,b;
    public:
      void set_data(int x, int y)
      { a=x, b=y;
      }
      void show_data()
      { cout<<"a="<<a<<"b="<<b;
      }
      Complex add(Complex c)
      { Complex temp;
        temp.a = a + c.a;
        temp.b = b + c.b;
        return temp;
      }
};
int main()
{   Complex c1,c2,c3;
    c1.set_data(5,3);
    c2.set_data(4,7);
    c3= c1.add(c2);
    c3.show_data();   // a=9  b=10.
    return 0;
}
```

Classes and Objects-

- Class is a description of an object.
- Object is an instance (eg. example) of a class.

- Instance member variable -
  - Attributes, data members, fields, properties.
- Instance member functions.
  - Methods, procedure, actions, operations, services.

Static Member in C++ -

① Static local variable -

```
Eg-  int fun()
     { static int x;
       int y;
       return x;
     }
```

→ x is initialized with '0', it remains or exists till the program not terminates.

② static member variable -

```
Eg-  class Account
     { private:
         int balance;          // instance member variable
         static float roi;     // static member variable or class variable.

       public:
         void setBalance(int b)
         { balance = b; }
     };

     float Account::roi = 3.5f;   // Explicit declaration of static member variable.
                                  // is required otherwise no memory will be allocated for it.
     int main()
     {
       Account a1;   // a1 object only contain balance as member variable.
       return 0;
     }
```

membership label.

③ Static member function -

→ # static void setRoi(float r) // static member function.

```
{ roi = r; }

int main()
{
  Account::setRoi(4.5f);
  return 0;
}
```

// a1.set(4.5f), This is also valid

Class function. or static member functions.

They can only access static members of the class.

Static member function could also be accessed by using object of the class.

Date 29/09/2020.

* function overloading -
  - name of function will be same.
  - the no. of arguments must differ.
  - Eg - int sum (int a, int b) , int sum (int a, int b, int c).

Constructor -

* These are the function called implicitly by the compiler (during) after a class object is declared. (default constructor).
  - has same name as of class.
  - no return type.
  - used to initialize member variables.

1.) default constructor.
2.) parametrized constructor.
3.) Constructor overloading.

Eg
```
Complex (int x, int y)
{ a = x, b = y; }

complex (int x)
{ a = x, b = 0; }.
```

Copy constructor -

- used to initialize object's member variable with another or already declared object.
- Eg - Complex c1 (c2); // Complex c1 = c2;
- // copy constructor - (Reference ↓)

```
Complex ( Complex& )
{ a = c.a;
  b = c.b;
}
```

Destructor -

- It should be defined to release resources of an object.
- Every object has single destructor function.
- Eg -

```
~ Complex ( )
{ cout << "Destructor called!";
}.
```

Operator overloading -

* When an operator is overloaded with multiple jobs, it is known as operator overloading.
* It is a way to implement compile time polymorphism.

A symbol can be used as function name -
- if it is a valid operator in 'C' language.
- if it is preceded by operator keyword.

Eg-
```
class Complex
{ private:
    int a, b;
  public:
    Complex (int x, int y)
    { a = x, b = y; }

    Complex ( ) { }
    void show ( )
    { cout << "a = " << a << "b = " << b; }
    Complex operator + (Complex c)
    { Complex temp;
      temp.a = a + c.a;
      temp.b = b + c.b;
      return temp; }
};

int main ( )
{ Complex c1 (3,4), c2 (5,6);
  Complex c3;
  c3 = c1 + c2;   // depicted as: c1.operator+(c2)
  c3.show ( );    // a = 8  b = 10
  return 0;
}
```

Caller function (left).

Binary overloading.

Unary operator overloading - (-)

```
Complex Complex :: operator - ( )
{ Complex temp;
  temp.a = - a, temp.b = -b;
  return temp;
}
int main ( )
{ Complex c1(2,3), c2;
  c2 = -c1;   // depicted as. c1.operator-( );
  c2.show( );
  return 0;
}
```

Operator overloading - preincrement , postincrement -

Eg -

```
class Integer
{   private:
        int x;
    public:
        void setData(int i)
        { x = i; }

        Integer operator ++ ( )  // preincrements.
        { Integer i;
          i.x = ++x;
          return i;
        }

        Integer operator ++ (int)  // post increment.
        { Integer i;                  ← this reminds compiler in differentiating b/w prefix & postfix.
          i.x = x++;
          return i;
        }
        void show( )
        { cout << "x = " << x;
        }
};

int main( )
{   Integer i1;
    i1.setData(5);
    i1.show( );      // x = 3
    Integer i2, i3;
    i2 = ++i1;
    i3 = i1++;
    i1.show( );      // x = 5
    i2.show( );      // x = 4.
    i3.show( );      // x = 4.
    i1.show( );      // x = 5.
    return 0;
}
```

Date 30/09/2020

friend function -

- Friend function can access any member of the class to which it is friend.
- Friend function cannot access members of the class directly
- It has no caller object.
- It should not be defined with membership label.
- Example -

```
class Complex
{   int a,b;
    public:
        void set_data(int x, int y)
        {  a=x, b=y; }

        friend void fun(Complex); // only declaration of friend function is required in class.
};
void fun(complex c) // friend function definition
{  cout<< "sum of a and b :" << c.a + c.b; // 5
}
int main()
{
    Complex c1;
    c1.set_data(2,3);
    fun(c1); // call to friend function fun()
}
```

Usage of friend function -

If Friend function can become friend to more than one class, i.e. friend friend can access private members of two or more classes simulatenously of which it is declared as a friend function.

Eg -

```
class B;
class A
{  int a;
   public:
      void setData(int x) { a=x; }
      friend void fun(A,B);
};
class B
{  int b;
   public:
      void setData(int x) { b=x; }
      friend void fun(A,B);
};
```

```
void fun ( A o1 , B o2 )
{  cout << " Sum of A+B is : " << o1.a + o2.b ;
}
int main( )
{  A obj1 ;
   B obj2 ;
   obj1.setData(2);
   obj2.setData(3);
   fun(obj1, obj2);
   return 0;
}
```

2) Overloading of operators as a friend function -

Eg-

```
class Complex
{  private:
      int a, b ;
   public:
      void setData ( int x, int y ) { a = x , b = y ; }
      void showData( ) { cout << " a = " << a << " b = " << b ; }
      friend Complex operator + (Complex, Complex);
};
Complex operator + (Complex x, complex Y)
{
   Complex temp;
   temp.a = x.a + Y.a ;
   temp.b = x.b + Y.b ;
   return temp;
}
int main( )
{ complex c1, c2, c3;
  c1.setData ( 3, 4);
  c2.setData ( 4, 5);
  c3 = c1 + c2 ;   // operator + (c1, c2)
  c3.showData( ) ; // a = 7  b = 9
  return 0;
}
```

3) Overloading of unary operator as a friend function.

Eg-

```
Class Complex
{ private:
    int a, b;
  public:
    void setData (int x, int y) { a = x, b = y;}
    void showData ( ) { cout<< "a = " <<a<< "b = " <<b;}
    friend Complex operator - (Complex);
};
Complex operator - (Complex x)
{   Complex temp;
    temp.a = -x.a;
    temp.b = - x.b;
    return temp;
}
int main ( )
{   Complex c1,c2;
    c1.setData (5, -2);
    c2 = -c1;  // operator - (c1)
    c2.showData ( );  // a = -5  b = 2
    return 0;
}
```

4) Overloading of insertion (<<) and extraction (>>) operator-

```
Eg - Class Complex
   { private:
       int a,b;
     public:
       friend ostream& operator << (ostream &, Complex);
       friend istream& operator >> (istream &, Complex);
};   Reference of class, since object of ostream & istream is not created
ostream & operator << (ostream& dout, Complex c)
   { cout<< c.a << " " << c.b;
     return dout;
   }
istream & operator >> (istream& din, Complex& c)
   { cin>> c.a >> c.b;
     return din;
   }
int main ( )
   { Complex c1;
     cin>> c1;  // operator >> (cin, c1)
     cout<< c1;  // operator << (cout, c1)
   }
```

⇒ Member function of one class can become friend to another class.

Eg -
```
class A
{ public:
    void func();
    { ... }
    void foo()
    { ... }
};

Class B
{
    friend class A;   // To make all member function of class A as a friend function of class B
};  //friend void A::fun();
```

To explicitly make member function of class A as a friend function of Class B.

Date 01/10/2020

Understand the need of inheritance -

- class is used to describe properties and behaviour of an object.
- property names and values.
- Behaviour means action.
- Eg -

Car

properties	Methods
price	setPrice()
fuelType	setFuelType()
engine	setEngine()
color	setColor()
capacity	setCapacity()
	getPrice()
	getFuelType()
	getEngine()
	getColor()
	getCapacity()

Sports car

properties	Methods
price	setAlarm()
fuelType	setNavigator()
engine	setSafeGuard()
color	getAlarm()
capacity	getNavigator()
alarm	getSafeGuard()
navigator	setPrice()
SafeGuard	setFuelType()
	setEngine()
	setColor()
	setCapacity()
	getPrice()
	getFuelType()
	getEngine()
	getColor()
	getCapacity()

⇒ Sports has properties and methods of car class but with additional properties and methods.

⇒ To reuse the car class to develop sport class, inheritance came into existence.

Inheritance -

- It is a process of inheriting properties and behaviours of existing class into a new class.
- Existing class = old class = Parent class = Base class
- New class = Child class = Derived class.

Syntax -

```
class Base_class
{
};
class Derived_class : Visibility-Mode Base_class
{
};
```

- Eg -

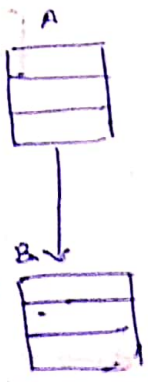
```
class Car
{
};
class SportsCar : public Car
{
};
```

Types of Inheritance -

- Single Inheritance.
- Multilevel Inheritance.
- Multiple Inheritance.
- Hierarchical Inheritance.
- Hybrid Inheritance.

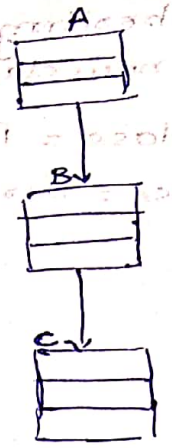
Single Inheritance -

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



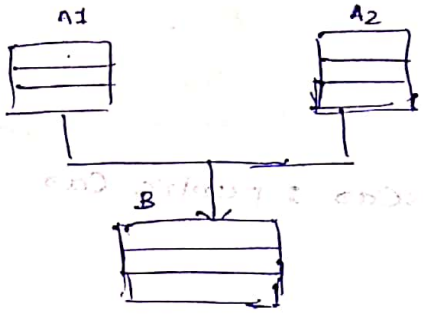
Multilevel Inheritance -

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



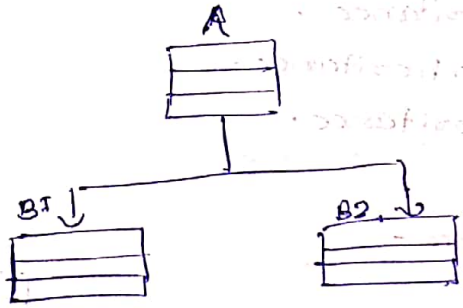
Multiple Inheritance -

```
class A1
{
};
class A2
{
};
class B : public A1, public A2
{
};
```

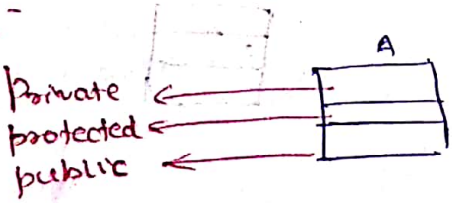


Hierarchical Inheritance -

```
class A
{
};
class B1 : public A
{
};
class B2 : public A
{
};
```



Visibility Modes -



- Types of users of a class -
  - User 1 will create object of ~~your~~ class.
  - User 2 will derived class from our class.

• Availability Vs Accessibility -

1) private, protected and public members are available for the user1 who made object of our class. But only public members are accessible to the user.

⇒ private members are not accessible to both user1 & user2.

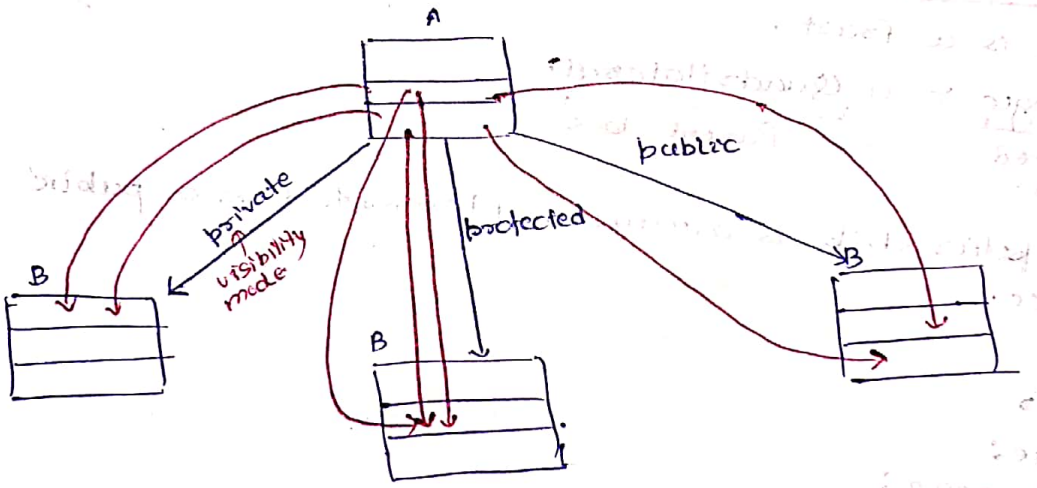
2) protected members are not accessible to user1, but it is accessible to the user2, who has derived class from our class.

3) public members are accessible to both user1 and user2.

Visibility mode explanation -

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

};
```

```
class B : A        (by default private)
{
};
```



1) If visibility mode is private, then, class A's protected member & public member in class B's ~~object~~ is treated as private for user2 of class B.

2) If visibility mode is protected, then, class A's protected & public members in class B's is treated as protected for user 2 of class B.

3) If visibility mode is public, then, class A's protected member & public member in class B's is treated as protected and public for user 2 of class B.

Eg-
```
class A
{
  private:
   int a;
  protected:
     void setValue(int k)
      { a=k; }
};
class B : public A
{
  public:
     void setData(int x)
      { setValue(x); }
};
int main()
{
  B obj;
//obj.setValue(4); Users cannot access protected members of a class
  obj.setData(4);
}
```
↑ setValue() is protected in class B. i.e. users.

Is-a Relationship -

Eg- Banana is a fruit.

Rectangle is a Quadrilateral.
(Derived Class.) (Parent class)

=> 'Is a' relationship is always implemented as a public inheritance.

Eg-
```
class Car
{ private:
    int gear;
  public:
    void incrementGear()
     { if(gear<5)
         gear++;
     }
};
class SportsCar : public Car
{
};
int main()
{ SportsCar c1;
  c1.incrementGear(); // incrementGear() can only be accessible by SportsCar object if and only if SportsCar is public inherited.
}
```

When to use protected, private inheritance -

Eg.

```
class Array
{   private:
        int a[10];
    public:
        void insert (int index, int value)
        {   a [index] = value; }
};
class STACK : protected Array
{   int top;
    public:
        void push (int value)
        {   insert (top, value);
        }
};
int main()
{   STACK s1;
    s1.push(20);
    // s1.insert(2,50);
}
```

// if this could be possible then the rule of stack will be voilated that elements inserts always on top, not on random indexs.

But since STACK is inherited as a protected from Array, thus public member of Array in class STACK becomes protected and thus not accessible by STACK's object.

Date
02/10/2020

Constructor and destructor in C++ -

- We know that constructor is invoked implicitly when an object is created.
- In inheritance, when we create object of (child) derived class, what will happen?
  - first constructor of derived class is called by the object, but constructor of derived class, calls the constructor of parent class and execute it first and then execute its own code.

- Eg -

```
class A
{ private:
    int a;
  public:
    A(int k)   // parent class constructor executed first.
    { a=k; }
    ~A(){ }  // After derived class destructor, parent class destructor is called.
};
class B : public A
{ private:
    int b;
  public:
    B(int x, int y) : A(x)   // B's constructor calls its parent class constructor first.
    {
      b=y;   // then derived class constructor executes.
    }
    ~B(){ }  // first object's destructor calls & executed
};           // derived class
int main()
{
  B obj(2,3);  // B's constructored is called.
} return 0;
```

Object pointer -

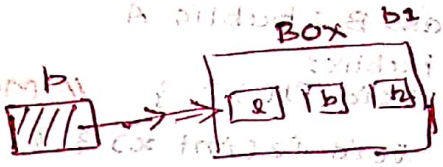
- A pointer contains address of an object is called Object pointer.
- Eg - 'this' pointer.

'this' pointer -

- 'this' is a keyword.
- 'this' is a local object pointer in every instance member function containing address of the caller object.
- 'this' pointer cannot be modify.
- It is used to refer caller objects in member function.

• Eg -

```
class Box
{  private:
     int l, b, h;
   public:
     void setDimension(int l, int b, int h)
     { this->l = l, this->b = b, this->h = h; }
     void showDimension ( )
     {   cout<<"l = "<< l <<"b = "<< b <<"h = "<< h;
     }
};
int main( )
{   Box b1,*p; // p is a pointer to object.
    p = &b1;
    p->setDimension (2,4,6);
    p->showDimension ( );
}
```



Similary,

new and delete in C++ :-

- SMA - static memory allocation (memory consumption decided at compile time)
- DMA - Dynamic memory allocation. (memory allocation at runtime)

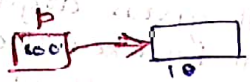
```
int x;
float y;
Complex c1;
Student s1;
```

← Declaration statement } SMA.

new -

```
int *p = new int;
```

→ memory is allocated at runtime using 'new' keyword.




```
float *q = new float;
Complex * ptr = new Complex;
```

`float *q = new float[5];`


```
int x;
cin>>x;
int *p = new int[x]; // variable size array using new key word.
```

delete :- delete memory created using new key word.

Eg - `delete p;` // simple variable memory release.

`delete []p;` // array variable memory release.

Method overriding in C++ -

- Method overloading
- Method overriding
- Method Hiding.

Eg-

```
class A
{
    public:
        void f1(){ }
        void f2(){ }
};

class B : public A
{ public:
    void f1(){ }    // Method overriding
    void f2(int x){ }  // Method hiding.
};

int main()
{ B obj;
  obj.f1();  // B's f1 will be called.
  obj.f2();  // error
  obj.f2(4); // B's f2 will be called.
}
```

Early Binding (Binding of function at compile time).

if the same function is defined in parent & child class, then if it is called by the object of child class, then the function defined in child class will be executed. (This is called method overriding i.e. method of parent is overrided by child)

if the same function is defined in parent & child class but with different argument, then function in child with given argument will be executed, only if the child's object, passes exact argument as defined in child class, if it is no such, then error will occur, even though the parent has function with same name and same arguments. (This is called method hiding).

Conclusion - The parent class function will only be called by the object of its child class, if its child has not defined those functions.

Method overloading - functions with same name but different arguments within the same class.

Virtual Function - (Parent)

Base class pointer -

- Base class pointer can point to the object of any of its descendant class.
- But its converse is not true.

Eg-

```
class A
{
  public:
virtual void f1(){ }
};
class B: public A
{ public:
      void f1(){ }   // Method overriding -
      void f2(){ }
};
int main()
{ A *p, o1;
  B o2;
  p = &o2;   // address of child is assigned to pointers of parent class
  p->f1();   // Due to early binding, A's f1 function will be called as during compilation early binding is done which states that based on the type of pointer, call its function.
}
```

virtual keyword is written Before f1 in parent class, then late binding will occur and B's f1() will be called.

After

Since, p is of A's type, thus A's f1() will be called not B's, inspite of B's address is assigned into p.

To solve this binding problem, Concept of virtual function came into existence. ie. late binding or binding at runtime.

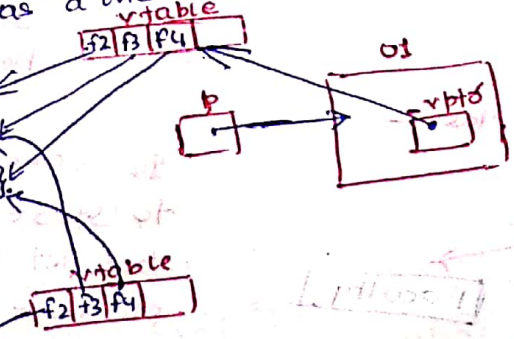
virtual Keyword is used for late binding of functions.

How virtual function work -

Eg-

```
class A
{ public:
    void f1(){ }
    virtual void f2(){ }
    virtual void f3(){ }
    virtual void f4(){ }
};
class B : public A
{ public:
    void f1(){ }
    void f2(){ }
    void f4(int x){ }
};
```

*vptr ——→ Class having at least one virtual function, *vptr is associated to it as a member by the compiler.

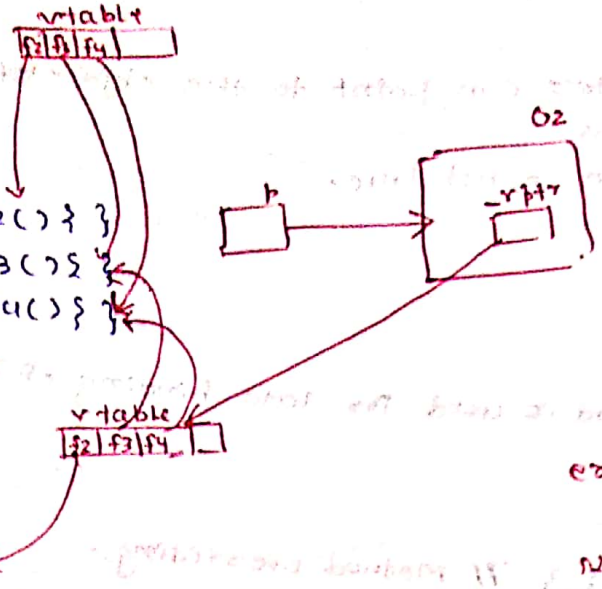



```
main()
{
  A *p, o1;
  p = &o1;
A -> p->f1();  //EB
A -> p->f2();  //LB
A -> p->f3();  //LB
A -> p->f4();  //LB
eeeee -> p->f4(5);  //EB
}
```

No such function exists with same parameter or arguments

Eg-

```
class A
{
  *_vptr
  public:
    void f1(){}
    virtual void f2(){}
    virtual void f3(){}
    virtual void f4(){}
};

class B : public A
{
  public:
    void f1(){}
    void f2(){}
    void f4(int x){}
};
```




```
main()
{
  A *p;
  B O2;
  p = &O2;   // B's obj address
```

A's → p→f1(); EB

B's → p→f2(); LB

A's → p→f3(); LB

A's → p→f4(); LB

error → p→f4(5); EB

No A's member function f4() has altered or accepting single argument.
(∵ Early binding, it doesn't point to B's object).

Date
03/06/2020

Pure virtual function-

- A do nothing function is pure virtual function.

Eg-

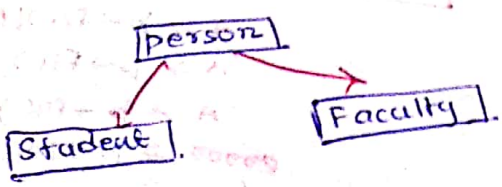
```
class Person
{ public:
    virtual void fun() = 0; // pure virtual function.
      void f1(){}
};

class Student : public Person
{ public:
    void fun()
    {
    }
};
```

↓
class containing at least one pure virtual function is called abstract class

↓
no object of abstract class is made.

↓
But derived class's object from abstract class is could be made.

Why abstract class:-



In school or college, we want to store data of student & faculty, but not of a person. But since student & faculty passes properties & methods of person class, and we do not want to create person's class object, then in that case Person class is made abstract.

abstract class helps in generalization of code.
i.e. we can derived many classes from abstract class without creating abstract class object.

Template:-

- The keyword 'template' is used to define function template and class template.
- It is a way to make your function or class generalize as far as data type is concern.

- Eg -

Function Template:-

```
template<class T>          → Function template used for generalize function of every data types.
T big(T a, T b)
{   if(a>b)
        return a;
    return b;
}.

int main()
{
    cout<< big(4,5) <<endl;      // passed 'T' is replaced with
    cout<< big(5.5, 3.4) <<endl; // that data type.
}.  return 0;
```

→ depending upon the type data

- Function template is also known as generic function.

Syntax- `template <class type> type func_name(type arg1, ...);`

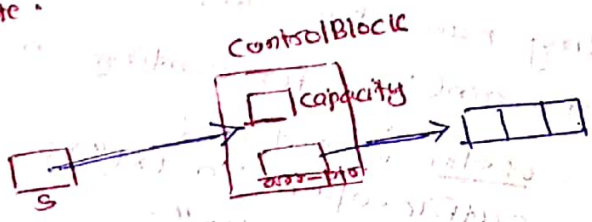
Class Template:-

- class template is also known as generic class.

- syntax → `template <class type> class class_name{...};`

- Eg-

```
template<class I>  // class Template.
class ArrayList
{   private:
        struct controlBlock
        {   int capacity;
            T * arr_ptr;
        };
        controlBlock * s;
    public:
        ArrayList(int capacity)
        {   s = new controlBlock;
            s->capacity = capacity;
            s->arr_ptr = new T[s->capacity];
        }
};
int main()
{   ArrayList <int> list1(4);
}
```

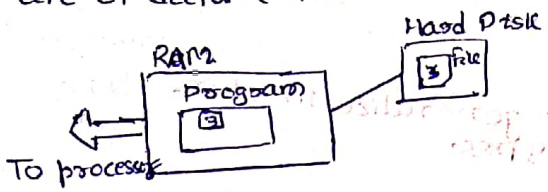


↳ pass the data type accordingly. (Generic class).

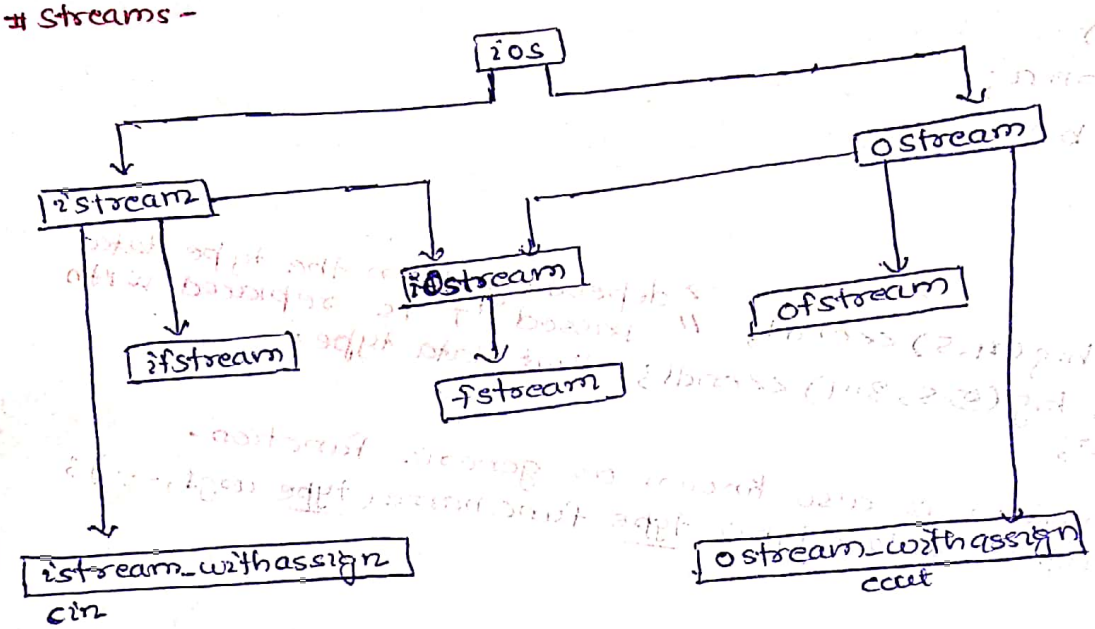
Date 04/10/2020

File handling in C++ :-

- Data Persistence -
  - * Life of data (existence of data):



Streams -



Eg - To write in a file -

```
#include<fstream>
using namespace std;
int main()
{   ofstream outFile;        // for writing in a file.
    outFile.open("a.txt");
    outFile<<"Hello";
    outFile.close();
}.
```

Eg- To read from a file -

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{   ifstream fin;            // for read from a file.
    char ch;
    fin.open("a.txt");
    fin>>ch;                 // >> operator treats spaces (' '), newline ('\n')
    while(!fin.eof())        // and tab ('\t') as
    {  cout<<ch;             // delimiter. thus >> will
       fin>>ch;}             // ignore spaces and move
    fin.close()              // ahead.
}
```

To store spaces from file -

```
ch = fin.get();
```

File opening modes -

- ios::in — input / read
- ios::out — output / write.
- ios::app — append
- ios::ate — update
- ios::binary — binary

Syntax -

```
ofstream fout;
fout.open("file name", file_opening_mode);
```

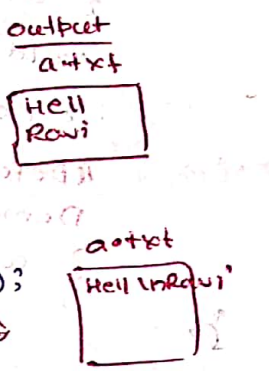
Eg-

```
fstream fout;
fout.open("a.txt", ios::out | ios::in);
```

Diff. b/w binary mode & text mode -

Eg-

```
ofstream fout;
fout.open("a.txt", ios::out);
fout << "Hello \n Ravi";
fout.close();
fout.open("a.txt", ios::binary | ios::out);
fout << "Hello \n Ravi";
```



→ By default, files are opened in text mode, in which \n, \t is special meaning is considered.

Initializer List -

- Initializer list is used to initialize data member of a class.
- The list of members to be initialized is indicated with constructor as a comma seperated list followed by a colon.
- There are situations where initialization of data members inside constructor doesn't work and Initializer List must be used.
  - For initialization of non-static const data members.
  - For initialization of reference members.

Eg-

```
class Dummy
{ private:
    int x;
    const int y;
    int &z;
  public:
    Dummy(int &a) : y(5), z(a)
    {
    }
};

int main()
{ int m=6;
  Dummy d1(m);
}
```

const variable is initialized

Initializer list.

m variable passed as reference to initialize z.

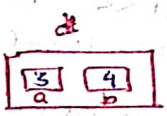
Deep Copy & Shallow copy –

- Copy - How can we create a copy of object.
  - Copy constructor.
  - Implicit copy assignment operator.

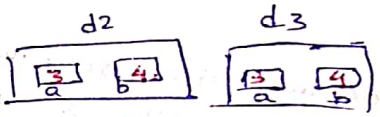
Shallow Copy :-

Eg-

```
class Dummy
{  private:
     int a;
     int b;
   public:
     void setData(int x, int y)
     { a=x, b=y; }
     void showData()
     { cout<<"a="<<a<<"b="<<b; }
   // Default copy constructor looks like -
     Dummy (Dummy &d)
     { a=d.a; b=d.b; }.
};

void main()
{
    Dummy d1;
    d1.setData(3,4);
    Dummy d2 = d1; // Default copy constructor called.
    Dummy d3;
    d3=d1; // Implicit copy assignment operat called.
    d2.showData();
    d3.showData();
}.
```



After d2 = d1 , d3 = d1



Default copy constructor & implicit copy assignment operator simply copies all data members of object into another.

⇓

This is called shallow copy.

Deep copy –

- Creating an object by copying data of another object along with the values of memory resources resides outside the object but handled by that object.

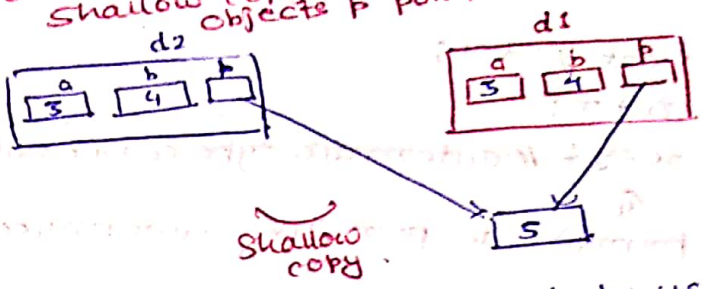
Eg-

```
class Dummy
{  private:
     int a, b;
     int *p;
   public:
     Dummy ()
     { p = new int; }
```

```
void setData(int x, int y, int z)
{ a=x, b=y, *p=z; }

void showData()
{ cout<<"a="<<a<<"b="<<b<<"*p="<<*p; }

Dummy(Dummy &d)   // copy constructor made by programmer.
{ a=d.a;
  b=d.a;
  p=new int;
  *p=*(d.p);
}
};

int main()
{ Dummy d1;
  d1.setData(3,4,5);
  Dummy d2=d1;
  d2.showData();
}
```

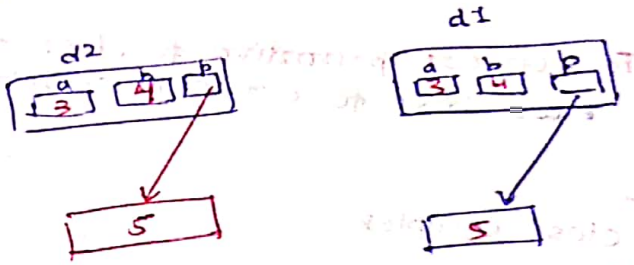
// Destructor should also be made to release memory -
~Dummy() { delete p; }

- if default, copy constructor called then shallow copy will occur and both objects p points to same memory location.



Shallow copy.

- if copy constructor, made by us, then shallow copy should be avoided for class having pointer as a data members.



Deep copy.

Problems with shallow copy -

→ two objects pointers, points to the same memory location.

→ Even if the first object deletes and releases the memory of pointer, the second pointer still pointing to that memory location, which is harmful.

→ Also, each objects should have their own resources, which is not possible in shallow copy, if a class is having a pointer as data member.

Date 05/10/2020

Type conversion primitive to class type -

- int, char, float, double are primitive types.
- class type is any class that we defined.

Eg -

```
int x = 4;
float y;
y = x; // automatic type conversion      // y = 4.0
```

```
float y = 3.4;
int x;
x = y; // automatic type conversion      // x = 3
```

⇑ primitive to primitive automatically type conversion happened.

```
Complex c1;
int x = 5;
c1 = x; // error // primitive to class type conversion could not be automatically happened.
```

⇒ To convert primitive to class type -

- we have to write a constructor for it in the class.

Eg -

```
class Complex
{ private:
    int a, b;
  public:
    void setData (int x, int y)
    { a = x; b = y; }
    void showData ()
    { cout << " a = " << a << " b = " << b ; }
    Complex()
    { }
    Complex (int x)          // This constructor is called for the assignment of int into complex i.e. primitive to class.
    { a = x, b = 0; }
};
int main()
{ Complex c1;
  int x = 4;
  c1 = x;                    // depicted as c1.Complex(x)
  c1.showData(); //  a = 4 b = 0
  return 0;
}
```

→ To convert class type to primitive type -

Eg - Complex c1;
c1.setData(3,4);
int x;
x = c1; // error // class type could not be converted into primitive type automatically.

casting operators -

- class type to primitive type can be implemented with casting operator.

- operator type()
{ ...
return (type-data);
}

Eg-
class Complex {
private:
int a, b;
public:
void setData(int x, int y)
{ a=x, b=y;}
void showData()
{ cout<<"\n a= "<<a<<" b = "<<b;}

operator int() // casting operator.
{ return a;
}

};
int main()
{ Complex c1;
c1.setData(3,4);
c1.showData(); // a = 3 b = 4
int x;
x = c1; // depicted as c1.operator int()
cout<<"\n x = "<<x; // x = 3
return 0;

}

⇒ Type Conversion of one class type to another class type-

- This is possible using -
  - conversion through constructor
  - conversion through casting operator.

Eg- class Item; void Item::setData(int, int);

```
class Product
{  private:
       int m,n;
   public:
       void setData(int x, int y)
       { m=x, n=y; }
       void showData( )
       { cout<<"\nm = "<<m<<"n = "<<n;}
       operator Item( )        // casting operator converts Product type into Item type.
       {    Item i1;
            i1.setData(m,n);
            return i1;
       }
       int getM( ) {return m;}
       int getN( ) {return n;}
};
class Item
{  private:
       int a,b;
   public:
       void setData(int x, int y)
       { a=x, b=y;}
       void showData( )
       { cout<<"\n a = "<<a<<"b = "<<b; }
       Item( ) { }
       Item ( Product P )      // Casting of Product into Item using constructor.
       {    a = p.getM( );
            b = p.getN( );
       }
};
int main( )
{   Item i1;
    Product p1;
    p1.setData(3,4);
    i1 = p1;
    i1.showData( );   // a = 3   b = 4
    return 0;
}
```

Date 06/10/2020

Exception Handling -

- **Exceptions -**
  - Exception is any abnormal behaviours, runtime error.
  - Exception are off beat situation in our program where our program should be ready to handle it with appropriate response.

- C++ provides a built-in error handling mechanism that is called exception handling.
- Using exception handling, we can more easily manage and respond to runtime errors.

- **try, catch, throw -**
  - Program statements that we want to monitor for exceptions are contained in a try block.
  - If any exception occurs within the try block, it is thrown (using throw).
  - The exception is caught, using catch, and processed.

Syntax -

```
try
{
}
catch (type1 arg)
{
}
catch (type2 arg)
{
}
.
.
.
catch (typeN arg)
{
}
```

Eg -

```
void fun()
{  throw 10;
}
int main()
{   int i = 3;
    cout<<"Welcome";
    try{  if (i==3)
              fun();
          cout<<"\n Try";
    }
catch (int){
    cout<<"\n Exception no:-";
}
catch (double e)
{  cout<<"\n Exception" <<e;
}cout <<"Last Line";)
}
```

output → Welcome
Exception no:-
last line.

#throw-

- The general form of the throw statement is:
  throw exception;
- Throw must be executed either within the try block or from any function that code within the blocks calls.

#Dynamic Constructor-

- Constructor can allocate dynamically created memory to the object
- Thus, object is going to use memory region, which is dynamic created by constructor.

Eg-

```
class A
{ private:
    int a,b;
    int *p;
  public:
    A()                          // Dynamic constructor
    { p=new int;
      a=0, b=0, *p=0;
    }
    A(int x, int y, int z)       // Dynamic constructor
    { p = new int;
      a=x, b=y, *p=z;
    }
};
int main()
{
  A ob1, ob2(2,3,4);
}
```

constructor calls for '0' arguments

this constructor call for 3 arguments.

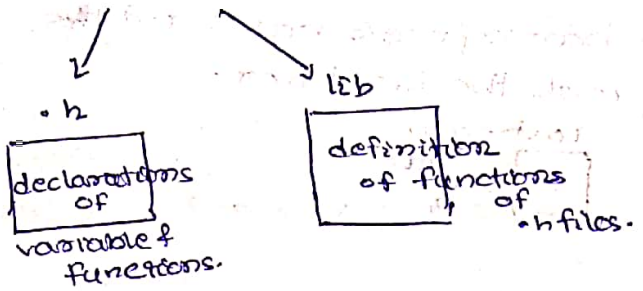
objects made using class A

↓

because it is allocating dynamically created memory to the object's member p.

#Namespaces in C++ -

Eg- #include <stdio.h>



- **Namespace** -
  - Namespace is a container for identifiers.
  - It puts the names of its members in a distinct space so that they don't conflict with names in other namespaces or global namespace.

- **how to create namespace** -

Syntax - Eg.

```
namespace Myspace{
  // Declarations
}
```

- Namespaces definition doesn't terminates with a semicolon like in class definition.
- The namespace definition must be done at global scope, or nested inside another namespace.
- We can use an alias name for our namespace name, for ease of use.
  - Eg - namespace ms = Myspace;

- **Unnamed namespaces** -
  - There can be unnamed namespaces too.
  - namespace
    { // declarations
    }

⇒ Namespace is not a class, we cannot create instance of namespace.

- **namespaces can be extended** -
  - A namespace definition can be continued and extended over multiple files, they are not redefined or overridden.

Eg- file1.h

```
namespace Myspace
{ int a,b;
  void f1();
}
```

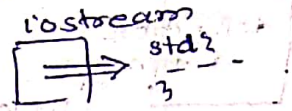
file2.h

```
namespace Myspace
{ int x,y;
  void f2();
}
```

} consider's both file1's & file2's myspace definition as a single definition.

Accessing members of namespace -

- Any name (identifier) declared in a namespace can be explicitly specified using the namespace's name and the scope resolution :: operator with the identifier.



Eg-

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

namespace Myspace{
        int a;
        int f1();
        class A
        {  public:
              void fun1();
        };
}

int  Myspace::f1(){
     cout<<"Hello f1";
     return 0;
}
void Myspace::A::fun1()
{  cout<<"Hello fun1";
}
int main()
{   Myspace::a = 5;
    Myspace::f1();     // Hello f1
    A obj;
    obj.fun1();        // Hello fun1
}  return 0;
```

The using directive -

- using keyword allows us to import an entire namespace into our program with a global scope.
- It can be used to import a namespace into another namespace or any program.

File1.h

```
namespace Myspace
{ int a,b;
  class A{
    // same code.
  };
}
```

File2.h

```
#include "file1.h"
namespace myNewspace
{ using namespace Myspace;
  int x,y;
  A obj;
};
```

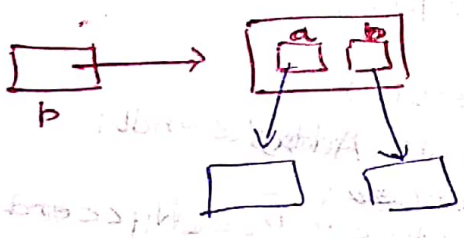
virtual destructor –

Eg- class A

```
{   int *a;
public:
    void f1(){}
    virtual ~A(){ --- }   // virtual destructor for late binding.
};
class B : public A
{   int *b;
public:
    void f2(){}
    ~B(){ ---};
};
void fun()                    // pointer is declared of type class A but pointing to B class's object.
{   A *p = new B;
    p->f1();   // correct, Due to early binding A class's f1() will be called.
    p->f2();   // error, Due to early binding, A class's function will be called, but there is no function in 'A' with name f2, thus error happens
    delete p;
}
int main()
{   fun();
}
```

Now, A's destructor will be called (due to early binding) but this should not be happen, first derived class's object destructor should be ran, then its parent destructor should be ran.

void func(){



}

if early binding occurs, then only A's memory which was dynamically allocated is going to be released.
But this is not right.

To prevent this –
we make destructor of parent class as virtual for late binding.
After this, first derived class destructor will be called then parent class destructor
This is the right way.

Date
07/10/2020

Nested class -

- Class inside a class is called nested class (inner class).
- A nested class is a member and has the same access rights as any other member.
- The member of an enclosing class have no special access to member of a nested class; the usual access rules shall be obeyed.
- Eg -

```
Class Student
{  private:
       int rollno;
       String name;
       class Address // nested class defined in private member of class student
       {  private:
              int houseno;
              string street;
              string city;
              string state;
              string pin;
          public:
              void setAddress (int housno, string st, string ct
                               , string state, string pin)
              {  this -> houseno = houseno;
                 street = st;
                 city = ct;
                 this -> state = state;
                 this -> pin = pin;
              }
              void showAddress ( )
              {  cout << "student's Address" << endl;
                 cout << houseno << " ";
                 cout << street << " " << city << endl;
                 cout << state << " " << pin << endl;
              }
       };
       public:
              Address ad; // nested class object declaration.
              void setName (string name)
              { this -> name = name; }
              void setRollno (int r)
              { rollno = r; }
              void showData ( )
              {  cout << "student's data" << endl;
                 cout << rollno << " " << name << endl;
                 ad.showAddress ( );
              }
```

```
void setAddress (int hn, string st, string ct, string state, string pin)
{
    ad.setAddress(hn, st, ct, state, pin);    // Address is set using address object.
}
};
int main()
{
    Student s;  // Student's object declaration.
    s.setName("Ravi");
    s.setRollno(12);
    s.setAddress(204, "Mirchaibari", "Katihar", "Bihar", "854105");
    s.showData();
    return 0;
}
```

Output - Student's data
12 Ravi
Student's Address
204
Mirchaibari, Katihar
Bihar 854105

Note -
The need of address class inside student is that since practically Adress object is meaningless unless it is associated with some meaningful class.
Since, Student has property of Address, Thus Address class is defined inside of it as a member.

⇒ `Student::Address adobj;` // It is valid if address class in public part of Student class.

Date 08/10/2020

STL -

- STL is standard Template Library.
- It is a powerful set of c++ templates classes.
- At the core of the c++ standard Templates library. These are the following three well-structured components:
  - Containers.
  - Algorithms.
  - Iterators.

Containers -

- Containers are used to manage collections of objects of a certain kind.
- Containers library in STL provide containers that are used to create data structure like arrays, linked list, trees etc.
- These containers are generic, they can hold elements of any data types.
- Eg- vector can be used for creating dynamic arrays of char, integer, float and other types.

Algorithms

- Algorithms act on containers. They provide the means by which we will perform initialization, sorting, searching and transforming of the contents of containers.
- Algorithms library contains built in functions that perform complex algorithms on the data structure.

Eg- One can reverse a range with reverse() function, sort a range with sort() function, search in a range, with binary-search() and so on.

- Algorithm library provides abstraction ie. coe dont necessarily need to know the algorithm works.

Iterators -

- Iterators are used to step through the elements of collection of objects. These collection may be containers or subsets of containers.
- Iterator in STL are used to point to the containers.
- Iterators actually acts as a bridge between containers and algorithms.

Eg- sort() takes two parameters, starting and ending iterator to sort the elements.

Containers -

- Container library is a collection of classes.
- The containers are implemented as generic class templates.
- Containers help us to replicate and implement simple and complex data structure very easily like arrays, list, tree, associate arrays and many more.
- Containers can be used to hold different kind of objects.

Common Containers -

- vector : replicates arrays
- queue : replicates queues.
- stack : replicates stacks.
- priority_queue : replicates heaps.
- list : replicates linked list.
- set : replicates trees.
- map : associative arrays.

classification of containers -

- sequence containers.
  - like array, linked list, etc.
- Associative Containers.
  - Sorted Data structures like map, set, etc.
- Unordered Associative containers.
  - unsorted data structures
- Container Adapters.
  - Interfaces to sequence containers.

How to use container library?

- When we use list containers to implement linked list we just have to include the list headers file and use list constructor to initialize the list.

```
#include <iostream>
#include <list>

int main()
{   list <int> mylist;

}.
```

Array -

- Array is a linear collection of similar elements.
- Array container in STL provides us the implementation of static array.
- use header array i.e. #include<array>

#Member functions -

- Following are the important and most used member functions of array template.
  - at
  - [] operator
  - front() → returns element at index 0.
  - back() → returns element at index (n-1)
  - fill() → fill all elements of array with a given value.
  - swap() → swap two array of same type and same size.
  - size() → returns size of array
  - begin() → returns iterator to begin.
  - end() → returns iterator to end +1.

- Eg int main() {
  array <int, 5> data_array = {11,22,33,44,55};
  cout<< data_array.at(2); // 33
  data_array.fill(10); // fill all the elements of data_array with 10.
  array <int, 5> dtI = {1,2,3,4,5} ,10,10,10,10,10}
  data_arry.swap(dtI);
  ↓
  //array values are swapped.
  // data_array = {1,2,3,4,5}
  // dtI = {10,10,10,10,10}

}.

Date 09/10/2020

Pair -

- pair is a template class in standard Template library.
- pair is not a part of containers.
- Syntax -
  pair <T1, T2> pair1;

  eg - pair <string, int> p1; //
- Eg. pair <string, float> p2; // Declaration.
  p2 = make_pair ("c++", 12.30f); // insertion
  cout << p2.first << " " << p2.second; // accessing

comparision b/w two pairs is possible -

- == • != • < • > • <= • >= } for same type of pairs.

→ comparision is possible for tuples of same type.

tuple -

- Just like in pair, we can pair two heterogeneous objects, in tuples we can pair multiple objects.
- syntax - tuple <T1, T2, T3> tuple1;
- Eg - tuple <string, int, int> t1;
  - tuple<string, int, int> t1; // Declaration
    t1 = make_tuple ("Ravi", 15, 30); // Initialization.
    cout << get<0>(t1) << " ";
    cout << get<1>(t1) << " ";
    cout << get<2>(t1);

vector -

- The most popular or general purpose container is the vector.
- It supports dynamic array.
- What is dynamic Array?
  - Array which can extend its size or capacity at runtime
- syntax - vector <T> v; Eg - vector<int> v1 {1,2,3};
  ↑ size vector ↑ size vector

functions -

- begin()
- end()
- at()
- []
- front()
- back()
- insert()
- capacity()
- size()
- push_back()
- pop_back()
- clear()

List -

- List supports a bidirectional, linear list.
- vector supports random access but a list can be accesed sequentially only.
- List can be accessed front to back or back to front.
- Syntax -

```
list <int> l1;
list <string> l2;
list <int> l2 {1,3,2};
```

function -

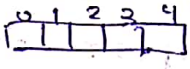
- sort() Eg - l2.sort() // 1,2,3
- size() // 3
- push_back(4) // 1,3,2,4
- push_front(3) // 3,1,3,2,4
- pop_back() // 3,1,3,2
- pop_front() // 1,3,2
- reverse() // 2,3,1
- clear() //
- remove(4) // remove all elements with value 4.

map -

- Maps are used to replicate associative arrays.
- Maps contain sorted key-value pair, in which each key is unique and cannot be changed, it can be inserted or deleted but cannot be altered.

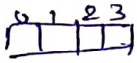
Numeric array -

Eg - int ar[5];

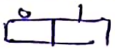


← index (int) from 0 onwards.

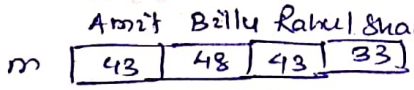
float b[4];



string c[2];



Associative Array -



m["Rahul"]; // 43

- Maps always arrange its keys in sorted order.
- Incase the keys are of string type, they are sorted in dictionary order.
- functions -
  - at() . [] . size() . empty() . insert() . clear()

String -

- **Traditional way :**
  - using null-terminated character arrays are not technically data types.
  - So, c++ operators cannot be applied to them.
  - Eg - `char s1[10], s2[10];`

Eg -

```
#include<iostream>
using namespace std;
int main() {
char s1[10] = "Hello";
 s1 = "Students"; // wrong
strcpy(s1, "Students");
 char s2[10];
 s2 = s1; // wrong
strcpy(s2,s1);
 s2>s1; // wrong
int i = strcmp(s2,s1);
 char s3[30];
 s3 = s1+s2; // wrong
strcpy(s3, strcat(s1,s2));
}
```

- **String class -**
  - The string class is a specialization of more general template class called basic_string.
  - Since defining a class in C++ is creating a new data type, string is derived data type.
  - This means operators can be overloaded for the class.

- **String is in STL -**
  - String is an another container class
  - To use string class, we have to include string header.
    `#include<string>`

- **Constructors -**
  - string class supports many constructors, some of them are - (constructor called)
    - string() Eg - string s1;
    - string(const char *str) Eg. string s1 = "Happy";
    - string(const string &str) Eg; string s2 = s1;

operators to be used -

=, +, +=, ==, !=, <, <=, >, >=, []

Date 10/10/2020

Mixed operations-

- We can mix string objects with another string object or cstyle string.
- C++ string can also be concatenated with character constant.
- Eg-

```
string s1 = "Hello";
char str[] = " students";
string s2;
s2 = s1 + str;
cout<< s2;   // Hello students.
s2 = s2 + 'A';
cout<< s2;   // Hello studentsA.
```

Useful methods-

- assign() Eg- string s1 = "Hello"; or s1.assign("Hello");
- append() Eg- s1.append(" students"); cout<<s1; → Hello students
- insert() Eg- s1 = "Hello"; s1.insert(2, "123"); cout<<s1; → He123llo
- replace()
- erase() (clear all)
- find()
- rfind() (reverse find)
- compare()
- c_str()
- size()

replace() → Eg. s1 = "Hello"; s1.replace(2, 2, "A"); // s1 → HeAo
(index, length)

find() → s1 = "Hello"; s1.replace(2, 3, "ravi"); // s1 → Hravi
Eg- s1 = "Hello Ravi Kumar"; s1.find("Ravi"); // 6

compare() → if s1 = s2 → 0
s1 > s2 → 1
s1 < s2 → -1

c_str() → s1 = "Hello";
char str[20];
strcpy(str, s1.c_str());

File handling -

seekg() - (for reading)

- Defined in istream class.
- Its prototype is
  - istream & seekg(streampos pos);
  - istream & seekg (streamoff off, ios_base::seekdir way);
- pos is new absolute position within the stream (relative to the beginning).
- off is offset value, relative to the way parameter.
- way values ios_base::beg, ios_base::cur and ios_base::end.

Eg-

a.txt

Hello Students mysirg
0 1 2 3 4 5 6 7 8 9 10 11 - - - -

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{  ifstream fin;  //inputstream object.
   fin.open("abc.txt");
   cout<<fin.tellg(); //returns 0 i.e. beg. of file.
   cout<<"\n"<<(char) fin.get(); // H
   cout<<"\n"<<fin.get(); // 101 (Ascii val)
   cout<<"\n"<<fin.tellg(); // 2
   fin.seekg(6); // set get pointer to 6th pos.
   cout<<"\n"<<fin.tellg(); // 6
   cout<<"\n"<<(char) fin.get(); // S.
   cout<<"\n"<<fin.tellg(); // 7
   fin.seekg(2, ios_base::cur);
   cout<<"\n"<<fin.tellg(); // 9
}
```

→ increment 2 to the current position of tellg()

seekp() - (for writing)

- defined in ostream class.
- rest is same as seekg.

```
main()
{
Eg- ofstream fout;
    fout.open("abc.txt", ios::ate | ios::app);
    cout<<fout.tellp(); // 0
    fout<<"ABCDEFG";
    cout<<fout.tellp(); // 7
    fout.close();
}
```

update — append — for random access in file.

[tellp is the output pointer].

→ a.txt → ABCDEFG

```
fout.seekp(2, ios_base::beg);
cout<<fout.tellp(); // 2
```