* Classes & Objects

⇒
```
Class classname
{
  Access specifier:
                variable declarations;
  Access specifier:
                function declarations;
};
```

- Access Control
  - → Access specifier (data hiding)
    - → Private (default)
    - → Public
    - → Protected.

	Same class	Derived class	Outside the class
Private	✓	X	X
Public	✓	✓	✓
Protected	✓	✓	X

5 Minutes Engineering

Public

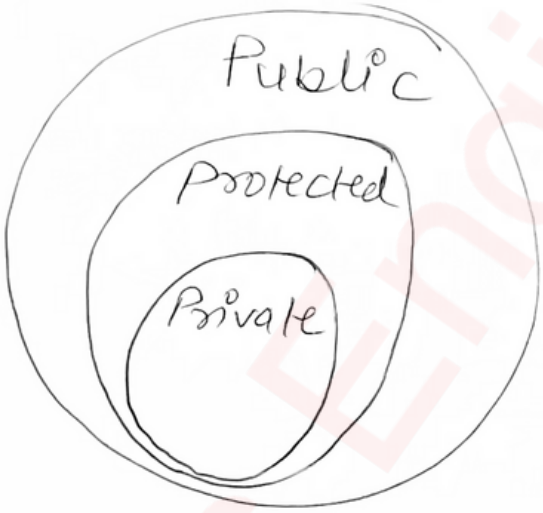
```
Class PublicAccess
{
  Public:
      int x;
      void display();
}
```

Private

```
Class PrivateAccess
{ Private:
      int x;
      void display();
}
```

Protected

```
Class Protected
{ Protected:
      int x;
      void display();
}
```



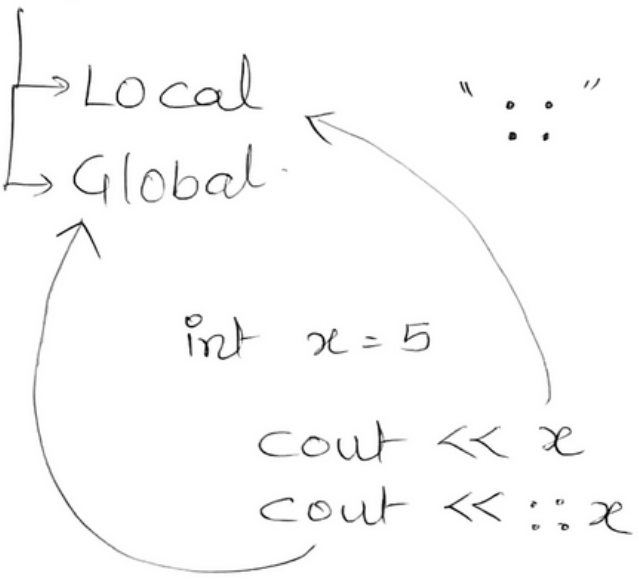
Object: Instance of a class

↳ classname objectname;

```
PublicAccess P1;
P1.x = 5
P1.display();
```

5 Minutes Engineering

Scope Resolution



Note: (::) is used to define a fn outside a class.

```
eg: Class abc
    { Public:
        void display();
    }
    void abc :: display()
    { cout << "Shridhar"; }
```

* functions

```
returntype fn name()
{   fn body statements;
}

int main()
{ fn name(); }
```

5 Minutes Engineering

→ function Prototype [①, ② & ③]

fn header [① returntype ② functionName (parameter List) ③

④ → {

function statements

return variable / literals of return type.

⑤ ⑥

⑦ → }

- Types
  - → User defined function
  - → Library function (built in)

- Parameter Passing to functions

```
class add{                          ⟶ formal Parameter
  int addf(int a, int b){
    return a+b }
  }
  main{
  add x
  int c = 5 , d = 10                ⟶ Actual Parameter
  int r = x.addf(c, d);
  cout<<r }
```

5 Minutes Engineering

Call by Value Vs Call by reference

Call by Value	Call by reference
① Copy of value is passed	① Address of value is passed.
② Actual & formal parameter will have different memory location	② Actual & formal Parameter will have same memory location.
③ can't change value of AP by using FP	③ can change value of AP by using FP
④ No pointers used	④ Pointers are used
⑤ less efficient	⑤ More efficient

eg:
```
swap( x , y)
{ int t;
  t = x
  x = y
  y = t
}
```

swap(a, b)

↳ a = 5, b = 10 } same for after & before swap

```
swap (*x , * y )
{ int t
  t = * x
  *x = * y
  *y = t
}
```

swap(&a , &b)

a = 5 ← before, 10 ← after

b = 10, 5 } different

5 Minutes Engineering

- Main function
  - without parameters
    - int main() { ... return 0; }
  - with parameters
    - int main(int argc, char *argv[])
      { ... return 0; }

Argumen count — argument vector

Terming I/p

⇒ ./main Sheidhar Mankar.

argc (2)

argv[0] → ./main
argv[1] → Sheidhar
argv[2] → mankar

- Inline function
  - fast, No overhead of fn call & return
  - inline returntype fnname(parameter)
    { Block }

5 Minutes Engineering

Normal function

```
main()
{
  myfun();
}
```

flow of control transfer →

```
myfun()
{ body
}
```

Inline function

```
main()
{
  myfun();
    myfunc()
    { body
    }
}
```

⇒ where it won't work

1. loops
2. static variables
3. Recursive fn
4. switch or goto.

eg:-

```
inline int add (int a, int b)
{ return a+b }

main()
{
  count << "addi is " << add(1,2)
}
```

ImpNote:- All functions defined inside the class are implicitly `Inline`.

Explicitly → declare in class
define outside with inline keyword

5 Minutes Engineering

- Memory allocation for objects:
  - At the time of declaration
  - Not when class is defined.
  - Every obj. have separate individual copy of all variables of class.
  - Objects doesn't have separate individual copies of fn., only one copy is shared among each object.

- Static Data members
  - Only a single copy of data member is shared among all objs of class.

eg:-
```
class student
{ public:
        static int total;
        int Roll no;
        void addstudent()
        { total++; } };
int student :: total = 0;
```

5 Minutes Engineering

- Static Methods / member function.

  - Can have access to only other static (data) members declared in same class.
  - Can be called using class name instead of objects.

  ⇒ classname :: functionname.

eg:-

```
class abc
{   int code;
    static int count
  public:
        void setcode()
        {
          code = ++count
        }
        static void count()
        { cout << count }
};
int abc::count
main()
{ abc a1, a2
    a1.setcode
    a2.setcode
  abc::count() }
```

5 Minutes Engineering

- Array of objects: (AOB)
  - Array of variables of type 'class'.
  - syntax:

```
class classname
{
    Public:
        datatype members;
        memberfunctions
};
```

(AOB)↓

```
classname objectname[size];
```

size of array.

eg: students S[50]

Eg:

```
class students
{
    int rollno;
    Public:
        void set(int a)
        {
            rollno = a;
        }
        int get()
        { return rollno; } };
int main()
{ students S[50];
    for(i=0; i<50; i++) {
        S[i].set(i) }
    return 0; }
```

5 Minutes Engineering

* friend functions:

→ The function friendly to a class
→ declaring a fn in class & defining it outside class with friend keyword.

eg:-

```
class calculate
  {

    Public:
        int add ( int x, int y );
      friend int Sub ( int p, int q );
  };
int Calculate::    add (int x, int y)
  { return x+y }
int sub (int p, int q)
  {   return  p-q }
  void main()
  {
    calculate c1;
      int A = c1.add(2,3)
      int S =     sub(5,3)
```

5 Minutes Engineering

Friend class:

→ It has access to private & protected members of other classes in which it is declared as a friend.

→ Syntax:

```
Class abc{
    friend class xyz;   } Base class.
}
class xyz{
    statements;         } friend class
}
```

Eg:-

```
class abc{
    private:
        int m
    protected:
        int n
    public:
        abc()
        { m = 5
          n = 10 }
    friend class xyz;
};

Class xyz{
    public:
        void display
            (abc & p)
        { cout<<p.m
          cout<<p.n }};

main()
{ abc K;
  xyz R;
  R.display(K);
}
```

5 Minutes Engineering

• Constructors and Destructors

* Constructors

- Special member function
- Initialize the obj of its class
- Same name as the class name
- Invoked when an obj is created (associated class)
- Should be declared in public section
- NO return type.
- Types
  - Default
  - Parameterized
  - Copy

① Default:

```
Class abc{
  int a, b
    public:
        abc()
      { a = 5
        b = 10 }
      void get()
      { cout << a << b }
};
```

```
void main()
{ abc x;
  x.get()
}
```

5 Minutes Engineering

② Parameterized:

```
class abc
{ int a, b
  public:
     abc (int m, int n)
     {
       a = m;
       b = n;
     }
    void get()
    {cout<<a<<b}};
```

```
void main()
  { abc x(5,10);
    x.get()
  }
```

③ Copy:

```
class abc
{ int a;
  public:
      abc()
      {a=0;
      }
     abc(int x)
     {
       a=x;
     }
     abc(abc &obj)
       {  a = obj.n
       }
    void display()
      {
       cout<<n
    }};
```

```
void main()
{ abc obj1(5)
  abc obj2(obj1)
  abc obj3=obj1
  abc obj4
      obj4=obj1
  obj1.display();
  obj2.display();
  obj3.display();
  obj4.display();
}
```

5 Minutes Engineering

Multiple Constructor:

```
class multiple {
    int x, y
    Public:
        multiple ()
        { x = y = 5;
        }
        multiple (int a)
        { x = y = a
        }
        multiple (int a, int b)
        { x = a
          y = b }
        multiple (multiple & z)
        { x = z.x
          y = z.y
        } };
```

```
void main ()
{ multiple m1
  multiple m2(3)
  multiple m3(1,2)
  multiple m4(&m1)
}
```

* Desteuctor: To desteoy the objects

- Same nam as class name
- But with '~'
- No arguments, No return
- Deallocates the memory.

```
~ multiple () {
    cout << " Desteuctor ";}
```

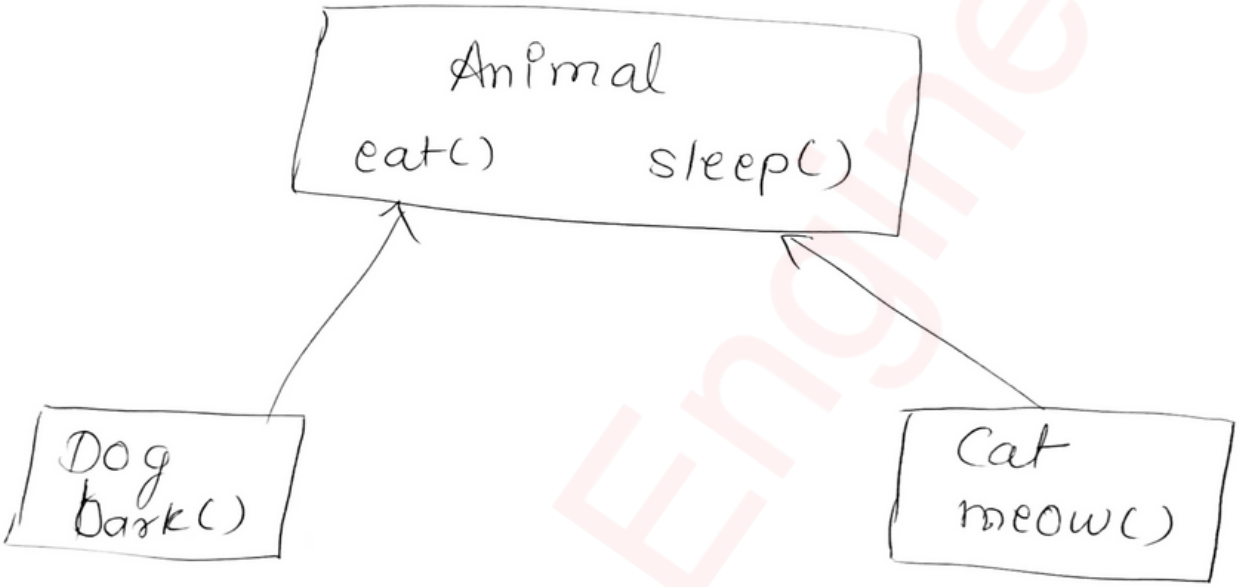
5 Minutes Engineering

* Inheritance ("is a relationship")

↳ It allows us to create a new class from an existing class

child/derived class

Parent/base class



⇒ class Dog : public Animal {...};
class cat : private Animal {...};

public → No change

protected → Public members (Base class) → protected member (derived class)

private → All members (Base class) (Public & Protected) → private member (derived class)

∴ private members of base class are inaccessible to derived class.

5 Minutes Engineering

Base class visibility	Derived class visibility		
	Public derivation	Private Derivation	Protected Derivation
Private →	Not inherited	Not inherited	Not inherited
Protected →	Protected	Private	Protected
Public →	Public	Private	Protected

* Types of Inheritance

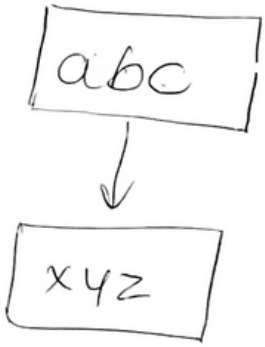
- Single
- multiple
- multi level
- Hybrid
- Hierarchical.

5 Minutes Engineering

① Single Inheritance

→ One derived class inherits from only one base class.

→ eg:



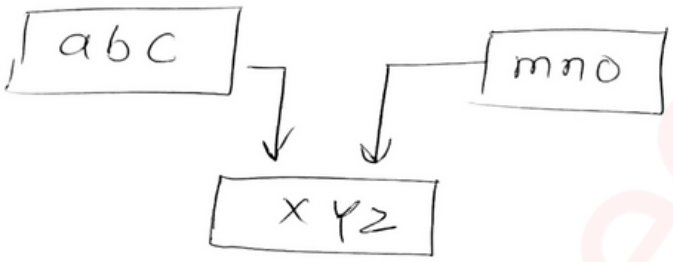
eg:

```
class abc{
public:
int a, b
 void get()
 { cout<<" Enter 2 values";
   cin >> a >> b;
 };

class xyz : public abc{
 int c;
   public:
        void add()
        {c = a+b;
         cout << c
         };

int main()
 {  xyz x1;
    x1.get()
    x1.add()
 }
```

5 Minutes Engineering

② Multiple Inheritance

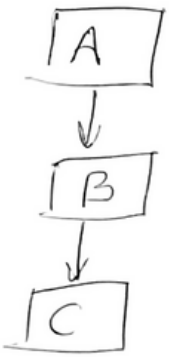
↳ Single derived class can inherit from 2 or more 2 Base classes.




```
Class abc
  { Public:
      int a
        void get1()
      { cout<<"enter value";
        cin>>a;
      };

class mno
    { Public:
        int b
          void get2()
        {cout<<"enter value";
         cin>>b };};

class xyz : public abc, public mno
  { public:
      int c  void add()
        {c = a+b
         cout <<c
        };};

int main()
 { xyz x1;
   x1.get1();
   x1.get2();
   x1.add();
 }
```

5 Minutes Engineering

③ Multilevel inheritance:



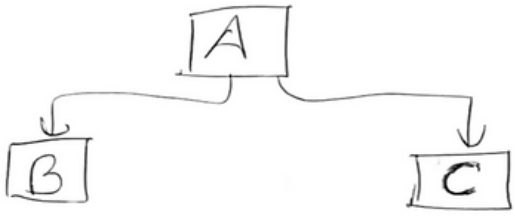
eg:-

```
class A
{ public:
        int a,b
        void get()
        { cout<<"Enter values"
          cin>>a>>b.
        }};

class B : public A
{ public:
        int c;
        void add()
        { c=a+b;
        }};

class C : public B
{ public:
        void display()
        {
          cout <<c
        }};

int main()
{ C c1;
  c1.get()
  c1.add()
  c1.display()
}
```

5 Minutes Engineering

④ Hierarchical Inheritance:

↳ One or more derived class inherit from common Base class.



eg:-

```
Class A
{ Public:
    int a, b
    void get()
    { Cout<<"enter value";
      Cin >> a ;
    }};

Class B: public A
{ public:
      void square()
      { get();
        Cout << a* a;
      }};

Class C : Public A
{ public:
      void cube()
      { get();
        Cout<<a*a*a;
      }};

int main()
{ B b1
    b1.square();
  C c1
    c1.cube();
}
```

5 Minutes Engineering

Pointers

* Memory Management

dynamic memory allocation

- C : [malloc() & calloc()]
  [free()] → deallocate memory.
- C++ : [new & delete]

Object can be created with 'new' & destroyed with 'delete'

eg:- ptr_var = new datatype

int *ptr = new int

int *ptr = new int[10]

eg:- delete ptr_var

delete ptr

delete [10] ptr

5 Minutes Engineering

Pointer Arithmetic

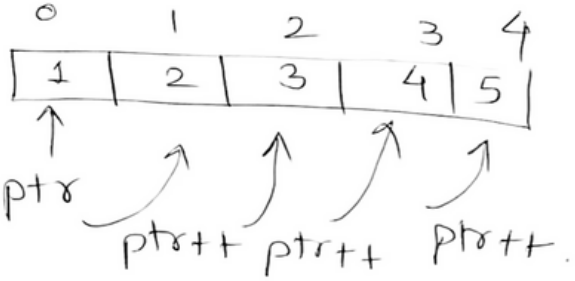
- (++) increment
- (--) decrement
- (+ or +=) An integer may be added to pointer
- (- or -=) An integer may be subtracted from pointer
- (ptr1 - ptr2) difference

eg:-

```
int arr[5] = {1, 2, 3, 4, 5}
int* ptr.
```

`ptr = V` ← addr of arr[0] to ptr.

```
for (int i=0; i<5; i++)
  {
    cout << *ptr;
    ptr++;
  }
```



2D Array & pointer Notation

=> Arr[i][j] ⟶ *(*(Arr+i)+j)

*(*Arr) ⟶ Arr[0][0]

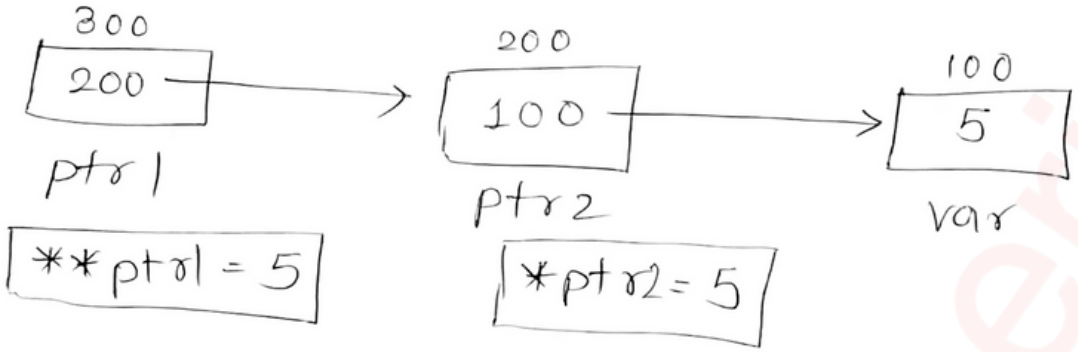
((Arr+0)+1) ⟶ Arr[0][1]

((Arr+1)+0) ⟶ Arr[1][0]

((Arr+1)+1) ⟶ Arr[1][1]

5 Minutes Engineering

◦ Pointers to pointers



At each level we require '*' unary operator

◦ void pointer

- NO associated datatype
- void keyword
- Generic pointer
- Syntax: void* ptr;

eg: int x = 5

void * ptr = &x

Cout<< ptr (addr of x)

◦ Null pointer

- It point nowhere
- int *ptr = 0
- int *ptr = NULL

5 Minutes Engineering

- Object & pointer

```
class A
  { int x;
    public:
        void put(int a)
          { x = a; }

        void display()
          { cout << x }};
```

⇒ A a1
A *ptr = &a1

① Accessing members using dot (.)

```
a1.put(5);
a1.display();
```

② Accessing members using pointer.

```
ptr→put(5)
ptr→display()
```

③ dereferencing (*) & (.) operator

```
(*ptr).put(5)
(*ptr).display()
```

5 Minutes Engineering

Pointers to derived class

↳ Declared to derived class, can be used to access members of base class & derived class.

eg:-

```
class A {
public:
        int x
        void put (int a)
        { x = a }
        void display ()
        { cout << x }};

class B : public A {
        int y;
        public:
                void putl (int a, int b)
                {
                  x = a
                  y = b }
                void display1 ()
                { cout<<x << y }};

main()
{ A a
  A *aptr
  aptr = &a
  aptr → put(5);
  aptr → display();

  B b
  B *bptr
  bptr = &b
  bptr → put(5);
  bptr → putl(5,10);
  bptr → display1(); }
```

5 Minutes Engineering

- Virtual function (method)
  - A member fn declared in base class and is re-defined by derived class.
  - To achieve Runtime polymorphism.
  - 'virtual' keyword.

* Polymorphism
  - Compile time
    - function Overloading
      - Static Binding
      - Static Linking
      - Early Binking
    - Operator overloading
  - Runtime
    - virtual function.
      - Dynamic Binding.

5 Minutes Engineering

Eg:

```
Class A {
    Public:
        virtual void display()
            {
              Cout << "Base class";
            }};
Class B : public A {
    Public:
            void display()
            {
              Cout << "Derived class";
            }};
int main(){
    A a
    B b
    A *ptr
    ptr = &a
    ptr -> display()  -> calls (A)
    ptr = &b
    ptr -> display()  -> calls (B)
    }
```

5 Minutes Engineering

Rules: (VF)

① It should be member of some class.
② It cannot be static
③ It is accessed by using obj. pointers
④ It can be a friend of another class.
⑤ It must be defined in base class even if it may not be used.
⑥ A class may have a virtual destructor but it cannot have a virtual constructor
⑦ The prototype of VF should be same in base as well as derived class.

* Static / compile time (Binding) polymorphism.

→ overloading

① function overloading

↳ fn have same name but different

eg: int abc() {...}
int abc(int x) {...}
int abc(double x) {...}
float abc(int x, double y) {...}

5 Minutes Engineering

But: int abc(int x)
int abc(int y) } error.

Same name
Same type
same no. of parameter

② operator overloading (+)

2+3 =5 2+3 =23

Syntax: class classname {
Public
returntype operator symbol
(parameters) {
. . . }};

++
−
+

⇒ Unary operator: (++) & (−−)

eg:- class abc {
private:
int x;
Public:
abc(int a)
{ x = a }
void operator ++ () {
++x }
void display()
{ cout << x }};

int main() {
abc m(5)
++m;
m.display()
return 0}

5 Minutes Engineering

To make ++ work as postfix

```
void operator ++ (int)
{ x++. }
```

⇒ Binary operator

↳ works with 2 operands.

syntax: → inside class

return type BOperator symbol (parameter) { ... };

Eg:-

```
class complex {
    Public
    float real;
    float img;

    complex (float x, float y){
        real = x
        img = y }

    Complex operator + (complex &obj){
        Complex t;
        t.real = real + obj.real;
        t.img = img + obj.img;
        return t;
    }
```

5 Minutes Engineering

int main() {

Complex c1(1, 3)

c2(2, 3)

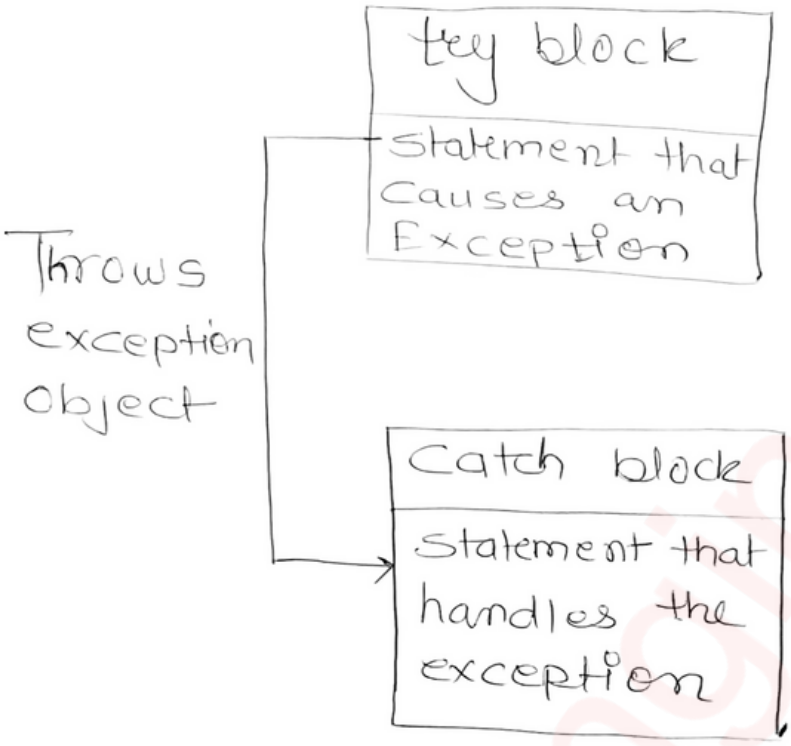
c3 = c1 + c2 → Complex & obj

↙ f^n call

⇒ sizeof, typeid, ::, ?: } cannot be overloaded.

5 Minutes Engineering

° Exception Handling



- Throwing will cause the normal program flow to be aborted, in a raised exception.

eg:-

```
int main()
{ int a, b
  cout << "Enter 2 Integer values";
  cin >> a >> b;
   try
   { if (b != 0)
       { cout << a/b; }
     else
       {
         throw b;
       }
   }
   catch (int exp)
   { cout << "Divide by zero"; } }
   return 0; }
```

5 Minutes Engineering

- **Throwing:**
  - `throw(exception);`
  - `throw exception;`
  - `throw;`

- **Catch**
  - `catch ( datatype arg ) { statement to handle exceptions }`
  - **Multiple catch statements:**

```
try {
 try block}
catch (dt1 arg)
{ CB 1 }
catch (dt2 arg)
{ CB 2 }
 catch (dt 3 arg)
{ CB 3 }
 .
 .
 .
 catch (dtn arg)
 { CB n }
```

In case of more than one match first / 1st handler is executed. (that matches exception type)

5 Minutes Engineering

eg:

```
void abc ( int a)
{
    try {
      if (a >= 1) throw a;
      else
      if (a==0) throw 'zero';
      else
      if (a<0) throw 5.0;
    }
    catch (char c)
    { cout << c
    }
    catch (int n)
    { cout << "+ve number";
    }
    catch (double d)
    { cout << "-ve numbee";
    }
}
```

```
intmain()
{
abc(1);
abc(0);
abc(-5);
return 0;
}
```

* catch All exceptions:

=> catch (...)
{
cout << " All exceptions";
}

5 Minutes Engineering

- Rethrowing an Exception

```
eg:- void abc()
     {  try
        {
          throw 1;
        }
        catch (int n)
        {
          throw;
        }
     }
     int main()
     {
        try
        {  abc();
        }
        catch (int m)
        {
          cout m;
        }
return 0;}
```

- Specifying Exceptions → "noexpt"

```
eg:- void abc(int a) throw(int, char)
     {
       switch(a)
       {
       case 1: throw a;
               break;
       case 2: throw 'a';
               break;
       case 3: throw float(a);
               break; }}
```

```
intmain()
{ try {
 abc(3); }
 catch(float n)
 {cout << " int
          type"; }
 return 0; }
```

5 Minutes Engineering

* Templates

- Allow us to write generic programs
- Function & class Templates.
- Generic types, that can work with variety of data types.
- Redundancy ↓↓
- flexibility ↑↑
- Reusability ↑↑

function Templates

```
int abc (int a, int b);
abc (float a, float b);
abc (char a, char b);
```

↓

```
T abc (T a, T b)
```
} Generic datatype.

eg:-
```
template <typename T> T abc (T a, T y)
{
    return (a > b) ?
}
int main()
{
    cout << abc <int> (1, 2);
    cout << abc <float> (1.0, 3.0);
```

5 Minutes Engineering

```
    cout << abc<char>('a', 'b');
    return 0;
}
```

- Class Templates
  - ↳

```
template <class T>
class abc {
    T x;
  Public:
    abc (T n) {
        x = n; }
    T get () {
      return x; } };
int main() {
    abc <int> obj1 (5);
    abc <float> obj2 (5.5);
    return 0; }
```

5 Minutes Engineering

Class Templates with multiple parameters

↳ eg:

default parameter

```
template <class T, class U, class V = int>
class abc {
    T x1
    U x2
    V x3
    Public:
        abc(T a1, T a2, T a3)
        {
            x1 = a1
            x2 = a2
            x3 = a3 };
int main()
{
    abc <float, char> obj1 (5.5, 'S', 5);
    abc <float, char, bool> obj2(5.5, 'S', true);
    return 0;
}
```

5 Minutes Engineering