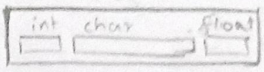
C++ OOPs →

NOTE: 'void main(){}' is only for understanding purpose. We must use 'int main(){}'

Structure:

- Structure is collection of dissimilar elements.
- Structure is a way to group variables.
- Structure is used to create data type.



ex:
```
struct book{
    int bookid;
    char title[20];
    float price;
};
```

(aka: member variables)

- doesn't consume space in memory until a variable of this type is declared
- "struct book" → just definition (no memory allocated)
- if in "int main(){"
  `struct book b1;` } → memory allocated
  (optional to use).

- We cannot access members of struct directly. ('public' by default)

```
void main(){
    book b1 = {100, "ankit verma", 10.0};
    book b2, b3;
    b2.bookid = 20;
    strcpy(b2.title, "ankit verma");   // makes string to array of 'chars'
    b3 = b2;   → Shallow copy → (b3 & b2 are not pointing to the same address)
}
```

- We can pass struct as arguments & make functions of 'struct' datatype.

ex:
```
// We can also have 'functions' inside a struct.
struct book{
    :
    void display(){
        cout<<"Hi";
    }
};

void main(){                    // Encapsulation concept
    book b1;
    b1.display();
}
```

ex:
```
struct book{
private:                  → can only be accessed inside this class
    int bookid;
public:                   → global access
    float price;
// display() func.
};
```

Class: · By default, everything is 'private'. } → (only difference btw class & struct)

ex:
```
class Complex{
    int a,b;          } instance member variable
    pubic:
    void set_data(int x, int y){   } instance member function
        a=x;  b=y;
    }
};
```

```
void main(){
    Complex c1;
    c1.set_data(3,4);
}
```

'c1' is an 'object'

- Object consumes memory & the class does not

ex:
```
class Complex{
    int a,b;
    public:
    ....
    Complex add(Complex c){
        complex temp;
        temp.a = a+c.a;  temp.b = b+c.b;
        return temp;
    }
};
```

```
void main(){
    Complex c1,c2,c3;
    c1.set_data(3,5);
    c2.set_data(3,7);
    c3 = c1.add(c2);
}
```

- Class = description of an object // Object = instance of a class
  {Instance ~ Object}

- Instance member variables = Attributes, Data members, fields, properties
- Instance member functions = Methods, procedures, actions

Static Members:

1) Static Local Variable:

- Concept as it is taken from C
- They're by default initialized to zero.
- Their lifetime is throughout the program.

ex:
```
void func(){
    static int x;
    cout<<x; //op: 0
}
```

2) Static Member Variable:

- Declared inside the class body
- aka "class member variable."
- they must be defined outside the class.
- Static member variable does not belong to any object, but to the whole class.
- There will be only one copy of static member variable for the whole class.
- Any object can use the same copy of class variable.

ex:
```
class Account{
    int balance;          // instance member variable
    static float roi;     // static member variable
    public:
    void set_bal(int b){
        balance = b;
    }
};
float Account:: roi = 3.5f;   // scope resolution operator
```
(membership label)

// I can declare many objects but 'roi' value will be same & will be declared only once

3) Static Member Function:

- They are qualified with the keyword 'static'.
- aka "class member functions"
- They can be invoked with or without object
- They can only access static members of the class.

ex:
```
class Account{
    static float roi;
public:
    static void setRoi(float r){
        roi = r;
    }
};
float Account::roi = 3.5; → one-time global memory

void main(){
    Account::setRoi(4.0);
}
```

⇒ Condition:

	Will 'roi' consume memory?
declared but not defined	NO
defined but never used	Depends on optimization
defined and used	YES

Constructor

- Constructor is a member func. of a class.
- Name of the constructor = Name of the class.
- It has no return type, so can't use return keyword.
- It must be an instance member func., i.e., it can never be static.

// How to call?

- Constructor is implicitly invoked when an object is created.
- It is used to solve prob. of initialization.
- If we did not write a constructor, then when an object is created, the compiler creates a constructor implicitly internally. If we made any type of constructor, then compiler won't create any constructor on its own.

ex:
```
class Complex{
    int a,b;
public:
    Complex(int x, int y)
    {a=x; b=y;}          → Parameterized Constructor
    Complex(int z){
        a=z;
    }
    Complex(){}          — Default Constructor
};

void main(){
    Complex c1 = Complex(3,4)
    , c2 = 5, c3, c4(3,8);
}
```

diff. ways of calling

→ CONSTRUCTOR OVERLOADING = More than 1 constructors defined

You Define ...	Compiler Creates... (internally)	
	Default Constructor	Copy Constructor
Nothing	✓	✓
Parameterized Constructor	✗	✓
Copy Constructor only	✓	✗
Copy + Parameterized	✗	✗

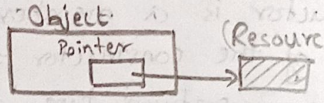
Ex:
```
public:
  // Parameterized Cons.
     :
  // Default Cons.
  Complex(){}
  // Copy Constructor
  Complex(Complex &c){
     a = c.a;
     b = c.b;
  }
```

```
void main(){
   Complex c1(3,4);
   Complex c4(c1);
   Complex c5=c1;
}
```

→ If we don't pass by ref., it will give error as the constructor (passed by value) will initialize itself as we pass. → ∞ recursion (constructor call)

Destructor:

- Destructor is an instance member func. of a class.
- The name of the destructor is same as the name of a class but preceded tilde (~) symbol. (ex: ~Complex(){})
- Destructors can never be static.
- They have no return type.
- Destructors takes no argument (No overloading is possible)
- It is invoked implicitly when object is going to destroy.



//Why Destructor → It should be defined to release resources allocated to an object.

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.

//Rules

- Any symbol can be used as function name → if it is a valid operator in C language
  → if it is preceded by 'operator' keyword.
- You can not overload 'sizeof' & '?:' keyword.

ex: public:
previously = 'add'

```
Complex operator + (Complex c){
    Complex temp;
    temp.a = a + c.a;
    temp.b = b + c.b;
    return temp;
}
```

also written as;

```
c3 = c1.operator+(c2);
```

```
void main(){
    Complex c1, c2, c3;
    c1.setData(3,4);
    c2.setData(3,6);
    c3 = c1+c2;
}
```

(Previously we used to write c1.add(c2))

// Here '+' now also adds non-primitive datatypes (like 'Complex')

→ Compiler can not use operators on non-primitive datatypes by default

// Overloading of Unary Operator

ex:

```
public:
....
Complex operator - (){
    Complex temp;
    temp.a = -a;
    temp.b = -b;
    return temp;
}
```

```
void main(){
    Complex c1, c2, c3;
    c1.setData(3,4);
    c2 = c1.operator - ();
    (or)
    c2 = -c1;
}
```

// ++ (Pre & Post)

ex:

```
class Integer{
    private:
    int x;
    public:
    void setData(int a){ x = a; }
    void showData(){ cout << "x: " << x; }
    Integer operator++(){ //pre increment
        Integer i;
        i.x = ++x;
        return i;
    }
    Integer operator++(int){ //post increment
        Integer i;
        i.x = x++;
        return i;
    }
};
```

```
void main(){
    Integer i1, i2;
    i1.setData(3);
    i1.showData();
    i2 = i1++; → post increment
    i1.showData();
    i2.showData();
    i2 = ++i1; → pre increment
    (  );
}
```

Friend Function:

- Friend func. is not a member func. of a class to which it is a friend.
- It is declared in the class using 'friend' keyword.
- It must be defined outside the class to which it is friend.

- Friend func. can access any member of the class to which it is friend.
- Friend func. cannot access members of the class directly.
- It has no caller object.
- It should not be defined with membership label.
- Friend func. can become friend to more than one class.
- Friend func. can be declared anywhere in the class. Since it's not a member func., access modifiers (public, protected, private) won't have any effect.

ex:
```
class B;
class A{
    int a;
public:
    friend void fun(A,B);
};
```
(pass only datatype)

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

```
void fun(A o1, B o2){
    cout<< o1.a+o2.b;
}
```

// Friend func. helps us in accessing variables of 2 classes at the same time.

```
void main(){
    A obj1;
    B obj2;
    obj1.setData(2);
    obj2.setData(3);
    fun(obj1, obj2);
}
```

// Overloading of operators as a friend function

// Refer prev. page → Operator overloading ← if we define as a member func.

ex:
```
class Complex{
    int a,b;
public:
    // setData--
    // showData--
    friend Complex operator +(Complex, Complex);
};
void main(){
    Complex c1,c2,c3;
    c1.setData(3,4);
    c2.setData(7,5);
    c3=c1+c2 ~ means; operator+(c1,c2)
}
```

```
Complex operator +(Complex X, Complex Y){
    Complex temp;
    temp.a= X.a+ Y.a;
    temp.b = X.b+ Y.b;
    return temp;
}
```

→ (refer how we wrote it)

(conclusion: if defined as a friend, we pass one more argument compared to when defined as a member func.)

// Similarly for unary operator

ex:
```
class Complex{
    friend Complex operator- (Complex);
};
```

```
Complex operator- (Complex x){
    // refer prev. page
}
```

```
void main(){
    :
    c2 = -c1;
    c2.showData();
}
```

// Overloading of insertion & extraction operator:

ex:
```
class Complex{
    int a,b;
    public:
    :
    friend ostream& operator(ostream&, Complex&);
    friend istream& operator(istream&, Complex&);
};
ostream& operator<<(ostream& dout, Complex& c){
    dout<< c.a << " " << c.b;
    return dout;
}
istream& operator<<(istream& din, Complex& c){
    din >> c.a >> c.b;
    return din;
}
```

```
void main(){
    Complex c1;
    cin >> c1;
    cout << c1;
}
```

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

ex:
```
class A{
    public:
    void fun(){...}
    void foo(){...}
};
```

```
class B{
    :
    // friend void A::fun();
    friend class A;
};
```

→ all member func. of 'A' will become friend.

Inheritance:

- It is a process of inheriting properties & behaviours of existing class into a new class.
- Syntax:
```
class Base_Class{
    :
};
class Derived_Class : Access_Modifier Base_Class{
    :
};
```
(by def. → private)

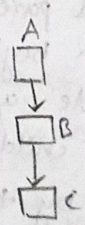
- Types: (we have 5 types)

1) Single Inheritance:

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

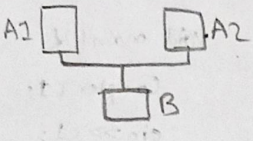
2) Multilevel Inheritance:

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



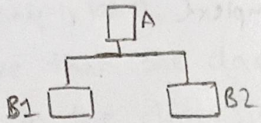
3) Multiple Inheritance:

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



4) Hierarchical Inheritance:

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



5) Hybrid Inheritance: Mix of above → Refer "diamond problem" after "Virtual Destructor"

- Access Modifier:

Access Modifier \ Scope	Global	Derived Class	Friend Class	Within Class
Private	✗	✗	✓	✓
Public	✓	✓	✓	✓
Protected	✗	✓	✓	✓

- "Is a" relationship is always implemented as a public inheritance.
  ex: Banana is a fruit

ex:
```
class A {
};
class B : public A { };
```
public A → protected A → private A

Member of A	Access in B	Access from obj. of B	Access in B	Access from obj. of B	Access in B	Access from obj. of B
public	public	ACCESSIBLE	protected	NO ACCESS	private	NO ACCESS
protected	protected	NO ACCESS	protected	NO ACCESS	private	NO ACCESS
private	NO ACCESS	NO ACCESS	NO ACCESS	NO ACCESS	NO ACCESS	NO ACCESS

Constructor & Destructor

- When a child object is instantiated, memory is allocated for both child and parent members. Though called from the child, the execution begins with the parent constructor & returns to the child code afterward.
- When a child object is destroyed, the child's destructor runs first, followed by the parent's destructor (opposite the constructor execution order)

ex: class A : public B, public C { };

//Constructor execution order : B() , C() , A()
//Destructor execution order : A() , C() , B()

'this' pointer

- A pointer contains address of an object is called Object pointer.
- 'this' is a keyword.
- 'this' is a local object pointer in every instance member function containing address of the caller object.
- 'this' pointer can not be modified. It is used to refer caller object in member function.

```
ex: class Box{
        int l, b;
    public:
    void setDim(int l, int b){
        this->l = l;
        this->b = b;
    }
};
```

'new' & 'delete'

- Static memory allocation(SMA) : → decision made at compile time.
  → fixed lifespan of variables.
- Dynamic memory allocation(DMA) : → enables memory allocation based on runtime logic using the 'new' keyword in C++

ex: int *p = new int; //Here 'p' holds the address of a newly created integer.

- Variables created using 'new' keyword do not get automatically deleted.
- The 'delete' keyword is used to release that memory.

ex: delete p;

Method Overriding : Same method name in 2 different scopes

```
ex: class Car{
    public:
    void shiftGear(){ }
    void f2(){ }
};
class SportsCar: public Car {
    public:
    void shiftGear(){ }
    void f2(int x){ }//Func. Hiding
};
```

(Same method name, diff. params) ; Since we have a func. named like that in child

```
void main(){
    SportsCar obj;
    obj.shiftGear();
    obj.f2();
    obj.f2(4);
}
```

method overriding ↑ shiftGear of child class is called

→ error ↓ as same func. name present, but arg. not matching → so error & it won't go to parent (→ B)

Virtual Function:

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

- **Early Binding:** → Func. call is resolved at compile time.
- **Late Binding:** → Func. call is resolved at run time.
  → Happens with virtual func.'s using a pointer / reference.

ex:
```
class A{                        void main(){
  public:                         A *p, o1;
  void f1(){ }                    B o2;
};                                p = &o2;
class B: public A{                o2.f1();   //B
  public:                         p->f1();   //A
  void f1(){ }
  void f2(){ }
};
```
(func. overriding)

→ In early binding, compiler only knows type of 'p' & not content of 'p', so it calls 'f1()' of class A.

But that is not what we want!

prob. of method overriding

So we introduce late binding with the help of 'virtual' keyword.

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

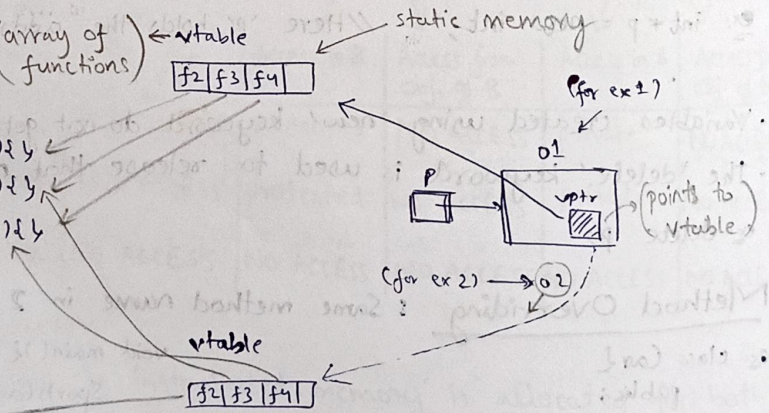
Now; 'p->f1()' → //B

All the classes inheriting parent class, the same func. name 'f1()' (i.e. function overriding) will also become virtual function, but we need not mention any keyword in child class.

Working:

When virtual func. is made, a pointer & table is also made.

ex:
```
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){ }
};
```



- If obj. of A is made '*vptr' is made. If obj. of B is made, '*vptr' is inherited from 'A'.

ex1:
```
main(){
    A *p, o1;
    p = &o1;
    p->f1(); // Early Binding (EB)
    p->f2(); // Late Binding (LB)
    p->f3(); // LB
    p->f4(); // LB
    p->f4(5); // Error (EB)
}
```
(Function Hiding ✓ Func. Overriding ✗)

ex2:
```
main(){
    A *p;
    B o2;
    p = &o2;
    p->f1(); // EB → A
    p->f2(); // LB → B
    p->f3(); // LB → A
    p->f4(); // LB → A
    p->f4(5); // Error (EB) → A
}
```
(at compile time, it sees only type & not content)

- **Pure Virtual function**: A do nothing func. is pure virtual func.
  - we cannot make its object. So to access it's other members, we need to make a child class.

Abstract class:

- A class containing pure virtual function is an abstract class.
- We can not instantiate abstract class.
- In Java, we use 'abstract' keyword to declare a class as abstract class whereas in C++, if the class has atleast 1 pure virtual func., then it becomes abstract class.

ex:
```
class Person {
public:
    virtual void fun() = 0;   // (Pure virtual func.)
    void f1() { }
};
```

```
class Student : public Person {
public:
    void fun() { } // We need to implement it in child (OR) we can again make it pure virtual func. in child but this we cannot use child class as well. To use it, make another child class.
};
```

- **Why do we use this?**: To implement 'generalization' concept of OOPS.
- A child class must define all the pure virtual functions from its parent abstract class.

Template:

- The keyword 'template' is used to define function template & class template.
- It is a way to make your function (or) class generalize as far as data type is concern.
- **Function Template**: (aka 'generic' function)

syntax:
```
template <class type> type func_name(type arg1, ....);
```
(or) placeholder

ex: template <class D>
D big(D a, D b){
if(a>b) return a;
else return b;
}

int main(){
cout<<big(4,5);
cout<<big(4.5, 1.5);
}

if I have 2 args. as (int, double)

template <class D, class E>
D big(D a, E b){
;
}

> Instead of declaring 2 func. with only difference in datatype, we can use above template.

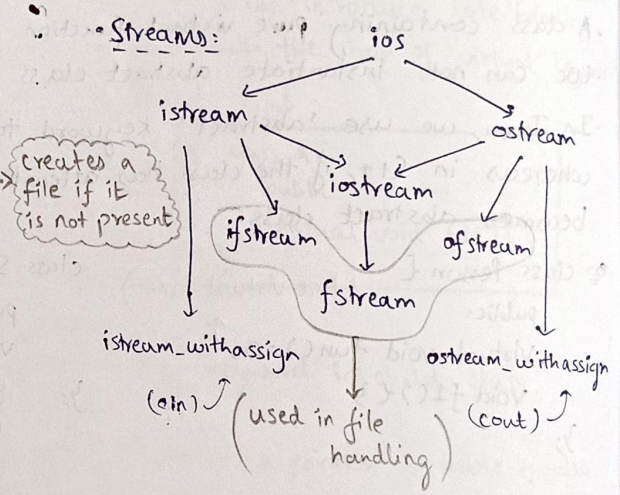
- Class Template: (aka generic class)
  - syntax: "template <class type> class class_name {...};"

File Handling:

- (For data persistence)

ex: void main(){
ofstream fout; → (object)
fout.open("myfile.txt"); → creates a file if it is not present
fout<<"hello";
fout.close();
}

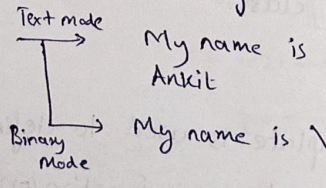
- Streams:



- File Opening Modes:
  - → ios::in = input/read
  - → ios::out = output/write
  - → ios::ate = update
  - → ios::app = append
  - → ios::binary = binary

- → Text Mode is the default opening mode.
- → Binary mode can be specified with ios::binary.

ex: fout<<"My name is \n Ankit";



ex: ofstream fout;
fout.open("myfile.txt", ios::app);

(modes are written as 2nd arg. of open() function)

→ We can initialize data members directly inside the class definition

ex:
```
class Myf
    private:
    int x = 10;
    String name = "Ankit";
    public:
    }
};
```

Deep Copy & Shallow Copy:

- **Shallow Copy**: creating copy of object by copying data of all member variable as it is.
- **Deep Copy**: creating an object by copying data of another object along with the values of memory resources residing outside the object but handled by that object.

ex: (Shallow Copy) ← default copy constructor does shallow copy

```
class Shallow{
    int* data;
    public:
    Shallow(int val){
        data = new int(val);
    }
    void show(){
        cout << "Val: " << *data;
    }
    ~Shallow(){
        delete data;
        cout << "Destructor ";
    }
};
void main(){
    Shallow obj1(10);
    Shallow obj2 = obj1;
    obj1.show();
    obj2.show();
}
```

ex: (Deep copy)

```
class Deep{
    int* data;
    public:
    Deep(int val){
        data = new int(val);
    }
    → deep copy constructor
    Deep(Deep& other){
        data = new int(*other.data);
    }   → //allocate new memory
    void show(){ cout << "Val: " << *data; }
    ~Deep(){
        delete data;
        cout << "Destructor ";
    }
};
void main(){
    Deep obj1(10);
    Deep obj2 = obj1;
    obj1.show();
    obj2.show();
}
```

Type Conversion:

1) **Primitive to Primitive**: ex:
```
int x = 4;
float y;
y = x;
```
→ automatic type conversion

ex:
```
float y = 3.4;
int x;
x = y;
```
→ automatic type conversion

2) Primitive type to Class type:

- Primitive type to Class type can be implemented through constructor.

ex:
```
class Complex{
    int a,b;
    public:
    Complex(){}
    Complex(int k){ a=k; b=0;}
    void setData(int x, int y){ a=x; b=y;}
    void showData(){cout<<a<<" "<<b;}
};
```

```
void main(){
    Complex c1;
    int x=5;
    c1 = x; //-> c1.Complex(x)
    c1.showData();
}
```

3) Class type to Primitive type:

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

```
operator type(){
    ...
    return (type-data);
}
```
→ Syntax

ex:
```
class Complex{
    public:
    void showData(){cout<<a<<" "<<b;}
    operator int(){
        return a;
    }
};
```

```
Void main(){
    Complex c1;
    c1.setData(3,4);
    c1.showData();
    int x;
    x=c1; //-> x=c1.operator int()
    cout<<x;
}
```

4) Class type to another class type:

- Conversion through constructor (OR) casting operator.

ex: (with constructor)
```
class Product{
    int m,n;
    public:
    void setData(int x, int y){
        m=x; n=y;
    }
    int getM(){ return m;}
    int getN(){return n;}
};
class Item{
    int a,b;
    public:
    Item(){}
    Item(Product p){ a=p.getM(); b=p.getN();}
    Void showData(){ cout<<a<<" "<<b;}
};
```

```
Void main(){
    Item i1;
    Product p1;
    p1.setData(3,4);
    i1=p1;
    i1.showData();
}
```

ex: (with casting operator)

- In 'class Item' → add 'setData(int x, int y)'
- In 'class Product'

```
{
} "public:
void showData() { cout<<a<<" "<<b; }
operator item(){
    item temp;
    temp.setData(m,n);
    return temp;
}
};
```

```
void main(){
//same as prev. ex.
}
```

Exception Handling

- **Exceptions:**
  - Exception is any abnormal behaviour, runtime error.
  - Exceptions are beat situation in your program where your 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, you can more easily manage & respond to runtime errors.
- → Program statements that you 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' & processed.

- **Syntax:**

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

- **ex:**

```
try{
    throw 10;
}
catch(double e){
    cout<<e<<" exception";
}
catch(int e){
    cout<<"int exception "<<e;
}
```

- **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.

```
// catch all type of errors
catch(...){
    cout<<"Exception";
}
```

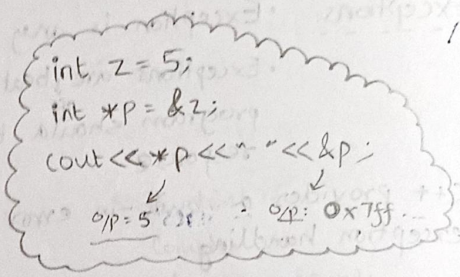
- **throw:** → the general form of throw statement is: "throw exception;"
  → Throw must be executed either within the try block or from any function that the code within the block calls.

Dynamic Constructor

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

ex:
```
class AC {
    int a,b;
    int *p;
    public:
    A() {
        a=0; b=0;
        p= new int;
    }
    A(int x, int y, int z){
        a=x; b=y;
        p= new int;
        *p = z;
    }
};
```

```
void main() {
    A o1, o2, o3(3,4,5);
}
```



namespace

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

// How to create namespace?

- "namespace MySpace { .. }"
- Namespace 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.
- You can use an alias name for your namespace name, for ease of use
  ↳ "namespace ns = MySpace;"
- There can be "unnamed namespaces" too → "namespace { }"
- A namespace definition can be continued & extended over multiple files, they are NOT redefined or overridden.

ex:

File1.h →
```
namespace My{
    int a,b;
    void f1();
}
```

File2.h →
```
namespace My{
    int x,y;
    void f2();
}
```

Access ? → Using "::"

ex:
```
#include<iostream>
using namespace std;
namespace MyS{
    int a;
    void f1();
    class Hello{
      public:
      void Hello(){
          cout<<"Hello";
      }
    };
}
void MyS::f1(){cout<<"In f1";}
int main(){
    MyS::a=5;
    MyS::Hello obj;
    obj.hello();
    MyS::f1();
}
```

- 'using' keyword allows you to import an entire namespace into your program with a global scope. It can also be used to import a namespace into another namespace or any program.

Virtual Destructor

- Base pointer can point to the object of derived class. (i.e. Parent class pointer ↓ can store address of any of its descendant child class.)

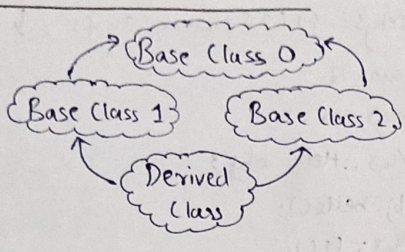
ex:
```
class A{
    int a;
    public:
    void f1(){}
    virtual ~A(){}
};

class B: public A{
    int b;
    public:
    void f2(){}
    ~B(){}
};

int fun(){
    A *p = new B;
    p->f1(); //correct
    p->f2(); //error
}
```

- Now when destructor is called → 'delete p'
  - due to early binding, compiler will think 'p' refers to class 'A' & deletes it. But this is wrong since 'p' points to class 'B'.
  - so we define virtual destructor, which does late binding i.e. it sees the content by 'p' at runtime

- If parent destructor is 'virtual' then automatically child destructors are also virtual & no need to mention keyword in child's destructor.
- If we don't call destructor → "memory leak" can happen.
  - occurs when a program allocates memory but fails to release it after it's no longer needed.

Diamond Problem: → comes in Hybrid Inheritance

ex:



Hierarchical + Inheritence

Multiple Inheritence

ex:
```
class base{
    protected:
    int val;
};

class der1 : public base{
    public:
    der1(){ val = 1;}
};

class der2 : public base{
    public:
    der2(){ val = 2;}
};

class der3 : public der1, public der2 {
    public:
    void get(){
        cout<<"val"<<val;
    }
};
```

(ERROR)

(as we have 2 copies of 'val' from parent classes)

to solve:

1) ↳ we can print like: "der1::get()" → define 'get()' func: which prints 'val'.
(or)
"der2::get()" →

2) Virtual Inheritance:

- Use 'virtual' keyword while inheriting

ex:
```
class der1 : virtual public base { _ };
class der2 : virtual public base { _ };
```

//Now in 'der3' →
```
class der3 : public der1 , public der2 {
    public:
    void get(){
        cout << der1::get() << der2::get();
    }
};
```
'2' '2'

(rightmost class value is considered)

If we want 'der1' to get printed, just write as:

"class der3 : public der2, public der1 {... }"

Nested class

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