imp lang - P-x

* C++
  |
  Compilation + linking
  |
  executable (or library)

* C++ introduces two concepts :-

  - Object oriented programming
    |
    ex - name, email, verified
    - Ashish, ashish@gmail.com, true
    - AK, AK@gmail.com, false
  - Generic programming (in C++ it is known as Vectors)
    |
    ex. - 10, 12, 43
    - Ashish, Avinash, AK

* If we are ~~coding~~ creating an app. we will use C++ instead of Java or .net when it comes to optimizing etc.

* There are 63 keywords defined in C++ so far.

Refer P-24

#

```
#include <iostream>
using namespace std;
int main()
{
    cout << "hello \n new";
    return 0;
}
```

OR

```
#include <iostream>
int main()
{
    std::cout << "Hello World \n";
    return 0;
}
```

* Namespace is basically a group of code so instead of just having ~~one~~ all your code in one gicent bin, imagine splitting it up into diffent groups called namespaces.
* It prevents naming conflicts

Benifits of C++

* Reaching large audience with less work like in Windows, macos, iOS, Android.
* Build connected application
  - Database
  - Middle-tier
  - cloud
  - Internet of things
* design faster with live data.
* design faster with powerful component libraries.
  - drag & drop components
* Smart responsive designing
  - build your UI once & have run it on multiple devices

* To print Hello "There"

```
cout << "Hello \"there\" ";
```

or

```
cout << "Hello \"there ";
```

↓

this prints " Hello "there

*

```
cout << "\\";
```

prints \

#

Bool datatype (Boolean)

* It only contains True & false

```
{
  bool pizza_is_good = -1 or non-zero
  cout << pizza_is_good << endl;
}
```

output

: for -1 it is 1

* To print false or True

```
{ bool found = false;
  cout << std::boolalpha << found << endl;
}
```

output

- false

#

*

```
{
  const int x = 5
}
```

Value of x is never gonna to change inside the entire code

*

```
enum { y = 100 }
```

this will also not change

* cout << remainder(10, 3.25) << "\n"; → 0.25
cout << 10 % 3.25 ; → error
cout << 10 % 3 → 1
cout << fmod(10, 3.25) << "\n"; → 0.25.
cout << fmax (10, 3.5) << "\n"; → 10
cout << fmin (10, 3.5) << "\n"; → 3.5
cout << ceil (fmin(10, 3.25)); → 4
cout << floor (fmin (10, 3.25)); → 3
cout << trunc (fmin(10, 3.25)); → 3
cout << trunc (-1.5) → -1
cout << floor (-1.5) → -2
cout << ceil (-1.5) → -1
cout << round(-1.49) → -1

Search more on google "common mathematical functions C++".

• string

```
#include <string>
{ string greeting = "Hello";
} cout << greeting + " there" << endl;
```

Hello there

•
```
{
    string name;
    getline (cin, name);
    cout << name << '\n';
}
```

hello there

It will print the string after space while giving the input string.

- string greeting = "Hello";
  ~~greetings = "there";~~ greeting += "there";
  cout << greeting << endl; → Hello there

- string greeting = "Hello";
  greeting.append("there!"); → Hello there

- string greeting = "Hello";
  greeting.insert(3, " ");
  cout << greeting << endl; → Hel lo

- ~~string~~ greeting.insert(3, " ");
  greeting.erase(3, 1); → Hello

- greeting.insert(3, " ");
  greeting.erase(3); → Hel

- ~~greeting.insert(3, "");~~
  greeting.erase(greeting.length() - 1);
  → Hell

- greeting.pop_back(); → Hell

- greeting.pop_back();
  greeting.replace(0, 4, "Heaven"); → Heaven

- ~~greeting~~ string greeting = "what the hell!";
  greeting.replace(greeting.find("hell"), 4, "****");
  → what the ****?

string hi = "Hello U";
cout << hi.length(); → 7
cout << hi.size(); → 7

- string greeting = "what is up?";
  cout << greeting.substr(5, 2); → is

- cout << greeting.find_first_of("aiou"); → 2
- cout << greeting.find_first_of("!"); → 18446744073709551615

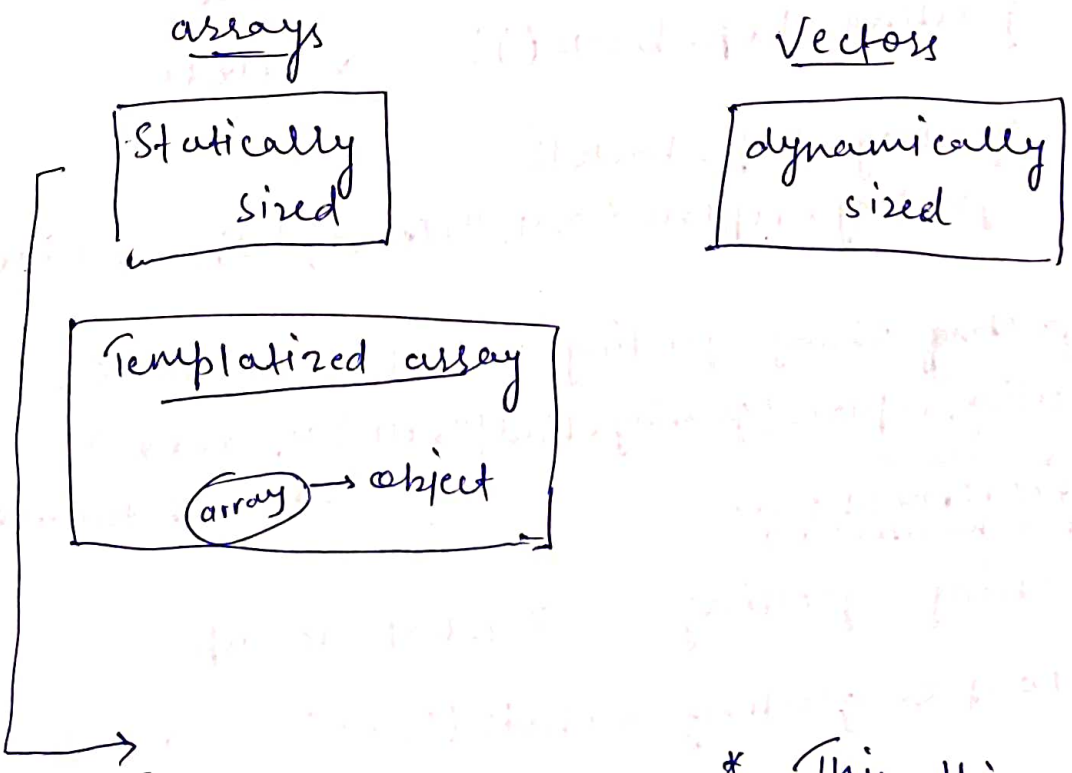
-1 = npos → not found

- `cout << greeting.find_first_of("!");`
  `if(greeting.find_first_of("!")==-1) cout << "Not found" << endl;` → Not found
- `if(greeting == "what is up?") cout << "equals" << endl;`
  → equals
- `if(greeting.compare("what is up?")==0) cout "equals";`
  → equals

Control flow

↳ Branching → if, switch
↳ looping → while, for, do-while

Arrays & Vectors



* They don't have any knowledge of size

* This things know its size meaning how many elements it has

Vectors

```
Vector <int> items = {12, 13}
items.push-back(100)
items[2]
items.size()  → return 3 (i.e, 12, 13, 100)
items[items.size() - 1]
```

↑ grab the last item inside the vector

```
items.pop_back();
```

Simple code start with

```
#include <vector>
{
Vector<int> data;   or  vector<int> data = {1,2,3};
items.push-back()  → any variable or number
```

This push the no. or variable to the store in data

```
data.size()
}
```

↘ Show the over all data on screen

```
Vector<int> &data
```

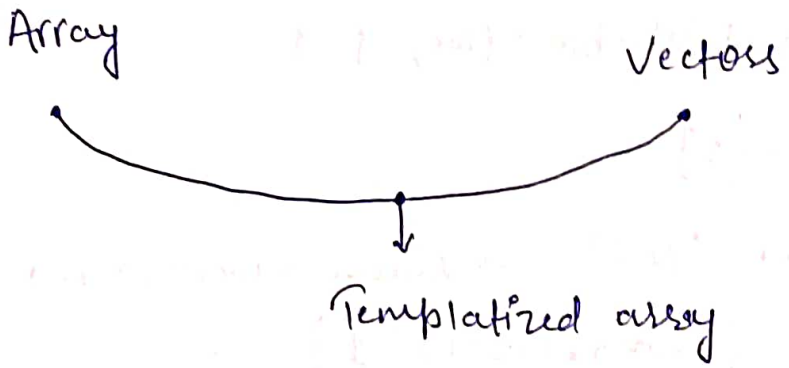
↓

for call by reference

out put

1st call 1, 2, 3, *
2nd call 1, 2, 3, * *
3rd call 1, 2, 3, * * *

STL (Standard template library) arrays



* It is in the middle of both the arrays & vectors.
* It is statically sized
* They remmembers the length while array don't.

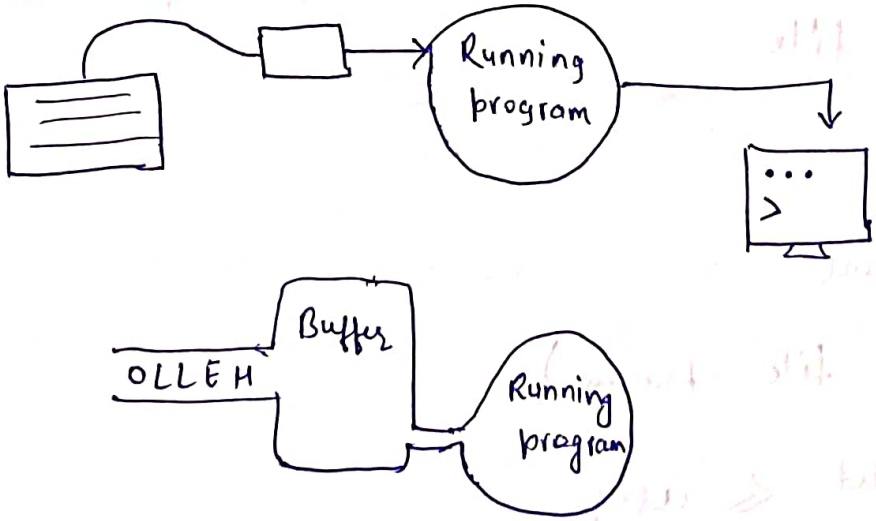
— array is passed as pointer while vectors are passed by value meaning they are copied

* Similarly templatized array is passed by value meaning they are copied

```
{
    array<int, 5> ages
        = {1, 2, 3};

    ages[0]

    ages.size()

    \ return - 3
```

Array	STL array	STL Vectors
* static	Static	dynamic
* decay to pointer	pass by value	pass by value
* must pass around size	• size()	• size
* X	assign to other variables	assign to other variable
<	<	<

Intro to IO Streams (Input output)



* When we input something in the keyboard it doesn't go directly to the Running program it goes first to the buffer & then it sends to the running program.

* istream (input stream) → cin

ifstream (input file stream) → Input

* input is same as cin but it is associate with some file

input >> Variable

* Works same as cin but data coming from file

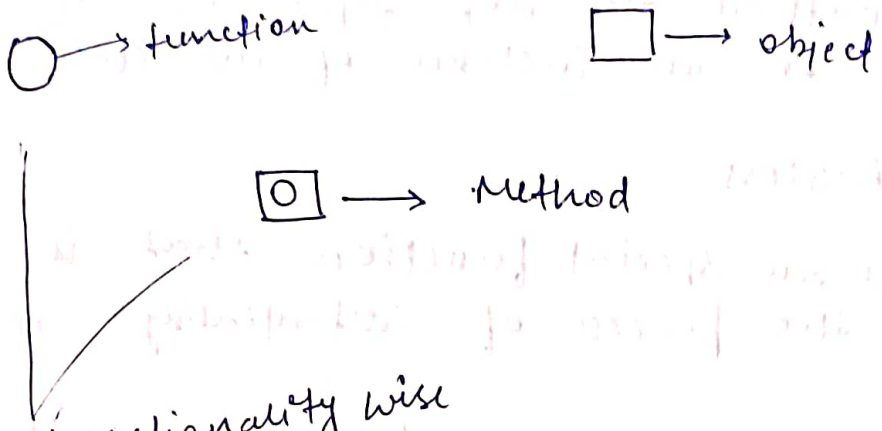
* Ofstream (output file stream)

output << user

Functions and constructors

1.) functions
2.) Method
3.) Static. Method
4.) constructor

> oop



* user , user ,
  user . do_something ()
  do_ something

* String tacos = "yum"
  tacos . length ()

  ↘ It is a function but attached to an object

- A method is attached to an object
- A static method is attached to a class

- we create class which is blueprints of how to make objects

- create instance of class which is known as object

- User user

  class object - instance of class

  ↑ ↑

  Static method Method

  User count () user . speak ()

- Static method are not dependent of specific entity or an instance of the class

* Constructors

- It is an special function that is called in the process of instantiating a class
- when we create a class we create a special function called constructor

ctor

- diffrence b/w normal function & constructor is that it doesn't have return type.

Multidimensional array

```
int grade [][3] =
    {
        (1, 2, 3);
        (4, 5, 6),
        (7, 8, 9)
    };
```

const Modifier

when we use function it can modify our data

```
main()
{
    int data [] = { 1, 2, 3 }
    Print_arr (data, 3)
    cout << data [0] << endl;
    return 0;
}
```

—— eg ①

```
void Print_arr (int data[] , int size)
{   for(int i=0; i<size ; i++)
    {   data[i]++;
        cout<<data[i];
    }
}
```

output :- 2, 3, 4

2

↙

Its still 2 instead of 1 from eq-①

* so we use const int i=0 so the data will not modify.

Pass by Value & Refrence

*
```
Void do(int x)
{   x=13;
}
int main()
{   int x=5;
    do(x);
}
```

- Value of x will not effect the main function variable x

↖ Pass by Value

*
```
void do(int &x)
{   x=11;
}
int main()
{   int x=5
    do(x);
}
```

↙ Pass by refrence

Here the value of x will change from 5 to 11 because we are passing address.

Function overloading

* void do_this (int &x)
  void do_this (string &x)

```
int main()
  {
     do_this ("hi");
  }
```

this will go to the string version

* The concept of having a function is unique basically it says the valid overload that is known as Method signature

Default arguments

```
double pow (double base, int pow = 2)
{ int total = 1
   for (int i=0 ; i<pow ; i++)
       { total *= base;
       }
       return total;
}
int main ()
  { cout << pow (3);
  }
```

default argument

output : 27

Multifile compilation

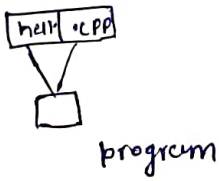
When you get larger programs, it will be good to break this file into multiple files.

usually break in 3 files

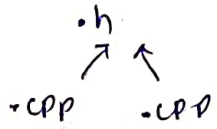
.h 1.) header file - declaration

.cpp 2.) implementation file - definition

.cpp 3.) main file - calling



include "filename.h"



To redeclare the cpp files again & again we use preprocessor directives.

~~# ifndef~~ ~~if no~~

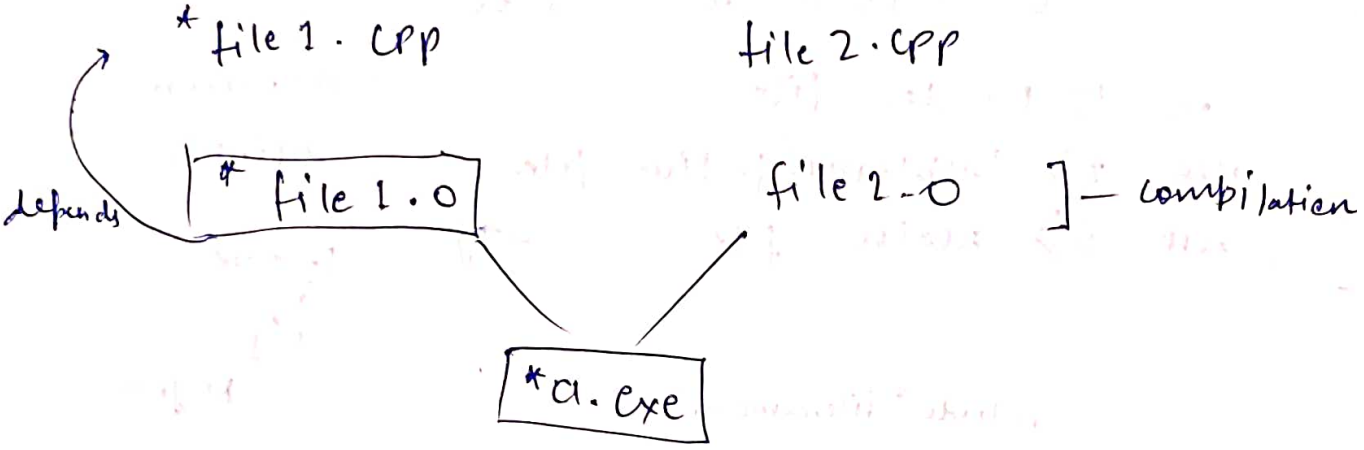
```
# ifndef CUSTOM        if not define
# define CUSTOM


# endif
```

To run multi file we use

```
g++ file1.cpp file2.cpp
```

Makefile

It's use to minimize the amount of ~~compilation~~ complication that is nessary



Makefile
makefile. or

In terminal
we will use
make
this will generate .o file & then
./filename
run the program

Namespace

namespace help in organazation of large programs. std is namespace in which the entire standard c++ library is declared

```
namespace unit
{

}
```

When we want to use namespace

1.) common functionality
2.) A particular company/project

3.) classifying / categories

```
namespace Unit
{
    double multiply()
}
Unit :: multiply(1,2);
```

Function Templates

```
template <typename T>
```

↓ or some uses class

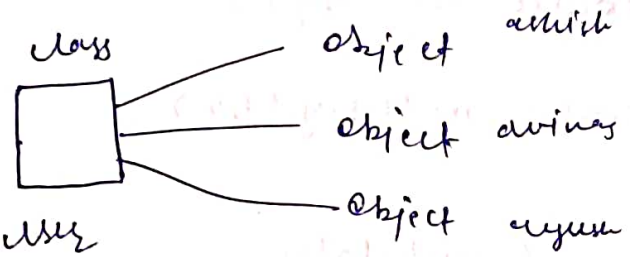
```
void swap(T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

whatever we learned in function overloading instead of creating two function which will accept the string & integer respectively we declare a template & create a function which will accept the both string & int type

P.t.o

Object Oriented Programming

OOP is based on class & object



class

ID	first	last	- - - -
01	ashish	kumar	
02	Caleb	Curry	

object

Struct

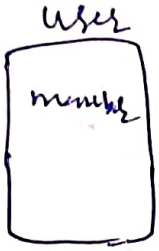
* In C++ Struct & class are identical

although

* Struct is used for small programs
* class is used for larger programs
* Struct can easily be converted to class

* Access Modifiers

Public
Private
Protected → used in inheritence



→ user 1

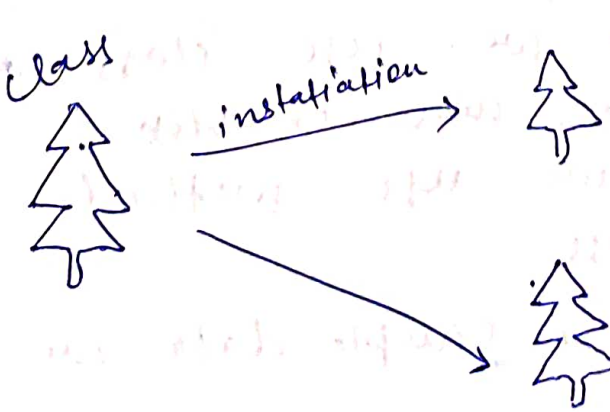
→ Public

- In structs members are by default Public
- " classes " " " " " Private

Struct — class → in c++

Struct ——————— class → in other language lot of difference

class & objects



The process of going from a class to object is known as ~~instantiation~~ instantiation

type - User user 1

Users

User user 2

user 2
int x
string y

these are type

User user 1

class

User ()
Speak ()
jump ()
string name

Method

s == s

user 1 == user 2

↑ overload

we can also now +

- By conventional we use class when things are more complex
- so when we use methods we use class
- if it just a simple data we use struct

Constructors

It is the exact same name of the class
It has no return or return type

* User usr1
  ↕
  User ()
  {
    // code
  }

default constructor is a with no arguments

↑
Default constructor or Implicity define construct

* custom constructor or explicity " "

↓

User usr1 ("caleb", "ashish");

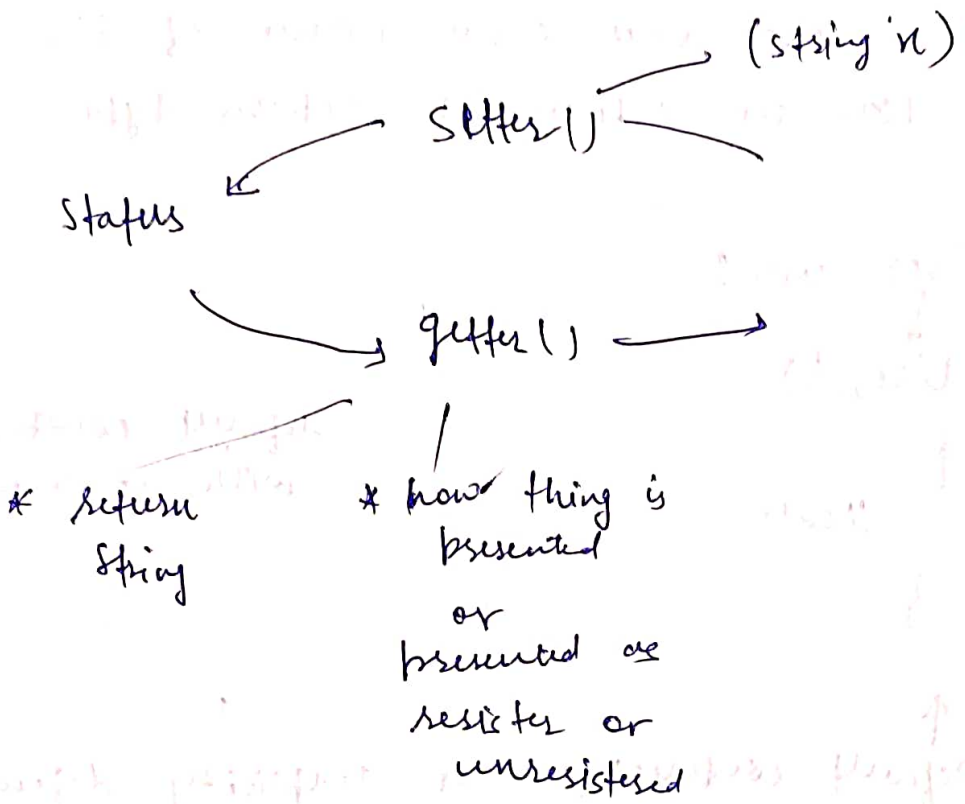
User (string f, string l)
{
  first_name = f;
  last_name = l;
}

* constructor is a special method that is automatically called when an object of a class is created

Destructor

~ User ()
{
  // code
}

Encapsulation



Static Data members

It is associated with class but not with objects

They can be both public & private

It is use to just count the members

count++; or count--;

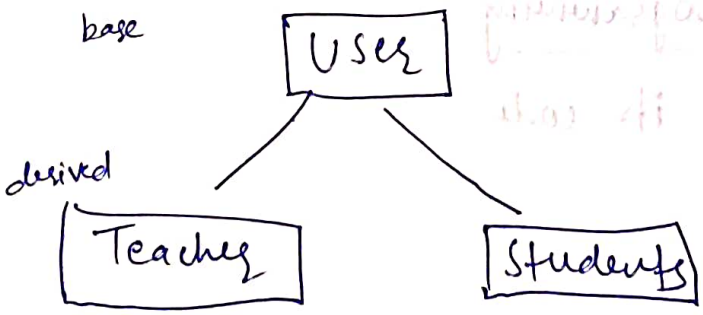
operator overloading

a + b

```
Point operator + (Point pos)
{
    Point point;
    point.x = x + pos.x
    Point.y = y + pos.y
    return point;
}
```

Inheritance & Polymorphism

We can one class inherit the memembers of other class




```
class Teacher : public User
{
}
Teacher teacher;
teacher.first_name;
```

We can treat a user as Teacher or a teacher as user which is known as polymorphism

a class can derived from multiple classes is known as multipolymorphism

*

Book

- a program can be organised in one of two ways.

① - around its code (what is happening)
② - around its data (who is being affected)

Structured programming

- around its code

OOPs

- around its data

- Modern c++ readers do not use .h extension

* cout << fixed << setprecision(x);
  #include <iomanip>

* #include <sstream>
  string stream ss("any")
  ss >> a >> ch;
  cout << ss.str();
  string stream can be used for both read string & write string data into string.

* set<int> s create set of integer
  s.insert(x);
  s.size();

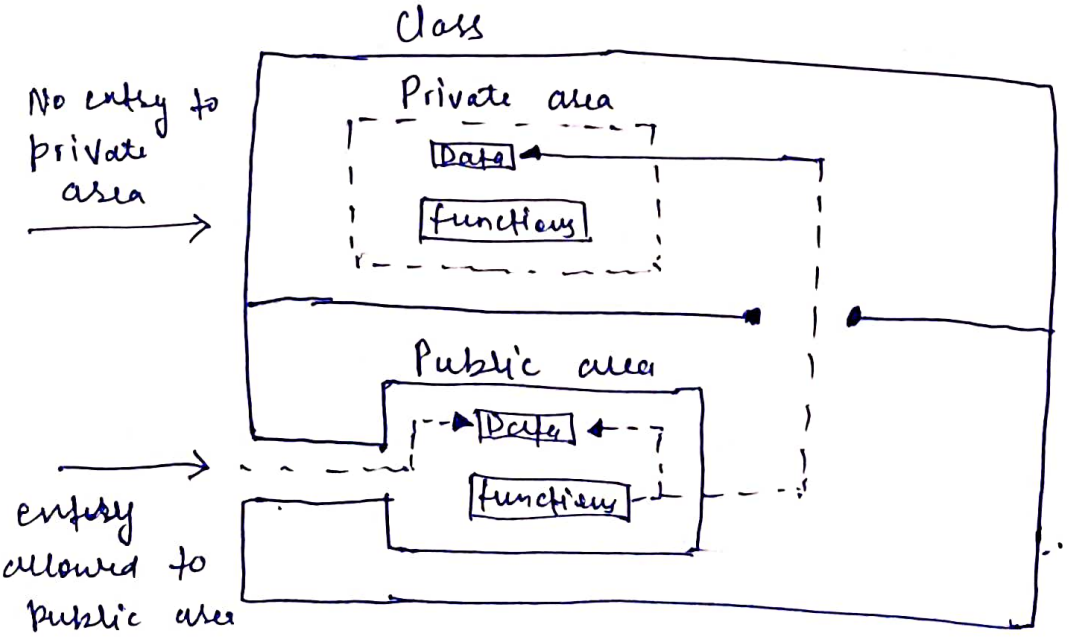
* Sorting
  sort(v.begin(), v.end());
  sort(v.begin(), v.end(), greater<int>());
  descending order

OOP

* Procedural programming is about writing procedures as functions that perform operations on data while OOP is about creating objects that contain both data & functions.
* OOP is faster & easier to execute.

Classes & Objects

```
Class class_name
{
    Private :
        Variable declaration
        function declaration
    Public :
        variable declaration
        function declaration
};
```



* When a variable is declared within a class they are called attributes.
* Methods are functions that belongs to the class

Defining member function

① - Outside the class definition
② - Inside the class definition

① - Inside the class definition

```
class name {
    Public:
    void mymethod ()
    {
        cout << " Hi ";
    }
};
int main ()
{
    name myobj;
    myobj.mymethod ();
    return 0;
}
```

② Outside the class definition

To define the outside function, we have to declare it inside the class & then define it outside of the class. This is done by specifying the name of the class, followed the scope resolution (::) operator, followed by the name of the function.

(::) → scope resolution operator

* these return type is void.

*
```
class myclass
{
    Public:
        void method();          ⟶ member function
};
void myclass :: mymethod()
{
    cout << "Hi";
}
int main()
{
    myclass obj;
    obj.mymethod();
    return 0;
}
```
(we can also add parameters)

* Several different classes can use the same function

* A non-member function cannot access the Private data of the class. (an exception of this rule is friend function discussed later).

* A member function can call another member function directly without using the dot (.) operator. called nesting of member function

* A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using (.) dot operator.

* • member function inside the class behaves as an inline function
  • member function outside the class can be made inlined just by adding qualifier inline in the header

ex- inline void item :: getdata (int a, float b)
{
}

Static Member functions

like static ~~function~~ ~~data~~ ~~access~~ ~~to~~ member variable, we can also have static member functions.

* A static function can have access to only other static members (function or variable) declared in the same class.
* A static member function can be called using the class name as follows

class-name :: function-name;

Friendly function

* C++ allows the common function to be made friendly with both classes, thereby allowing the function to have access to the private data of these elements. Such function need not be a member of any of these classes.

*- To make outside function friendly

```
class ABC
{
  ---
  Public
    friend void xyz(void);
};
```

* It can be declared either in the public or private part of a class without affecting its meaning.
* usually it has the objects as arguments.
* they are often used in operator overloading.

pointer to members

* a class member pointer can be declared using the operator (::*) with the class name.

```
class A
{   private:
        int m;
    public:
        void show();
};
```

->* — dereferencing operator

* int A::*ip = &A::m;

A::* — pointer-to-member of A class

&A::m — address of the m member of A class

* the pointer ip can now be used to access the member m inside member function (or friend function)

if a is object

cout << a.*ip; or cout << a.m;

ap = &a

cout << ap -> *ip; or cout << ap->m;

Constructor & Destructors

* a variable of built in type goes out of scope the compiler automatically destroys the variable
* But that's not if happend with objects thats constructor and destructor came into picture

Constructor

* A constructor is a special member function, whose task is to initialize the objects of its class.
* Constructor name is same as the class name.
* constructor is ~~created~~ invoked whenever an object of its associated class is created
*

```
class integer
{
        int m, n;
    Public:
        integer ();   //Contructor declared
};
integer :: integer ().   // cons. defined
{   m = 0;
    n = 0;
}
integer int1;   // object created
```

* the declaration (constructor) integer int1; not only creates the object int1 of type integer but also invoke the constructor function

* whenever a class contains constructor it is guaranteed that an object created by the class will be initialized automatically.

* A constructor that accepts no parameters is called the default constructor.

default constructor for class A is

Class A is A::A()

* They should be declared in Public section

* They don't have return type but they can have default argument

* Parameterized constructors

- The constructor initializes the data members of the objects to zero.
- The constructor integer() may be modified to take arguments.

```
Class integer
{
    int m, n;
  Public:
    integer(int x, int y); // parameterized
                           constructor
```

```
};

integer :: integer(int x, int y)
{
    m = x; n = y;
}
```

- When a constructor has been parameterized the object declaration statement such as integer int1; may not work ~~it can be done~~ we must pass the initial values as argument to the constructor function which can be done by
  - by calling the constructor explicitly
  - by calling " " implicitly

* - by explicitly

```
integer int1 = integer(0, 100);
```

- by implicitly (or shorthand method)

```
integer int(0, 100);
```

- a constructor can accept a reference of its own class as a parameter.

```
class A
{ ....
  Public
};  A(A&);
```

The constructor is called copy constructor

* Multiple constructors

```
class integer
{
    int m, n;
  Public:
    integer()
      { m=0; n=0; }
    integer (int a, int b)
      { m=a; n=b; }
    integer (integer &i)
      { m = i.m; n = i.n; }
};
```

- integer I1;

would automatically invoke the first constructor & set both m & n of I1 to zero

The first constructor which takes no argument, is used to create objects which are not initialized

- integer I2(10,20);
- integer I3(I2);

It would invoke the 3rd const. which copies the value of I2 into I3 (copy constructor)

* Constructor with Default arguments

- Complex (float real, float img = 0.0);

  complex (5.0);

  assign 5.0 to real & 0.0 to img.

  complex (2.0, 3.0);

  assign 2.0 to real & 3.0 to img.

- A::() → default constructor

  A::(int=0) → default argument constructor

When default argument constructor is called with no argument it becomes a default constructor.

- ### Destructors

* It is used to destroys the objects that have been created by a constructor.
* It never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up the storage.

Operator overloading & Type conversions

* In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading

* we can overload all C++ operators except
  - class member access operators (.,.*)
  - scope resolution operator (::)
  - size operator (sizeof).
  - conditional operator (?:).

Defining operator overloading

* general form

```
return type classname :: operator op(arglist)
{
        function body
}
```

* operator op is function name

* Operator function must be either member function or friend function
  - for friend function - one argumen for unary
    2 for binary operators
  - for member function - no arg. for unary
    1 for binary oper

* prototype used will be ***

- Vector operator+(Vector); // vector addition
- Vector operator-(); // unary minus
- friend vector operator+(vector, vector); // vector addition
- friend vector operator-(vector); // unary minus
- vector operator-(vector &a); // subtration
- int operator==(Vector); // comparison
- friend int operator==(vector, Vector) // comparison

* Process of overloading

① - Create a class that defines the data type that is to be used in the overloading operation

② - Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function

③ - Define the operator function to implement the required operations.

* - Operator function can be invoked by

op x or x op

- for unary

x . op y

- for binary operators : objk(op nop)

Operator op(x)

- for friend function

. x . operator op(y)

* for member function

operator op(x, y)

in case of friend function

unary when both forms are declared.

* Statement S2 = -S1; will not work it will work if the function modified to return an object.

∴ -S; will work in such case

* Note that argument is passed by reference it will not work if passed by value because only a copy of the object that activated the call is passed to operator -()
∴ change made inside the operator function will not reflect in the called objects

• Overloading binary operators

* we know how to add two complex numbers using a friend function

c = sum(A, B); // functional notation

But it can be replaced by a natural looking expression

C = A + B // Arithmetic notation

by overload the + operator using an operator +() function

* we should know the features
  - It recieves only one complex type argument explicitly
  - It returns a complex type value
  - It is a member function of complex.

* Now
  - C3 = C1 + C2; // invokes operator+() fun
  - the object C1 takes the responsibility of invoking the function & C2 plays the important role of an argument that is passed to the function.
  - It is equivalent to

```
C3 = C1.operator+(C2);
```

- **overloading binary operators using friends**

* as we discussed friends functions requires 2 arguments to be explicitly passed to it while member function required only one.

* so now for friend.

① friend complex operator+(complex, complex);
② complex operator+(complex a, complex b)
{
return complex((a.x+b.x),(a.y+b.y));

* In this case

C3 = C1 + C2; is equivalent to
C3 = operator+(C1, C2);

* we will get the result from both

member or friend function but there is a situation like

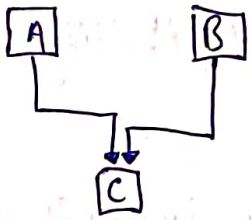
A = 2 + B; or (A = 2 * B) where we will have to use friend because here built-in-function i.e 2 is used to invoke the function which is not possible here because of built-in-function but this is possible in case of friend function

* 2 - here is should be an objects for using the member function.

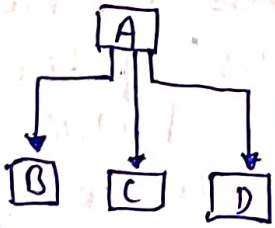
Inheritance

* The mechanism of deriving a new class from an old one is called inheritance.
  - old class → base class
  - new one → derived class or subclass
* A class can also inherit properties from more than one class or from more than one level
*


Single inheritance

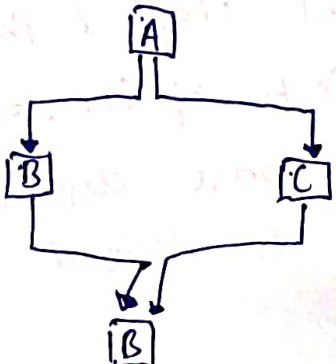


Multiple inheritance



Hierarchical inheritance

multilevel inheritance



Hybrid inheritance

* class derived-class-name : visibility-mode base-class-name

```
{
    ---//
    ---//    members of derived class
};
```

* The colon (:) indicates that the derived-class-name is derived from the base-class-name. The visibility-mode is optional & if present, may be either private or public. But the default visibility-mode is private

- when base class is privately inherited by derived class, 'public members' of base class become 'private members' of the derived class & therefore the public members of the base class can only be accessed by the member function of the derived class. they are inaccessible to the objects of derived class. no members of the base class is accessible to the objects of derived class
- when the base class is publically inherited, 'public members' of the base class become 'public members' of the derived class & therefore they are accessible to the objects of the derived class. The private members of a base class will never become the members of its derived class

* In inheritance, some of the data elements & member function are 'inherited' into the derived class.

* Although the data member 'a' is private in B & cannot be inherited objects of D are able to access it through an inherited member function of B

* A member declared as protected is accessible by the member functions within its class & any class immediately derived from it & it cannot be accessed by the function outside these two classes.
  - When a protected is inherited in public mode it becomes protected in the derived class too & therefore is accessible by the member function of the derived class.
  - When a protected is inherited in private mode it becomes private in the derived class although, it is available to the member function of the derived class.

* So the various function which have the access control to the private & protected member of a class are.
  - A function that is a friend of the class
  - A member function of a class that is friend of the class.
  - A member function of a derived class

- Multilevel inheritance

* A serves as a base class for derived class B which in turn serves as a base class for the derived class C. The class B is known as intermediate base class.

* class A {...};
  class B : public A {...};
  class C : class B {...};

- Multiple inheritance

* A class can inherit the attributes of two or more classes. This is known as multiple inheritance. It allows us to combine the features of several existing classes as a starting point for defining a new classes

* class D : visibility mode B-1, visibility B-2
  {

  .... (Body of D)
  };

* If multiple display() present in 3 class so in objects

```
B b;
b.display();
b.A::display();
b.B::display();
```

Hierarchical inheritance

* The base class will include all the features that are common to the subclass A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes & so on.

Hybrid inheritance

* Assume that we have to give weightage for sports before finilising the result. The weightage for sports is stored in a separate class called sports.

Virtual base classes

* Consider a situation where all three kinds of inheritance, namely multilevel, multiple & hierarchical, are involved. The child has two direct base classes 'Parent1' & 'Parent2' which themselves have a common base class 'grandparent'. The child inherits the traits of traits of grandparent via two seperate paths. The grand parent is sometimes referred to as indirect base class.

* A problem as all the public & protected members of 'grandparent' are inherited into 'child twice', first via 'parent1' & again via 'parent2'. This means child would have duplicate sets of the members inherited from grandparent.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes.

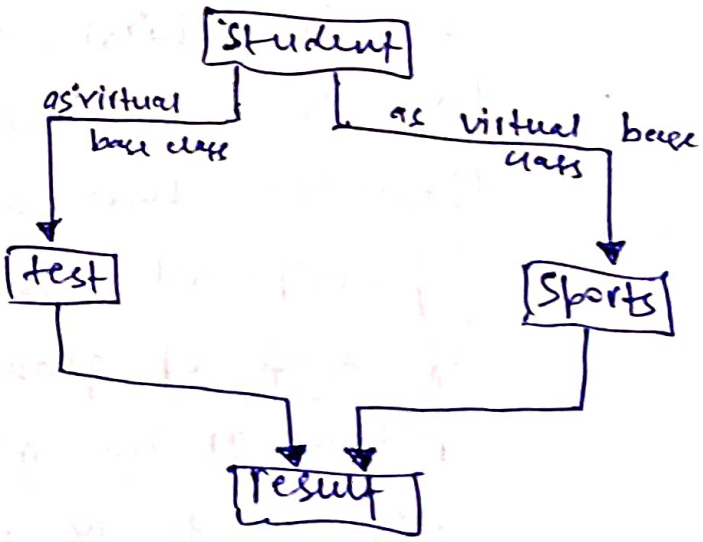
* When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between the virtual base class & a derived class

> note - The keyword virtual & public may be used in either order

*

```
Class A    (// grandparent)
{
};
Class B1 : virtual public A
                    (// parent1)
{
};
class B2 : public virtual A
                    (// Parent 2)
{
};
class C : public B1, public B2
                              (//child)
{       // only 1 copy of A
};      // will be inherited
```

*



Constructors in derived class

* As long as no base class constructors takes any argument, the derived class need not have a constructor function, However if any class contains constructor with one or more arguments, then its is mandatory for the derived class to have a constructor & pass the arguments to the base class constructors
* when both base class & derived class contain constructor the base constructor is executed first & then the constructor in the derived class is executed.
* In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. same is in case with multilevel inheritance.
* The constructor of derived class receives the entire list of values as its arguments & passes them to the base constructor in the order in which they are declared in the derived class. The base constructor are called & executed before executing the statements in the body of the derived constructor.
* The header line of derived ~~class~~ constructor function contains 2 parts seperated by a colon (:). The first part provides the declaration of the arguments they are

passed to the derived class constructor & the second part lists the function calls to the base constructors.

* The constructor for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed.

* Another method

- C++ supports another method of initializing the class objects.

```
constructor (arglist) : initialization-section
{
    assignment-section
}
```

- The assignment-section is nothing but the body of the constructor function & is used to assign initial values to its data members.
- The initialization-section is used to provide initial values to the base constructor & also to initialize its own class members
- we can use both section to initilize the data members of the constructor class.

```
class XYZ
{ int a, b;
  Public:
     XYZ(int i, int j): a(i), b(2*j) {}
};
main()
  { XYZ x(2,3);
  }
```

a = 2
b = 6

```
XYZ(int i, int j): a(i), b(a*j) {}
```

a = 2 , b = 6

a is declared first is initialized first & its value is used to initialize b.

```
XYZ(int i, int j): b(i), a(b*j) {}
```

is <u>not valid</u>.

because the value of b is not available to 'a' which is to be initialize first.

Polymorphism

* It simply means 'one name, multiple forms'

*
```
Class A
{   int x;
    public
    void show()          // show() in base class
      {}
};
Class B : public A
 {   int y;
     public :
       void show()       // show() in derived class
         {}
 };
```

How do we use the member function show() to print the values of objects of both the class A & B?

* It would be nice if the appropriate member function could be selected while the program is running. This is known as run time polymorphism. C++ supports a mechanism known as virtual function to achieve run time polymorphism

Pointers

We know that a pointer is a derived data type that refers to another data variable by storing the variable memory address rather than data.

```
data-type * pointer-variable
```

* int *ptr, a;
  ptr = &a

* The pointer which is not initialised in the program are called Null pointers.

* Manipulation of pointers

We can manipulate a pointer with the indirection operator, i.e '*' which is also known as defrence operator. Using the defrence operator we can change the contents of the memory location

* Pointer to function

- pointer to function is known as callback function, we can use these func pointer to refer to a function.
- C++ provide 2 types of function pointers
  - function pointer point to static member
  - " " " " non-static member / these requires hidden argument
- data_type (* function_name)();

* pointer to objects

  int x;

  by pointer

  item *it_ptr; = &x;

We can refer to the member function of item in 2 ways, one by using dot operator & the object and another method by using the arrow operator & object pointer

```
n.getdata(100,75,50);
n.show();
```

OR

```
ptr -> getdat(100,75.50)
btr -> show();
```

* this pointer (->)

c++ uses a unique keyword called this to represent an object that invokes member function this is a pointer that points to the object for which this function was called

```
class ABC
  {
    int a;
  };
```

a is private

```
a = 123
```

we can also use.

```
this ->a = 123;
```

- Virtual functions

* polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes.

* we use the pointer to base class to refer to all the derived objects. but a base pointer even, when it is made to contain the address of a derived class always executes the function in the base class. A compiler simply ignores the content of the pointer & chooses the member function. that matches the type of the pointer.
* when we use same function name in both the base & derived classes the function in base class is declared as virtual using the keyword virtual preceding its normal declaration
* when a function is made virtual C++, determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.
* we must access virtual functions through the use of a pointer declared as a pointer to the base class

Working with files

* The I/O system of C++ handles file operations which are very much similar to the console input & output operations.
  - The stream that supplies data to the program is known as input stream
  - The stream that recieves data from the program is known as output stream

* The I/O system of C++ contains a set of classes that defines the file handling method these include ifstream, ofstream, & fstream they are derivied from fstreambase & from the corresponding iostream class. these classes are designed to manage the disk files

Opening & closing a file

* For opening a file, we must create a file stream & then link it to the file name. A file stream can be defined using the classes ifstream, ofstream. & fstream that are contained in the header file fstream.

* A file can be opend in 2 ways

(1)- Using constructor function of the class
  - It is useful when we use only one file in the stream

(2)- Using member function open() of the class

- It is useful when we want to manage multiple files using one stream.

① - Opening file using constructor

Here, filename is used to initialize the file stream object.

- Create a file stream to manage the stream using the appropriate class that is to say, the class ofstream is used to create the output stream & the class ifstream to create input stream.
- Initialize the file object with the desired filename

ex - ofstream outfile("result"); // output only

- This creates outfile as an ofstream object that manages the output stream. This object can be any valid C++ name such as o-file, myfile or fout. this statement also opens the file result & attaches it to the output stream outfile.
- Similarly, the following statement declares infile as an ifstream object & attaches it to the file data for reading (input)

```
ifstream infile("data"); // input only
```

The program may contain statement like this

```
outfile << "Total";
outfile << sum;
infile >> number;
infile >> string
```

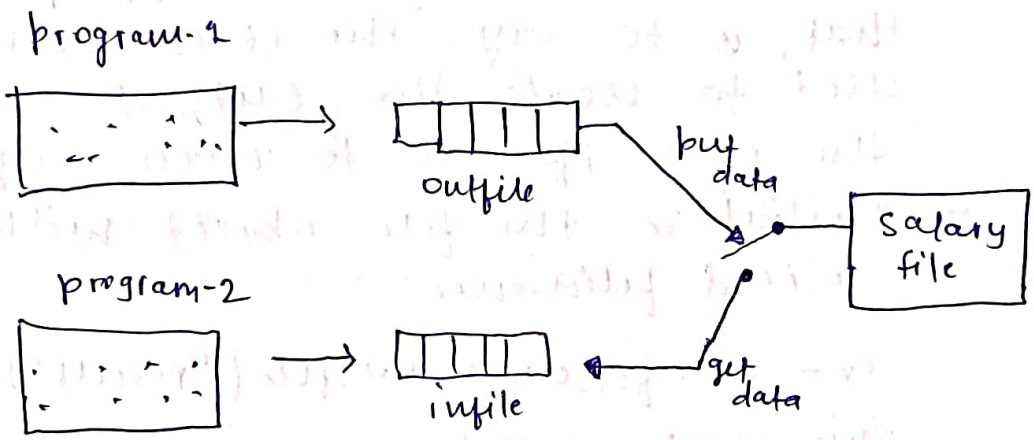
- We can use same file for both reading & writing

- program 1

```
ofstream outfile("salary");        // create outfile
                                   & connects
                                   "salary" to it
```

program 2

```
ifstream infile("salary");         // creat infile
                                   ()
```



The connection with a file is closed automatically when the stream object expires (when program terminate).

(2)- Opening file using open()

the function open() can be used to open multiple files that use the same stream object.

```
file-stream-class stream-object;
stream-object.open("filename");
```

ex- ofstream outfile;
outfile.open("Data1");
outfile.close();
outfile.open("Data2");
outfile.close();

* eof is a member function of ios class. It returns non-zero value if the end-of-file(EOF) condition is encountered & zero otherwise

*
stream-object.open("filename", mode);

the second argument is specifying the file mode

* other parameter are like ios::app & ios::ate take us to the end of file when it is opened. The difference b/w two is that ios::app allows us to add data to the end of the file only while ios::ate mode permit us to add data or to modify the existing data anywhere in the file

* Each file has associated pointers known as file pointers. one of them is called the input pointer(get pointer) which is used for reading the contents of a given file location and other is called output pointer (put pointer) which is used for writing to a given file location.

*
- seekg() - moves get pointer(input) to a specified location
- seekp() - moves put pointer (output) to a specified location
- tellg() - gives the current position of the get pointer
- tellp() - gives the current position of the put pointer

ex-

infile.seekg(10);

moves the file pointer to the byte no 10. The byte in file starts from 0 so pointing to 11th byte of the file

```
of stream fileout;
fileout.open("hello", ios::app);
int p = fileout.tellp()
```

on execution, the output pointer is moved to the end of the file "hello" & the value of p will represent the number of bytes in the file.

- **put() & get() functions**

* the function put() writes single character to the associated stream
* the function get() reads a single character from the associated stream

- **write() & read() function**

The functions write() & read(), unlike the functions put() & get(), handle the data in binary form. This means that the value stored in the disk file is the same format in which they are stored in the internal memory

- **Errors handling during file operations**

① - A file we are attempting to open for reading does not exist

② - The file name used for a new file may already exists

③ - we may attempt an invalid operation such as reading past the end-of-file

(4). there may not be any space in the disk for storing more data.

(5) we may use invalid file name

(6). we may attempt to perform an operation when the file is not opened for that purpose.

* The C++ file stream inherits the 'stream-state' member from the class ios. This member records information on the status of a file that is being currently used. The stream state member uses bit fields to store the status of the error condition stated above

- eof() - return true - if e.o.f. encountered
- fail() - return true - when I/O operation failed
- bad() - return true - if invalid oper. attempted
- good() - return true if no error has occured

Command line arguments

* like C, C++ too supports a feature that facilitates the supply of arguments to the main(). function. These arguments are supplied at the time of invoking the program.

C > exam data result

Here exam is the name of the file containing the program to be executed. & result & data are filenames passed to the program

as command-line arguments.

* The first argument is always filename (command name) & contains the program to be executed
* main (int argc, char* argv[])
* The first argument is argc (known as argument counter) represents the number of arguments in the command line. The second arguments argv (known as argument vector) is an array of char type pointers that points to the command line arguments. The size of array will be = to value of argc
* C > exam data result.

the value of argc would be 3 & the argv would be an array of 3 pointers to strings.

argv[0] ---> exam
argv[1] ---> data
argv[2] ---> results.

argv[0]. always represent the command name that invokes the program. The characters pointers argv[1] & argv[2] can be used as file names in the file opening statements.

Templates

* The class template defination is very similar to an ordinary class defination except the prefix template <class T> & the use of T is as a type name name in declaration
  - T is may be substituted by any data type including the user defined datatypes.

    Ex - Vector <int> v1(10);
  - The T may represent a class name as well
    Vector <complex> v3(5);
* A class created from a class template is called a template class

  for an object of a template class.

  ```
  classname <type> objectname(arglist);
  ```

* The process of creating a specific class from a class template is called instantiation. The compiler will perform the error analysis only when an instantiation takes place. It is advisable to create & debug an ordinary class before converting it into a template.

- Class template with multiple parameters

```
templat < class T1, class T2, ...>
class classname
  { ....        (body of the class)
};
```

Function Templates

- like class templates, we can also define function template that could be used to create a family of functions with different argument types.

```
template <class T>
returntype functionname (arguments of type T)
{
    //...
    //body of function
    // with type T
    //    wherever appropriate
    //...
}
```

- The function template syntax is similar to that of the class Template except that we are defining function instead of classes.

ex -
```
void f(int m, int n, float a, float b)
{
    swap(m,n);
    swap(a,b);
}
```

Exception Handling

* There are two most common types of bugs are logic errors and syntatic errors.
The logic errors occurs due to poor understanding of the problem & solution procedure.
The syntatic errors arise due to poor understanding of language itself. we can detect these errors by using exhaustive debugging & testing procedures
* Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing
Anamolies might include conditions such as division by zero, access of an assay outside of its bound or running out of memory or disk space.

* Exceptions are of 2 Kinds, namely, Synchronous & asynchronous exceptions.

- Synchronous.
  - out-of-range
  - overflow

Asynchronous
- Keyboard interrupts (as they are the errors caused by events beyond the control of program)

* C++ exception handling mechanism is basically built upon 3 keyboards, namely try, throw and catch

- try - the keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions.
- throw - when exception is detected, it is thrown using a throw statement in the try block.
- catch - the exception thrown by the throw statement in the try block & handles it appropriately.

```
...
try
{
    ...
    throw exception;
}
catch (type arg)          // catches exceptions
{
    ....
}
```

when try block throws an exception, the program control leaves the try block & enters the catch statement of the catch block

- **Multiple catch statements**

It is possible that a program segment has more than one condition to throw an exception.

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

- ## Rethrowing an Exception

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments.

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence & is caught by a catch statement listed after that enclosing try block

- ## Specifying exceptions

It is possible to restrict a function to throw only certain specified exceptions. this is

achieved by adding a throw list clause to the function definition.

```
type function(arg-list) throw(type-list)
{
    function body
}
```

Specifies type of exception that may be thrown

* throwing any other type of exception will cause abnormal program termination.
* If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty i.e we must use.

```
throw();
```

in function under line.