Operator overloading :-

$C3 = C1.add(C2)$

C1 ने add function को call किया
C2 as an argument pass हुआ

```
# include <iostream.h>
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;      → c1 का a
      temp.b = b+c.b;      → c2 का a
      return (temp);
  }
};
void main()
{
  complex c1, c2, c3;
  c1. set data (3,4);
  c2. set data (5,6);
  c3 = c1. add (c2);
  c3. show Data ();
}
```

→ complex operator + (complex c)
{
→ "
}

$C3 = C1 + C2$

→ C3 = C1. operator + (C2);
or
C3 = C1+C2;

$C3 = C1+C2$

$C3 = C1+C2$

(जिसमें उपयुक्त, कोई अलग से लिख सकते हैं)

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

• Any symbol can be used as function name
→ If it is a valid operator in c language
→ If it is preceded by operator keyword

operators come in three flavours

* unary - (++,--)
* Binary - (+,-)
* Terny - (?:)

→ you are free to overload most of the built-in oper
but you cannot create new operators of yo

→ operator are overloaded by creating operator fu

→ An operator function is created using the ke
operator.
member function of a class or
non-member function (mostly friend functio

Syntax - member operator function

→ for Binary operators

```
Retype class name operator # (arg-list)
                              ↳ place holder. ⇒ @
{
// operation
}.
```

⇒ operator function returns an object of the class they a

↳ while overloading binary operators using member
function, the arg-list will contain one parameter

friend RetType operator # (arg-list)
{
// operations
}

→ while overloading a binary operator using friend function, argument list must take 2 arguments. one of them must be of user defined type.

→ A friend operator function for unary operators...

friend RetType operator# (class name & obj)
{
// operations,
}.

→ while overloading a unary operator using friend function. argument list must have arguments, as reference to the object.

Operator overloading Rules

You cannot change any operators "precedence"
You cannot change system to use an operator
You cannot change predefined Job performed by any operator.
(i.e you cannot overload + operators in such a way to perform subtraction)
you can overload unary and Binary operators, but you cannot overload the ternary operator (?:)

Unary and unary operators can be overloaded by both Approaches.

(1) member function.
(2) friend function.

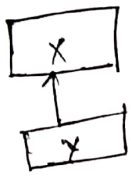
Inheritance :- It is a process of inheriting the properties and behaviours of existing class into a new class.

* Need of inheritance
  - → represent real-world relationships well.
  - → provides reusability
  - → supports transitivity.

→ A new class can be derived from an existing class. The new class derived class is called derived or subclass and existing class is called base class or super class.

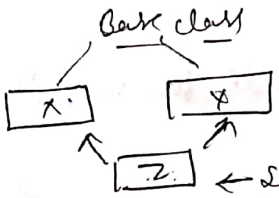
Types of inheritance

① Single inheritance :- when a derived class inherits only from one base class.



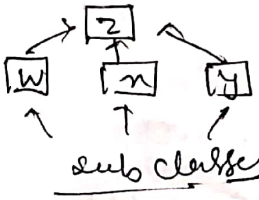

```
class A
{
};
class B : public A
{ };
```

② multiple inheritance :- when a subclass inherits from multiple base classes.



→
```
class x
{ };
class y
{ }
```
→ `class z : public x, public y`

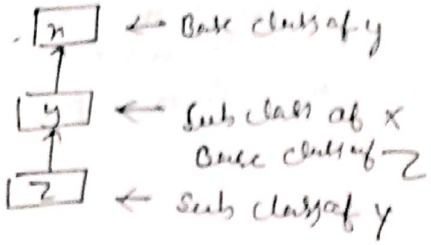
③ Hierarchical inheritance :- when many subclass inherits from a single base class.



→
```
class Z
{ };
class X : public Z
{ };
class W : public Z
{ };
class Y : public Z
```

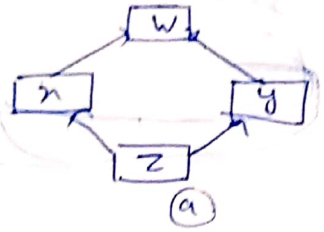
④ Multilevel Inheritance :- when a subclass inherits from a class that itself inherits from another class.

→ the transitive nature of inheritance is reflected by this form of inheritance.

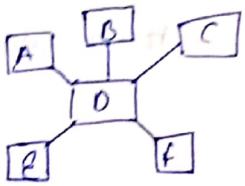



```
class A
{};
class B: public A
{};
class C: public B
{};
```

⑤ Hybrid inheritance :- Hybrid inheritance combines two or more form of inheritance..



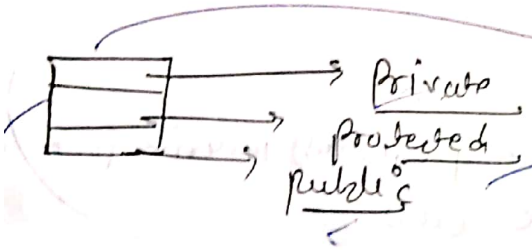
(a)



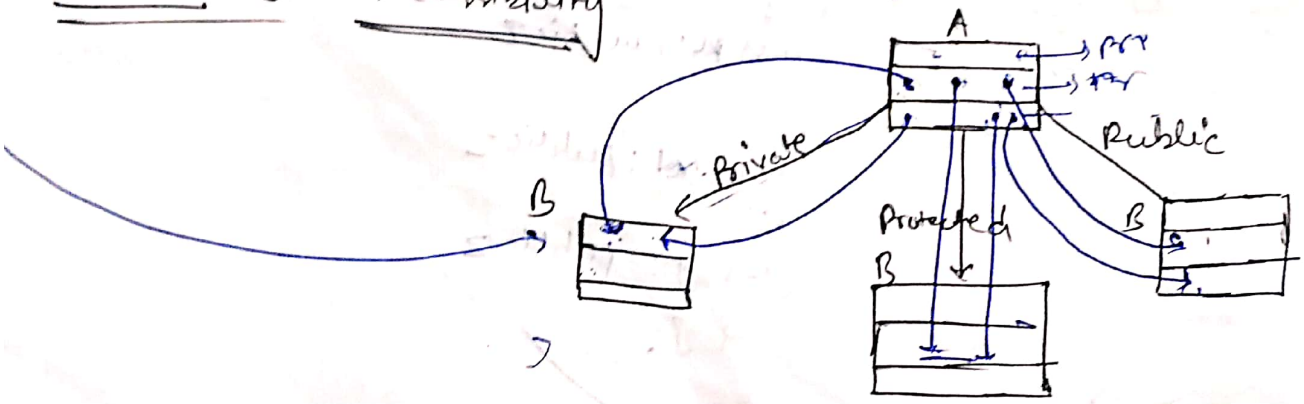
(b)

Visibility mode:- the visibility mode controls the visibility and availability of inherited base class members in the derived class.

→ If no visibility mode is specified, then by default the visibility-mode is considered as private.



→ Availability vs Accessibility



for ... 255, 255

→ private members of a base class are not directly accessible in the derived class. But that does not affect its visibility in the derived class.

→ the private members of the base class are visible in the derived but they are not directly accessible.

→ When a class is derived publicly, you cannot selectively deny access to base members by declaring them in the derived private section of the derived class.

```
class { public:
        int x,y,z;
      };
class Derived : public Base
{ public:
    int a;
  private:
    Base::x;   //Invalid ; Not Allowed
};
```

→ to allow access to some of the inherited members, you can selectively declare some of the base class members in public section of the privately derived class.

Abstract class → A class that serves only as a base class from which other classes are derived, but no objects of this base class type exist is known as Abstract class.

→ base class only serves for the derivation purpose. no object are declared for the base class.

Multiple inheritence → multiple inheritence allows a derived class to combine the strengths of other classes.

→ Note that when base classes are publicly inherited, protected and public members retain their access specifiers. i.e they remain protected and public respectively in the derived class.

→ The derived class can use all the inherited members in the same way it uses its own members.

→ Constructors in multiple Inheritance

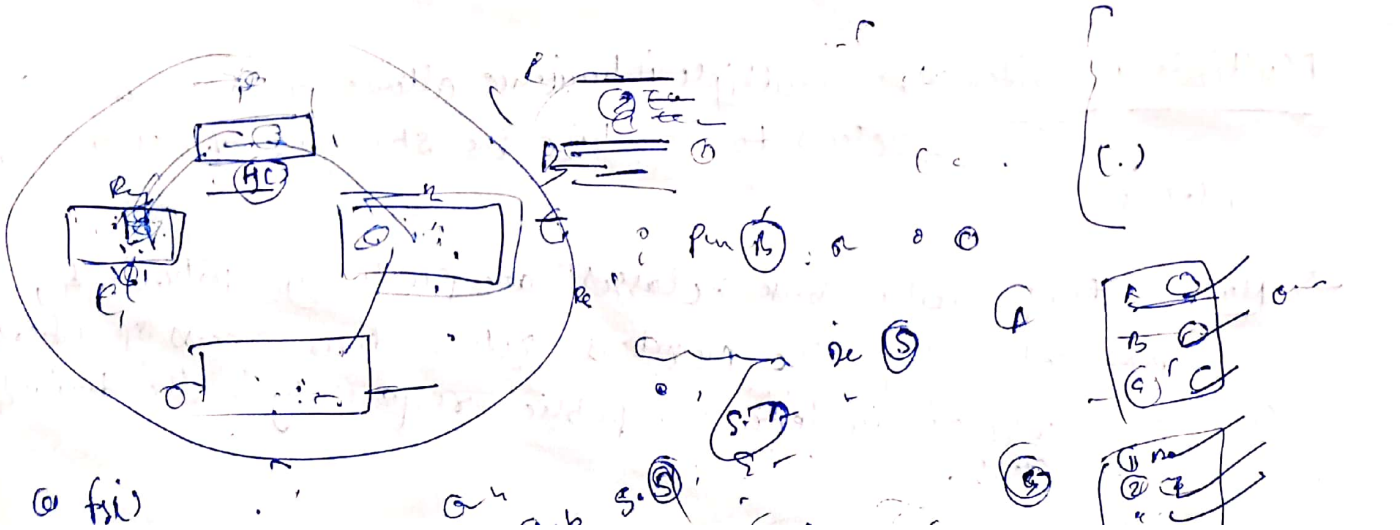
→ as long as no base class constructor takes any arguments, the derived class need not have a constructor function.

However if a base class contains a constructor with one or more arguments, then it is mandatory for derived class, to have a constructor and pass the arguments to the base class constructors - this is because constructors cannot be inherited though ; a derived class can explicitly call the constructors and destructors of the base class.

→ In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.

* Virtual Base class :- When two or more objects are derived from a common base class you can prevent multiple copies of base class from being present in an object derived from those objects by declaring the base class as virtual when it is inherited.

This is accomplished by putting the keyword virtual before the base class name. when it is inherited,



Virtual function — It is a function which is used to remove the problem of function overriding.

Base class pointer

- Base class pointer can point to the object of any of its derived class (child)
- But its converse is not true

```
class A
{
};
class B: public A
{};
void main()
{
  A *P, O1;
  B O2;
  P=&O2;
}
```

for ~~Virtual~~

Virtual function concept

```
#include<conio.h>
#include<iostream.h>
class A
{ public:
  virtual void f1(){}
};
class B: public A
{
public:
  void f1(){} # function overriding
  void f2(){}
};
void main()
{
  A *P, O1;
  B O2;
  P=&O2;
  P->f1();
}
```

for (late binding)

if virtual keyword is use in Base class then this pointer check content in f1 first

then type of f1

virtual function working concept

variable created by compiler itself

```
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){...}
};
```

vtable: f2 | f3 | f4 — static array created by compiler

vtable: f2 | f3 | f4

```
main()
{
  A *P, O1;
  B O2;
  P=&O2;
  P->f1();  EB
  P->f2();  LB
  P->f3();  LB
  P->f4();  LB
  P->f4(5); EB
}
```

error

O2: vptr

P → O2.vptr

* Abstract class in C++

⇒ Pure virtual function :- A do nothing function is called pure virtual function.

Syntax -

```
#include<iostream>
#include<conio.h>
class Person
{ public:
  virtual void fun() = 0;
};
```

virtual = #class which contain dummy function that its object is never defined (object नहीं बनेगा)

= 0; => do nothing

necessary (for late Binding)

pure virtual function = do nothing

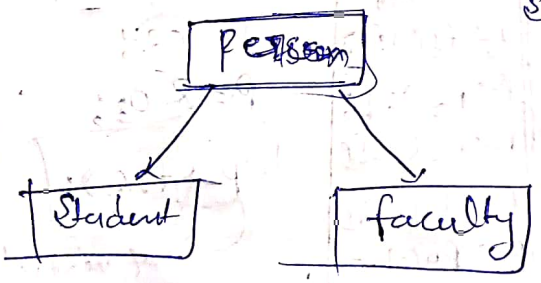
{ int x; }

⇒ Abstract class ⇒ A class containing pure virtual function is an abstract class. (जिसका कोई object नहीं बनेगा)

- We can not instantiate abstract class. (object नहीं बना सकते).

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

Why Abstract class? :- Just to provide common feature of Student and faculty from Person.



* Template in C++ :

- 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.

- function Template → function template is also known as generic function
- class Template

Syntax

template <class type> type func_name (type args1, type args2)
(keyword)

or

for more than one type

template <class type1, class type2 ...>

```
# include <iostream>
using namespace std;
template <class X>  → placeholder. It replaced by proper data type at the time of function call.
                type
X big (X a, X b)
{
  if (a>b)
     return(a);
  else return(b);
}
int main()
{
  cout << big(4,5);
  cout << big (5.6, 3.4);
}
```

Output

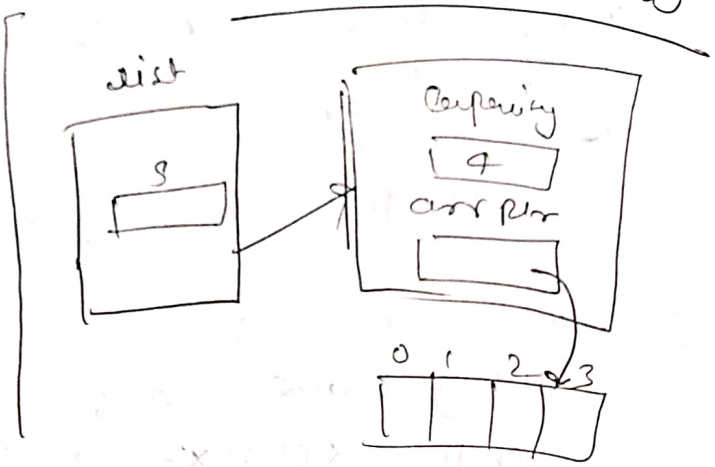
5 5.6

- class Template - class Template is also known as generic class.

```
template <class type> class class_name{ ... };
```

```
#include<iostream>
using namespace std;
template<class X> class Arraylist
{
  private:
    struct ControlBlock
    { int capacity;
      X *arr_ptr;
    };
    ControlBlock *s;
  public:
    Arraylist(int capacity)
    {
      s = new ControlBlock
      s->capacity = capacity;
      s->arr_ptr = new X[s->capacity];
    void addElement(int index, X data)
    { if(index>=0 && index<=s->capacity-1)
        s->arr_ptr[index]=data;
      else
        cout<<"\n Array index i's not valid";
    }
    void viewElement(int index, X &data)
    { if(index>=0 && index<=s->capacity-1)
        data = s->arr_ptr[index];
      else
        cout<<"\n Array index is not valid";
    }
    void listt()
    { int i;
      for(i=0; i<s->capacity; i++)
        cout<<"  "<<s->arr
```

diagram overview



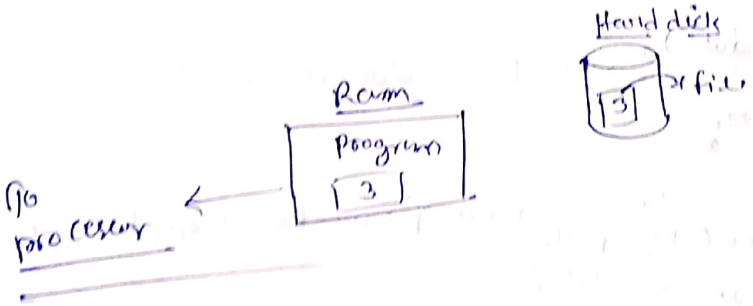
```
int main()
{ int data;
ArrayList list(4);
ArrayList <int> list1(4);
or
list1.addElement(0,32);
list1.addElement(1,41);
list1.addElement(2,95);
list.viewList();
```

* Concrete class
- Concrete class
- Pointer arithmetic
- File Handling
- Random Access
- Object Serialization
- Namespaces,
- Exception Handling.
- Overloading template function
- Constructor and Destructor and Exception Handling.

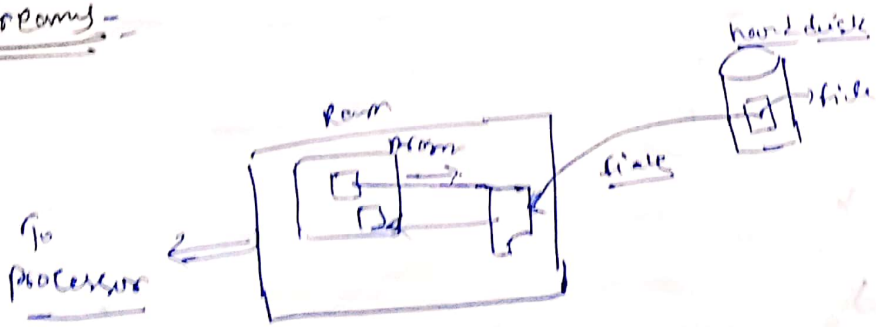
* Concrete class → Concrete classes implements a concrete concept they can be instantiated they may inherit from an abstract class or another concrete class. So far the classes we studied were concrete classes

→ File handling:

Data persistence - life of data

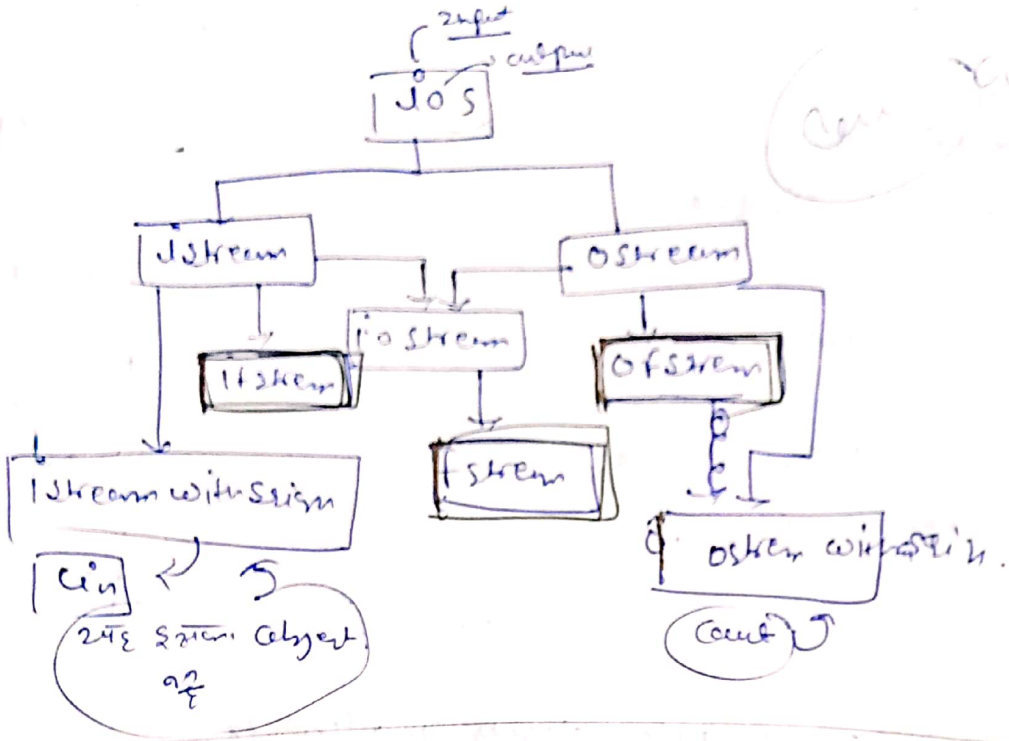


Streams -



Variable to file → out put stream

file to variable → input stream



for writing in file / Create file compile file (program to file)

ofstream

```
#include<fstream.h>   → ifstream
#include<iostream.h>  → fstream
#include<conio.h>
void main()
{
  ofstream fout;        → for writing in file
                  object
  fout.open("my file.dat");   → file hard disk में save होगी
  getch();
  fout<<"hello";
  getch();
  fout.close();
  getch();
}
```

function (member function of ofstream)

file transfer → from Ram to hard disk

---

file to program and program screen
or reading of file

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
void main()
{
  clrscr();
```

```
ifstream fin;
fin.open ("my file.dat");
~~(int) Ch;~~            (or) == ch = fin.get();
fin >> ch;
while (! fin.eof ())
{ cout << ch;
  fin >> ch;             or = ch = fin.get();
}
fin.close();
getch();
}
```

Hello Student   Hello Student
                          (gap)

file opening modes

- ios::in — input / read
- ios::out — output / write → ~~~~ old content erase ho jata hai
- ios::app — append → it is used to append the data (without erasing old data)
- ios::ate — update (randomly operation perform)
- ios::binary — binary

```
# include<conio.h>
# include<fstream.h>
# include<iostream.h>
~~# include~~
ofstream fout;
~~fout.open ("my file.dat", ios::out);~~
```
→ not necessary here (because output mode is default mode) ~~automatic~~ ' on

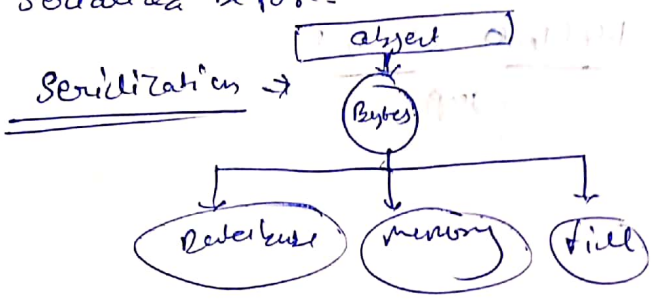
```
fout.open ("my file.dat", ios::app, ios::binary)
```
↳ इसका मतलब file binary mode में open होगी
अगर ये नहीं लिखा तो file Text mode में open होगी

Text mode vs binary mode

- Text mode is the default opening mode
- Binary mode can be specified with ios::binary
- Special characters are translation e.g. \n

Serialization

- Serialization is the process of converting an object into a stream.
- Deserialization is the process of reconstructing an object that has been serialized before.



De serialization
→ opposite process of serialization.

⟹ Why Serialize

1. Store data / state of an object. →
2. Transmit data / state of an Object
3. clone an object without overriding clone

Serialize Object ⟹ file

→ To store a serialized object in a file

Create an objectoutputstream
with a fileoutputstream

fileoutputStream vs objectoutputStream

→ writes binary data to a file
// it means. sequence of bytes → file

⇒ convert a java ob

Exceptions handling :-

Exceptions :-

- Exception is any abnormal behaviour which is run time error.
- Exception are off beat situation in your program where your program should be ready to handle it with appropriate response

Exception Handling :-

- C++ provides a built-in error handling mechanism that is (it is a way) called exception handling.
- using exception handling, you can more easily manage and respond to runtime errors.

try, catch, throw :- all are keywords which are used in exception handling process.

- program statements that you want to monitor for exception are contained in 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 (type 1 arg) {
}
catch (type 2 arg) {
}
---
catch (type N arg) {
}
```

-> all catch function are in sequence. i.e no coding are done b/w catch and try.

```
# include <iostream>
# include <conio.h>
using namespace std;
int main()
{
  cout<<"welcome";
  try {
     throw 10;
     cout<<"\n In Try";
  }
  catch (int e) {
  cout<<"\n Exception no:"<<e;
  }
   cout<<"\n Last line";
   getch();
}
```

(Fit) try and catch both are use together

→ it is data or data type of any kind

→ it is not necessary to mention argument.

Output →

Welcome
Exception no: 10
Last line

Catch

- When an exception is caught, arg will receive its value.
- If you don't need access to the exception itself, specify only type in the catch clause, arg is optional.
- Any type of data can be caught, including classes that you create.

throw :

- the general form of the throw statement is;

  throw exception;

- Throw must be executed either within the try block proper or from any function that the code within the

(Random Access)

* tellg()
  - Defined in istream class
  - Its prototype is
    - streampos tellg();
  - Returns the position of the current character in the input stream

* tellp()
  - Defined in ostream class
  - Its prototype is
    - streampos tellp();
  - Returns the position of the current character in the output stream

tellg →

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
  char ch;
  ifstream fin;
  fin.open("abc.txt");
  int pos;
  pos = fin.tellg();
  cout<<pos;
  fin>>ch;
  pos=fin.tellg();
```

ex. file contain →
Hello students
01234

Output = 0 1 2

```
cout << pos;
fin >> ch;
pos = fin.tellg();
cout << pos;
getch();
```

→ tellp:-

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
ofstream fout;
char ch;
fout.open("abc.text", ios::app);
int pos;
pos = fout.tellp();
cout<<pos;
fout << "my sirg";
fout.close();
getch();
```

Out 06

(get) function (integer type ka) value return karti hai

* seekg()

- 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

→ seekg() ->

file - ABC
on open.
Hello Student mySir

```
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
  ifstream fin();
    fin.open("abc.txt");
  cout<<fin.tellg();
  cout<<"\n"<<(char)fin.get();
  cout<<(char)fin.get();
  cout<<"\n"<<fin.tellg();
```

```
fin.seekg(6);
cout<<"\n"<<fin.tellg();
cout<<"\n"<<(char)fin.get();
cout<<"\n"<<fin.tellg();
fin.seekg(-2, ios_base::end);
cout<<"\n"<<fin.tellg();
getch();
```

Output

```
0
He
2
6
S
7
18
```

* seekp()
  - Refined in Ostream class
  - its prototype is
    - ostream& seekp(streampos pos);
    - ostream& seekp(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.