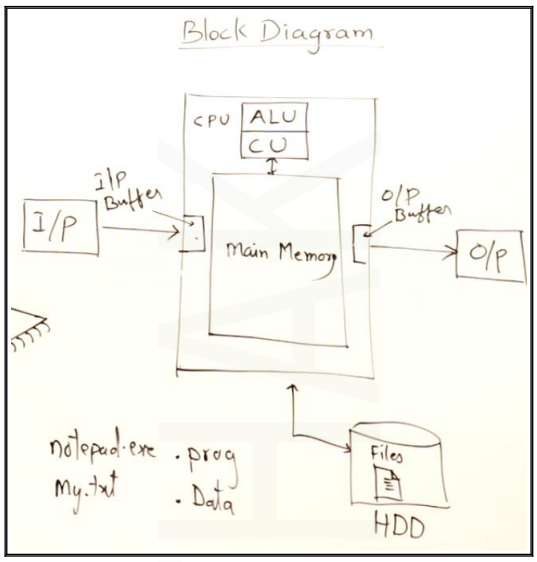
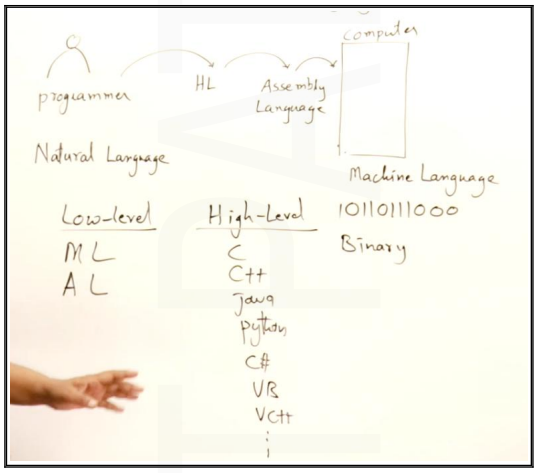
CPP NOTES

BASICS

- Block diagram of the computer



- Low level vs High level language



- Compiled vs Interpreted Language vs Hybrid language
  - Compiled Language
    - Donot need the compiler to run.They run independently.
    - Translation is done only once.
  - Interpreted language
    - They need the interpreter every time they need to run.
    - They can be run even if the code has errors.
    - Translation is done every time the program is run.
  - Compiled language
    - The code is first converted into the byte code which is the error free code using the compiler then the byte code is run using the interpreter.

- Programming Paradigm
  - Monolithic
    - All the source code is written in a single file.
    - No teamwork can be done.
    - Very lengthy code.
    - Take more time in development
    - Does not run even if a single error is there.
    - eg. BASIC language
  - Modular/Procedural

- Functions are used to make the program modular.
- Increases the reusability of the code in the same as well as other codes.
- Program can be easily developed as a team.
- eg. C

↳ Object Oriented
- The data and the functions acting upon the data are kept together and are defined as classes.
- It is higher level of modularity of the code
- eg. Java, C++

↳ Aspect oriented/Component assembly

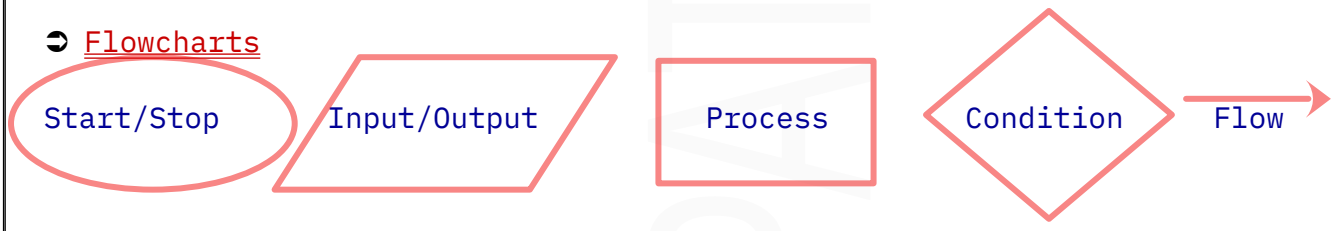
➲ Algorithms, Pseudocode and Programs

↳ Algorithms
- Step by Step procedure for solving a computational problem.

↳ Pseudocode
- Casual way of writing the procedure.

↳ Programs
- Algorithm or procedure written to solve the problem for the computer using a programming language.

➲ Flowcharts

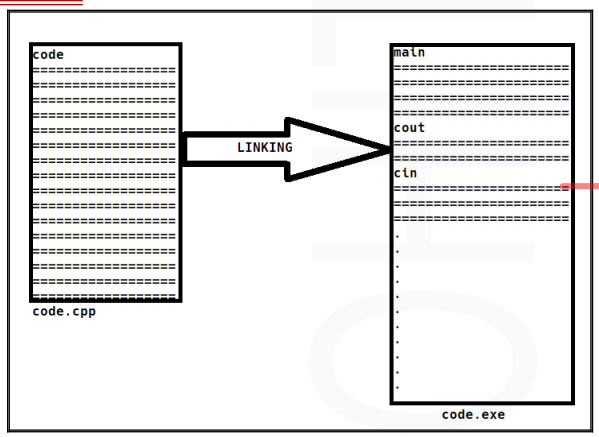


➲ Development and Execution of Program

↳ Editing

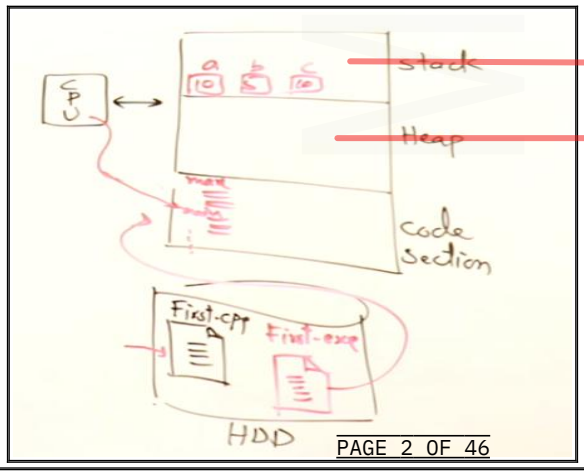
↳ Compiling

↳ Linking Libraries



The entire machine code of the header file is copied

↳ Loading



Variables and Data is stored in the stack

Heap is used for DMA

↳ Execution

C++ BASICS

➲ The first program

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

A library

Scope resolution operator

An Object: Console output

Insertion operator

namespace

↳ Namespace: All the built-in things available in the iostream library are grouped under the name std therefore for using them we need to include the word std before them.

➲ getline(cin,variable_name):cin cannot read strings having spaces in them therefore getline is used to read the strings having spaces in between, It reads until enter is encountered.

➲ Datatypes in c++

↳ Primitive

↳ Integral

↳ int : 2(turbo c) or 4 bytes; [-32768,32767]; 2^{15} =32768 and the one bit is the sign bit. -32767 t0 -0 and 0 to 32767 here -0 is taken as -32768 hence the range becomes from -32768 to 32767

↳ char : 1 byte; [-128,127]

↳ Bool : Undefined

↳ Float

↳ float : 4 bytes; [$-3.4*10^{-38}$,$3.4*10^{38}$]

↳ double : 8 bytes; [$-1.7*10^{-308}$,$1.7*10^{308}$]

↳ Userdefined

↳ Enum

↳ Structure

↳ Union

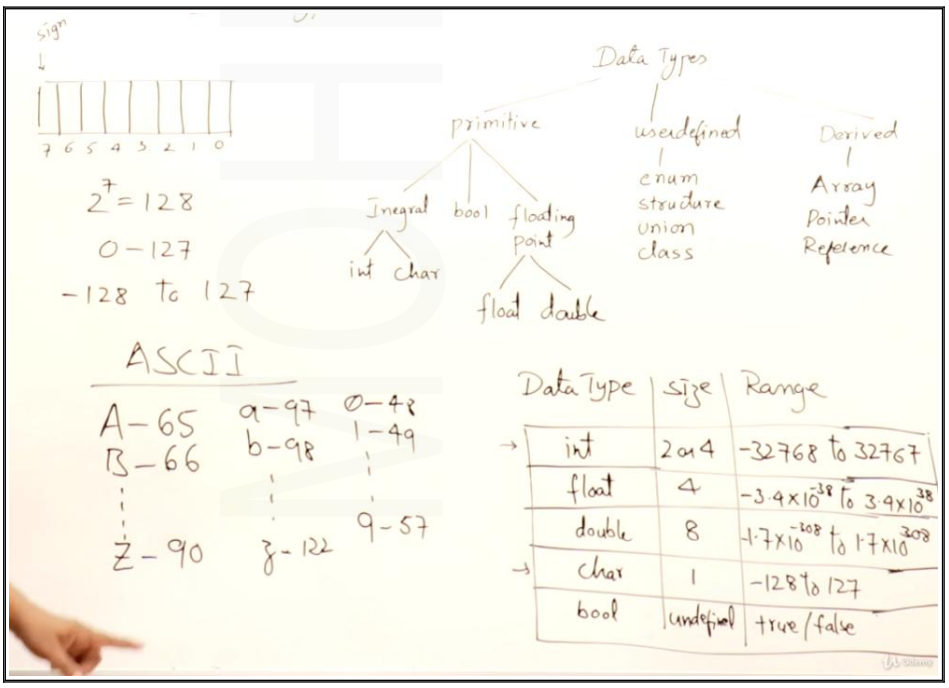
↳ Class

↳ Derived

↳ Array

↳ Pointer

↳ Reference



➲ Modifiers

↳ long

↳ long int

↳ long double

↳ unsigned

↳ unsigned int

↳ unsigned char

unsigned int : range [0 to 65535]

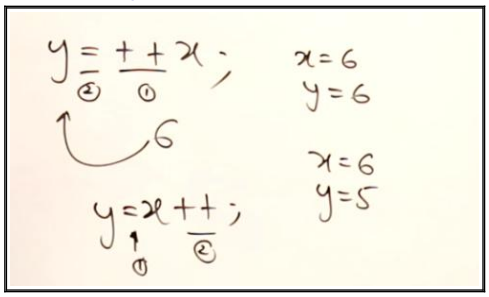
long int : size 4B(if was 2B) or 8B(if was 4B)

unsigned char : range [0 to 255]
long double : size is 10B

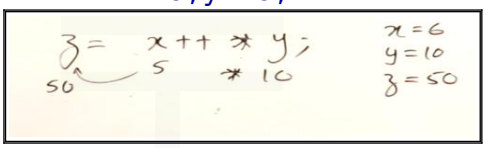
- By default any decimal number is taken as a double by the computer therefore it decimal number representing a float value should be written as the number followed by 'f'. eg. 12.75f
- int a=13,b=5;
  float c;
  if
  c=a/b;
  c=2.00
  if
  c=(float)a/b;[typecasting]
  c=2.6000
- float a=13,b=5,c=a/b;
  c=2.6000
- char a=13,b=5,c=a/b;
  c=2;
  it will print the character corresponding to the ascii code '2'

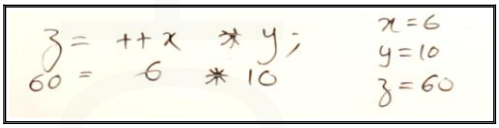
OPERATORS

- Arithmetic operations and the precendence
  Precedence: (,) ≥ *,/,% ≥ +,-
  Associativity: left to right
  eg. a+b*c-d/c
  execution: b*c,d/c,a+(b*c),(a+(b*c))-(d/c)
- Compound operators
  +=, -=, *=, /=, %=, &=, |=, >>=, <<=...
  Compound expressions are faster as compared to the normal expressions.
- Increment and Decrement operators
  int x=5;

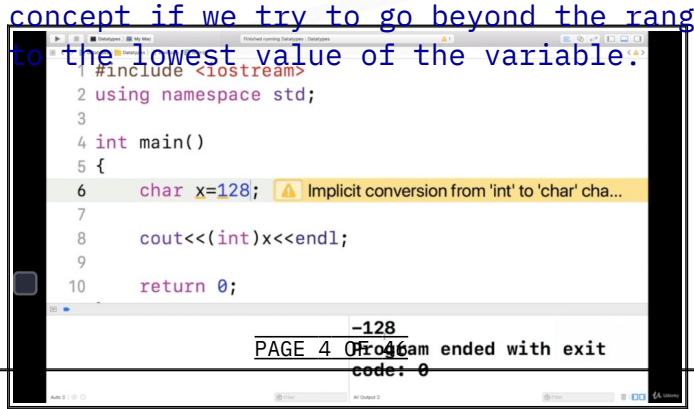


int x=5,y=10;





- The concept of Overflow
  ↳ According to this concept if we try to go beyond the range of the datatype the we cycle back to the lowest value of the variable.



- Bitwise operators
  - and(&),or(|),xor(^),not(~),leftshift(<<),rightshift(>>).
  - when using the ls or rs the sign bit is not disturbed.
  - ls(<<)=x<<i = $x*2^i$
  - rs(>>)=x>>i = $x/(2^i)$

- Enum and Typedef
- Enum: Used for defining a group of constants.
  -
```
enum day {mon, tue, wed, thu, fri, sat, sun};
        data type
int main()
{
    day d;
    d=mon;
    d=fri;
    d=0;
```

  -
```
enum day {mon=1, tue, wed=5, thur, fri, sat=9, sun}
```

- Typedef : Used for giving meaningful names to the default datatypes.
  -
```
typedef int marks;
typedef int rollno;

int main()
{
    marks m1,m2,m2;
    rollno r1,r2,r3;
```

CODITIONAL STATEMENTS

- false=0; True=any non zero value
- Short circuit:

```
if( a>b && a>c)
```

```
if( a>b || a>c)
```

- In the second condition of logical operators never use increment or decrement operator because they may or may not get executed due to short circuit.
- Dynamic declaration : The activation record of main function grow and shrink according to the need of new variables.
-

```
int main()
{
    int a,b,c,x;

    int k=exp;
    if (int k=exp; k<a)
}
```

- Limiting the visibility of a variables
  - ↳ Using a dummy block

```
int a=10,b=5;
{
int c=a+b;
if(c>10)
{

}
}
```

  - ↳ Using Decalaration

```
if(int c=a+b;c>10)
{

}
```

ARRAYS

- for each loop
  - ↳ for(int x:A)//the value of each element of the array is copied in x, We can
    cout<<x<< endl;//manipulate the value of x without changing the original
  - ↳ for(int &x:A)
    cout<<++x;//changes the original array
  - ↳

```
int main()
{
    int A[]={2,4,6,8,10,12};

    for(int x:A)
        cout<<x<<endl;

    return 0;
```

```
    float A[]={2.5f,5.6f,9,8,7};

    for(int x:A)
        cout<<x<<endl;
```

```
2
5
9
8
7
```

```
int main()
{
    float A[]={2.5f,5.6f,9,8,7};

    for(float x:A)
        cout<<x<<endl;
```

```
2.5
5.6
9
8
7
```

```
int main()
{
    float A[]={2.5f,5.6f,9,8,7};

    for(auto x:A)
        cout<<x<<endl;
```

```
2.5
5.6
9
8
7
```

```
char A[]={'A',66,'C',68};

for(auto x:A)
    cout<<x<<endl;
```

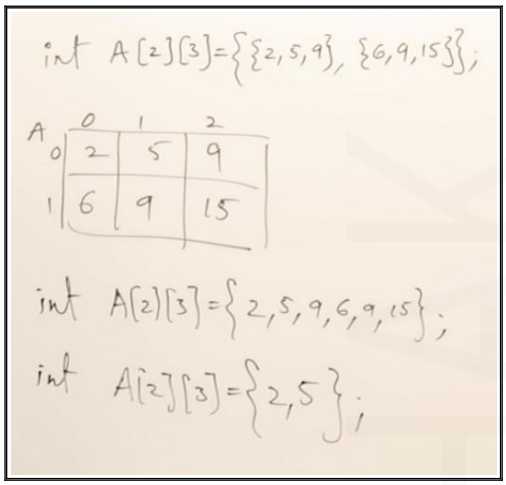
```
A
B
C
D
```

```
char A[]={'A',66,'C',68};

for(int x:A)
    cout<<x<<endl;
```

```
65
66
67
68
```

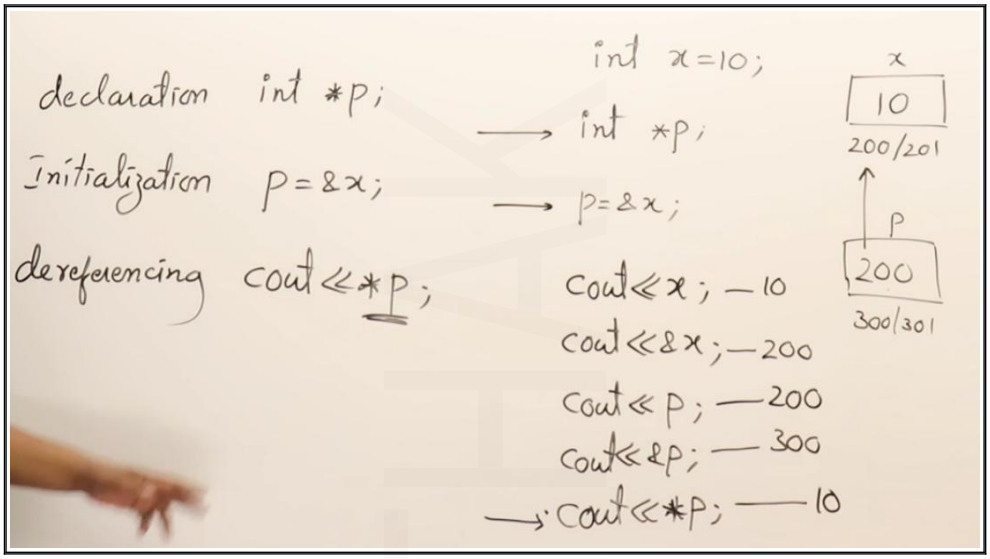
- Linear search{O(n)}
- Binary search{O(log(n))} : Sorted array required

- 2D array :

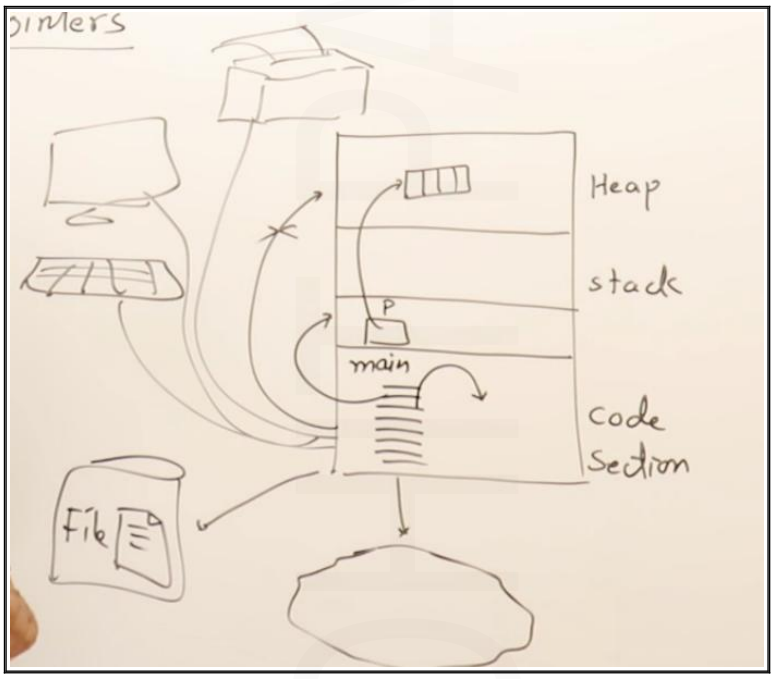


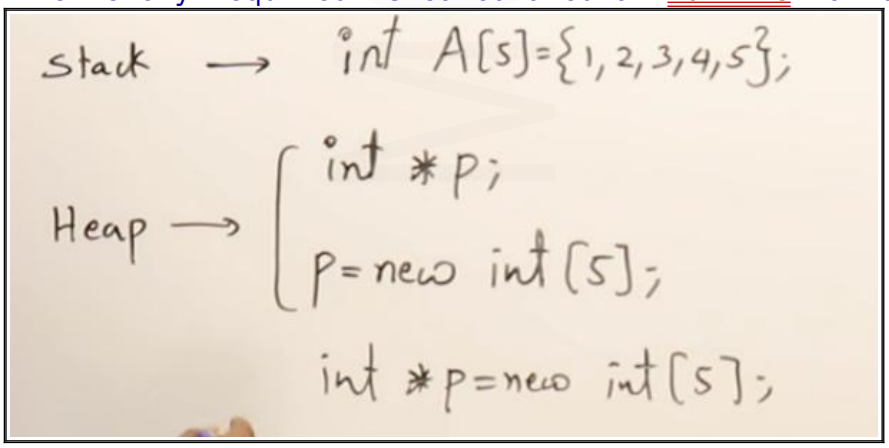
-
```
int A[2][3]={2,4,6,3,5,7};

for(auto& x:A)
{
    for(auto& y:x)
    {
        cout<<y<<" ";
    }
    cout<<endl;
}
```

POINTERS

- Variables
  - ↳ Data variable
  - ↳ Address variable
- 

- A code has direct access only to the code section and the stack area. For accessing everything else the program needs a pointer.



- Heap
  - ↳ The size of the memory required is calculated at runtime not compile time.
  - ↳ 

- Accessing the heap : p[i]=48;...
- Heap memory will persist as long as the program is running.
- deallocation
  - delete []p;
  - p=null;//p=nullptr {Pointer not pointing anywhere}
- Memory leak problem : When the ptr to heap memory is deleted before deallocating the memory.

Arrays of dynamic size:

- int size;
  cin>>size;//old size
  delete []p;//deletes the old array
  int *p=new int[size];
  cin>>size;//new size
  p=new int[size];

Pointer arithmetic

- ptr++; ptr--; ptr=ptr+k; ptr=ptr-k; variable=ptr1-ptr2(distance btw the 2 ptr)

Traversing an array

```
int A[5]{2,4,6,8,10};
int *p=A;

for(int i=0;i<5;i++)
{
    cout<<A[i]<<endl;
}
```

```
int A[5]{2,4,6,8,10};
int *p=A;

for(int i=0;i<5;i++)
{
    cout<<i[A]<<endl;
}
```

```
int A[5]{2,4,6,8,10};
int *p=A;

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

```
int A[5]{2,4,6,8,10};
int *p=A;

for(int i=0;i<5;i++)
{
    cout<<p[i]<<endl;
}
```

```
int A[5]{2,4,6,8,10};
int *p=A;

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

```
int A[5]{2,4,6,8,10};
int *p=A;

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

```
2
4
6
8
10
```

```
int A[5]{2,4,6,8,10};
int *p=A;

for(int i=0;i<5;i++)
{
    cout<<A+i<<endl;
}
```

```
int A[5]{2,4,6,8,10};
int *p=A;

for(int i=0;i<5;i++)
{
    cout<<p+i<<endl;
}
```

```
0x7ffeefbff520
0x7ffeefbff524
0x7ffeefbff528
0x7ffeefbff52c
0x7ffeefbff530
```

- A is same as A[0]

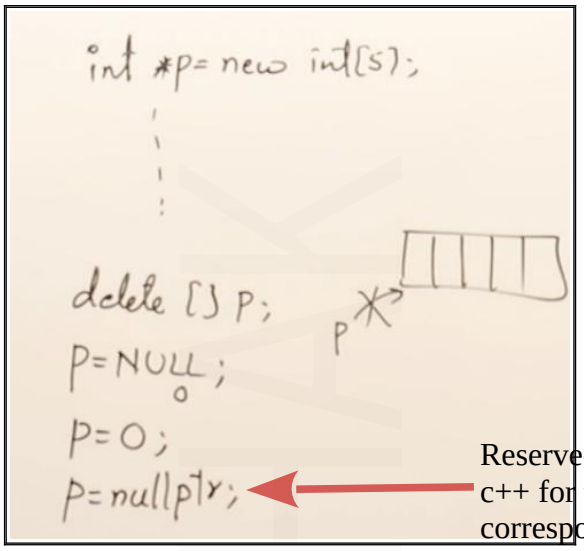
Problems in using pointers

- 1. Uninitialized pointers

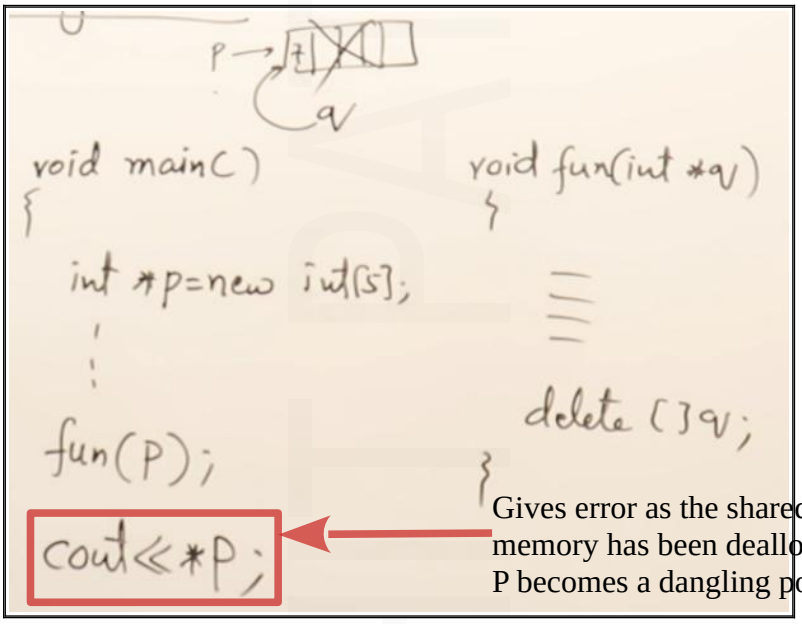
```
int x=10;
int *p;
① p=&x;
② p=(int *)0x5638;
③ p=new int[5];
```

heap memory

↳ 2. Memory leak



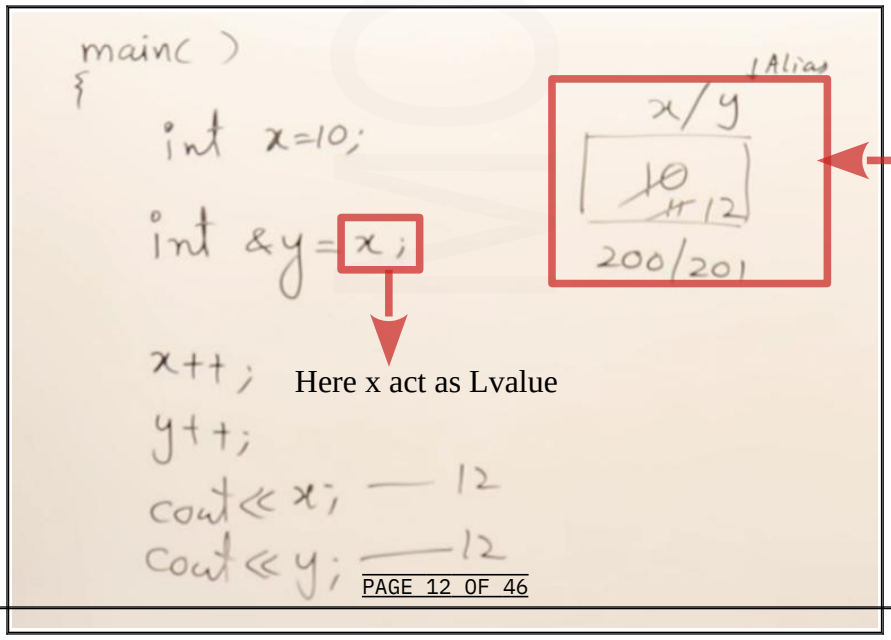
↳ 3. Dangling pointer



➲ Reference

↳ It must be initialized while declaration.

↳



↳ Referencing does not allocate any memory.
↳ lvalue : address of variable
↳ rvalue : data of variables
↳ once a variable is set as refrence it can't be used again in the program to refer to another memory.

➲ Pointer to a function

↳

```
void display()
{
    cout<<"Hello";
}

int main()
{
decl  ->  void (*fp)();
init  ->  fp=display;
call  ->  (*fp)();
```

↳

```
int max(int x,int y)
{
    return x>y?x:y;
}

int min(int x,int y)
{
    return x<y?x:y;
}

int main()
{
    int (*fp)(int,int);
    fp=max;
max is called  (*fp)(10,5);
    fp=min;
min is called  (*fp)(10,5);
```

↳ A function ptr can point to many functions which have the same signature.
↳ It is helpful in achieving runtime polymorphism using function overriding.

FUNCTIONS

- Use of cin and cout should be avoided in a called function(bad function).They should be present inside main function.
- Function overloading
  - Having multiple function with same name but different arguments
  -

```
int add(int x,int y)
{
    return x+y;
}
int add(int x,int y,int z)
{
    return x+y+z;
}

void main()
{
    int a=10, b=5, c,d;
    c=add(a,b);
    d=add(a,b,c);
```

  - Function Template
    -

```
template <class T>
T max(T x,T y)
{
    if(x>y)
        return x;
    else
        return y;
}

main()
{
    int c=max(10,5);
    float d=max(10.5f,6.9f);
}
```

```
int max(int x, int y)
{
    if(x>y)
        return x;
    else
        return y;
}

float max(float x,float y)
{
    if(x>y)
        return x;
    else
        return y;
}
```

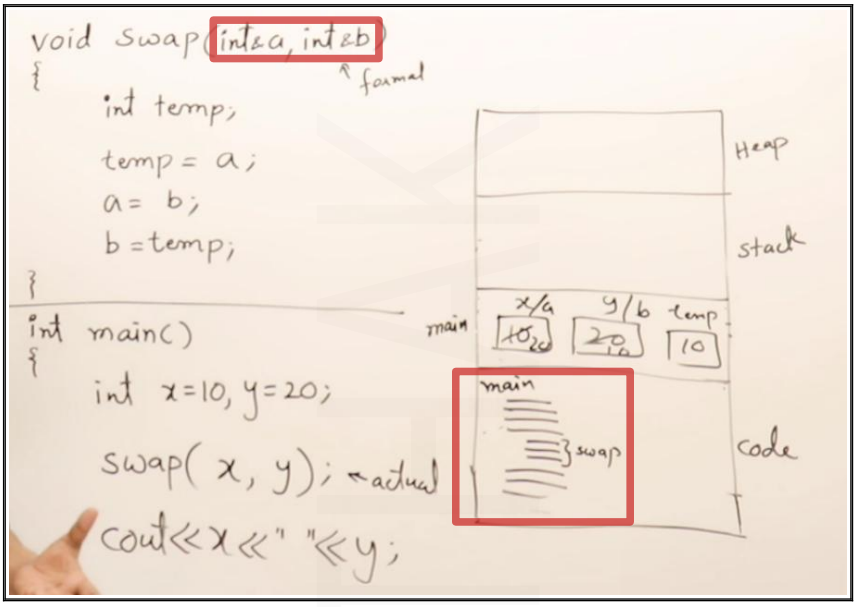
  - Default template
    -

```
int add(int x,int y,int z=0)
{
    return x+y+z;
}
int fun(int a,int b,int c,int d)

main()
{
    int c=add(2,5);
    c=add(2,5,8);
    c=add(2,5,0);
```

optional

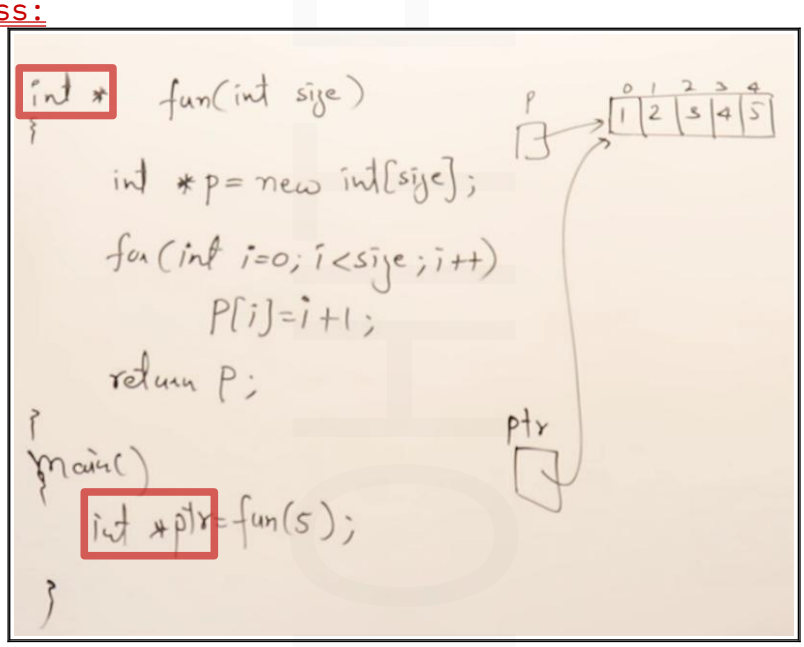
```
int add(int x,int y)
{
    return x+y;
}

int add(int x,int y,int z)
{
    return x+y+z;
}
```

- Parameter passing
  - Call by value
  - Call by address
  - Call by reference : The called function automatically becomes an Inline function.



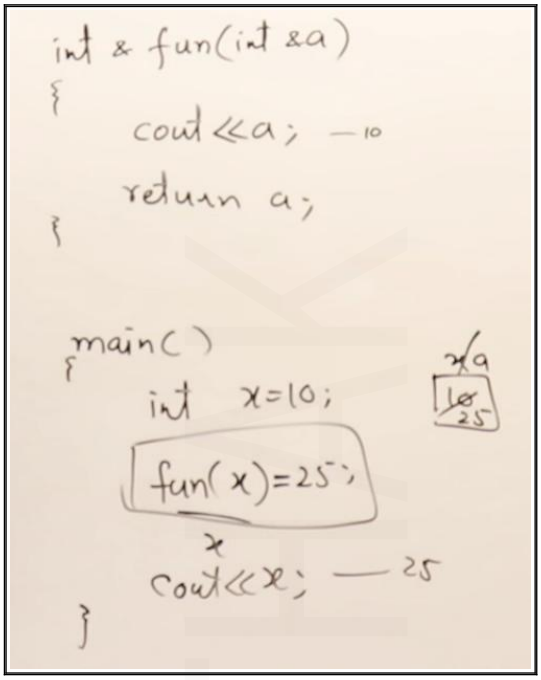
  - Complex code should not be written inside the function if the function is called as call by reference.

- Return by address:
  -



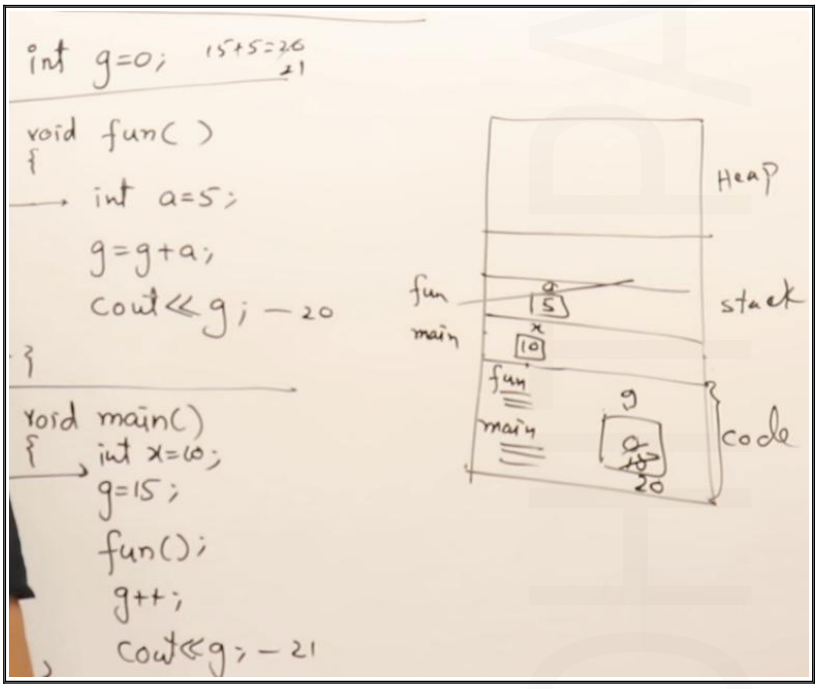
  - The variables used in the called function should be in heap memory as the variables in the stack (local variables) will be deleted as soon as the program will be finished.

Return by reference



Local and Global variables

- The memory for the global variable is allocated in code section. At loading time before the execution.



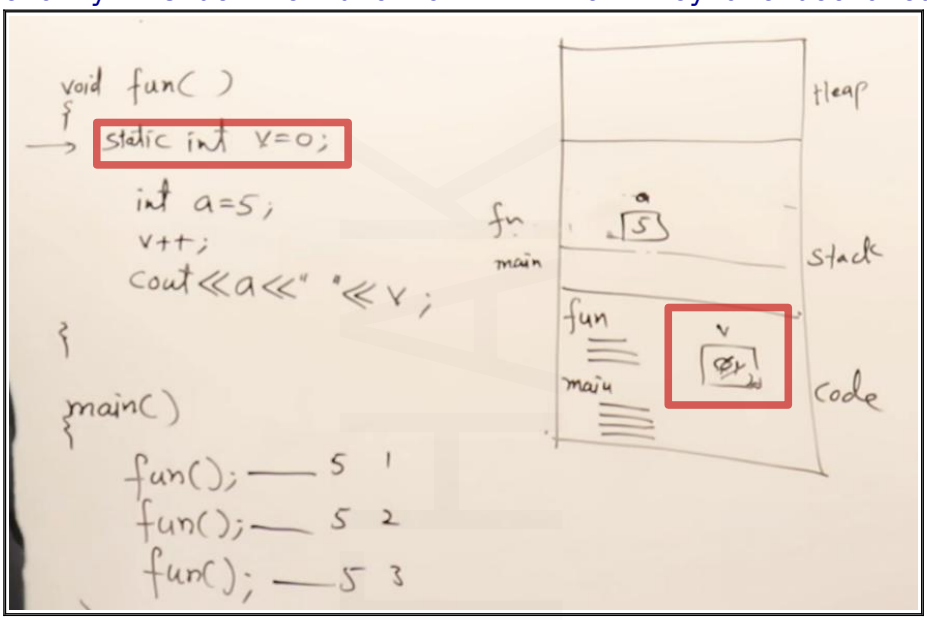

```
int x=10;
int main()
{
    int x=20;

    {
        int x=30;
        cout<<x<<endl;
    }
    cout<<x<<endl;
    cout<<::x<<endl;
```

30

20

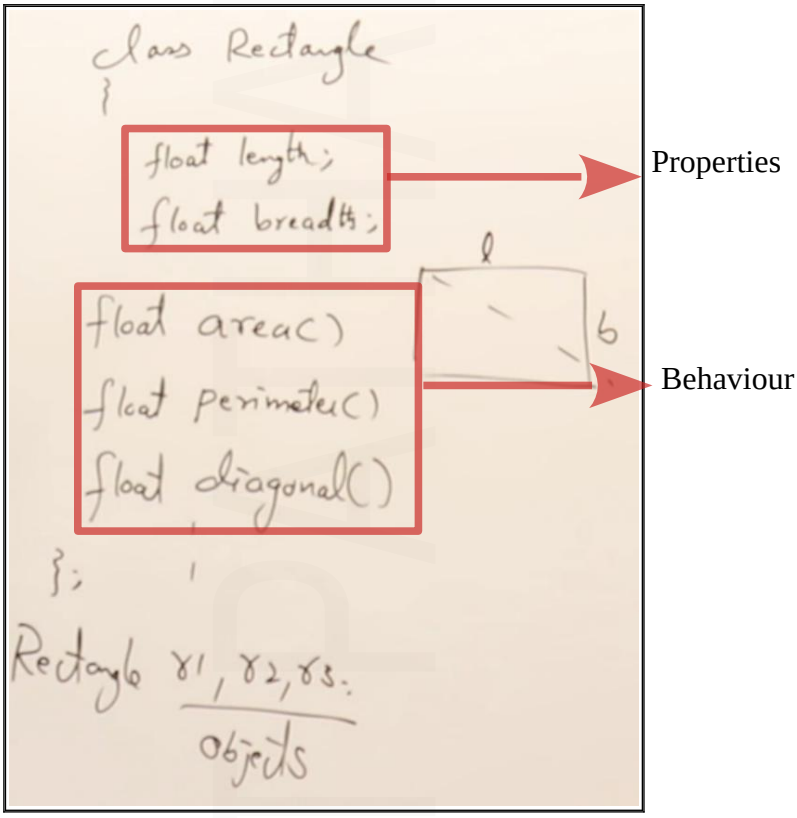
10=>global variable

➲ Static variables : The variables whose life is until the program ends but they are accessible only inside the function in which they are declared.

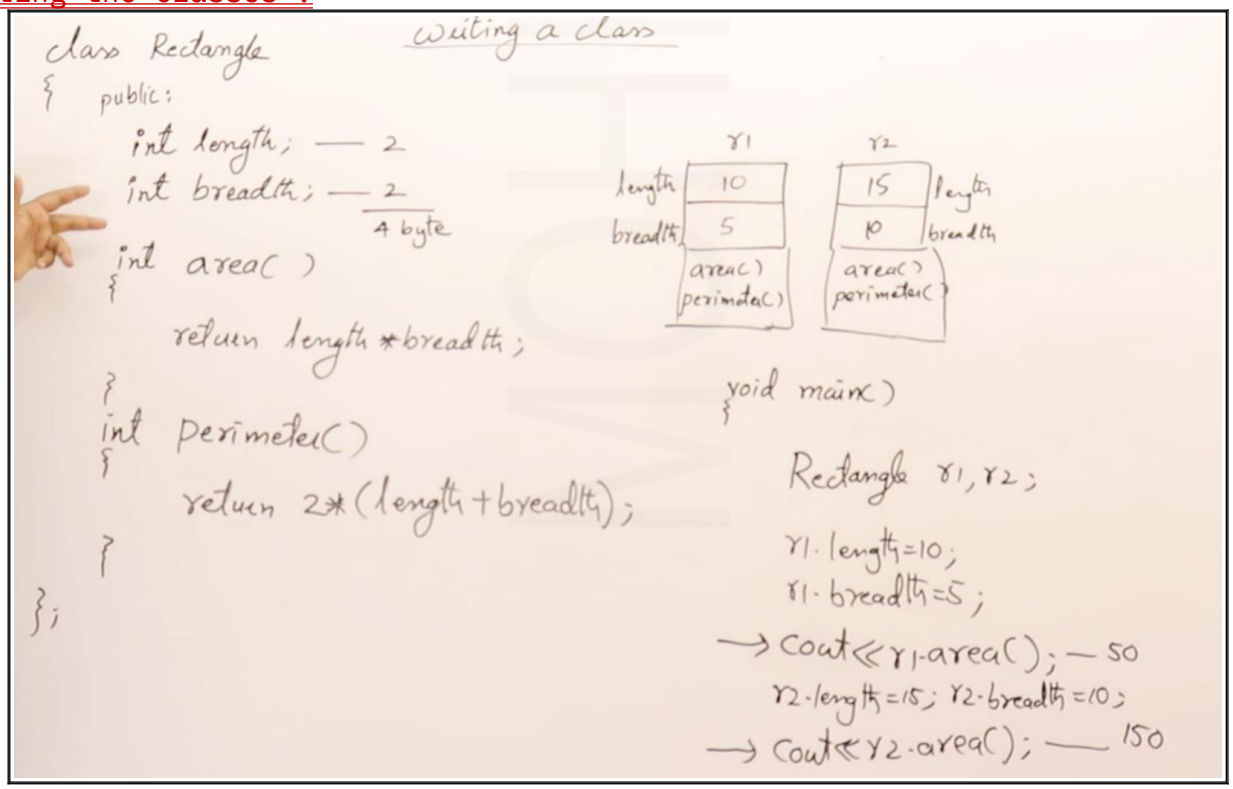


INTRODUCTION TO OOPS

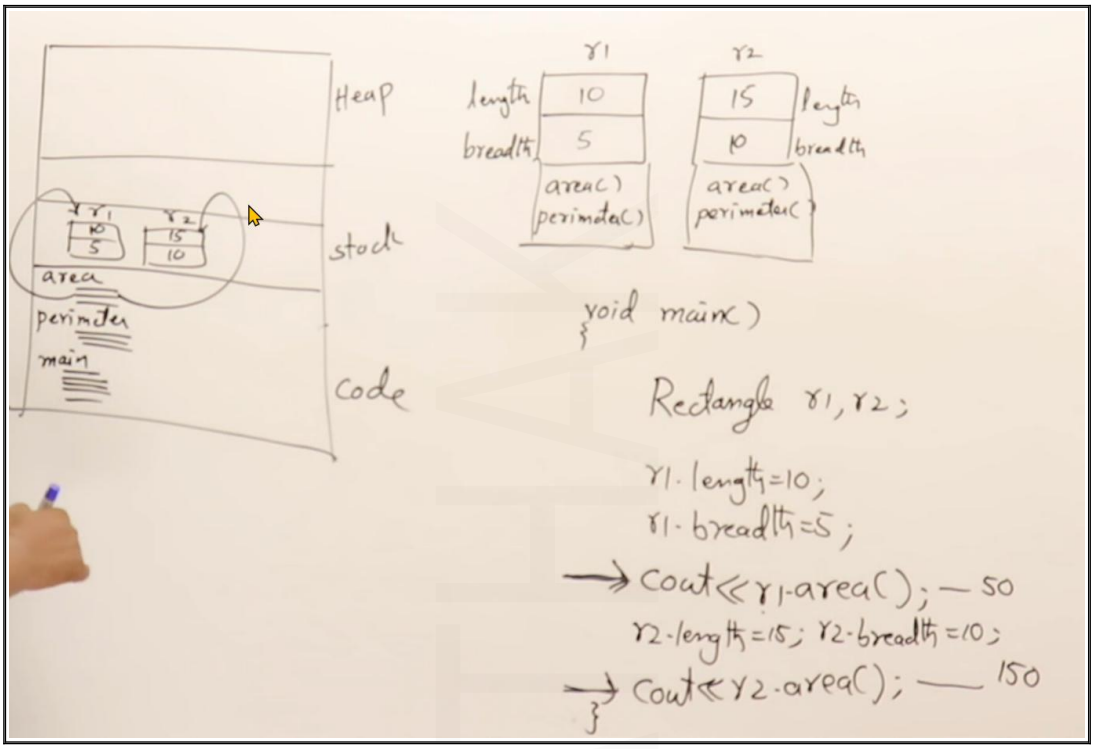
- A software is a collection of objects which contains all the relevant functions related to that object.
- Priciples of oops
  - Abstraction
  - Encapsulation
    - Data hiding
  - Inheritance
  - Polymorphism
- Classes and Objects
  - Classes:



- Writing the classes :
  -

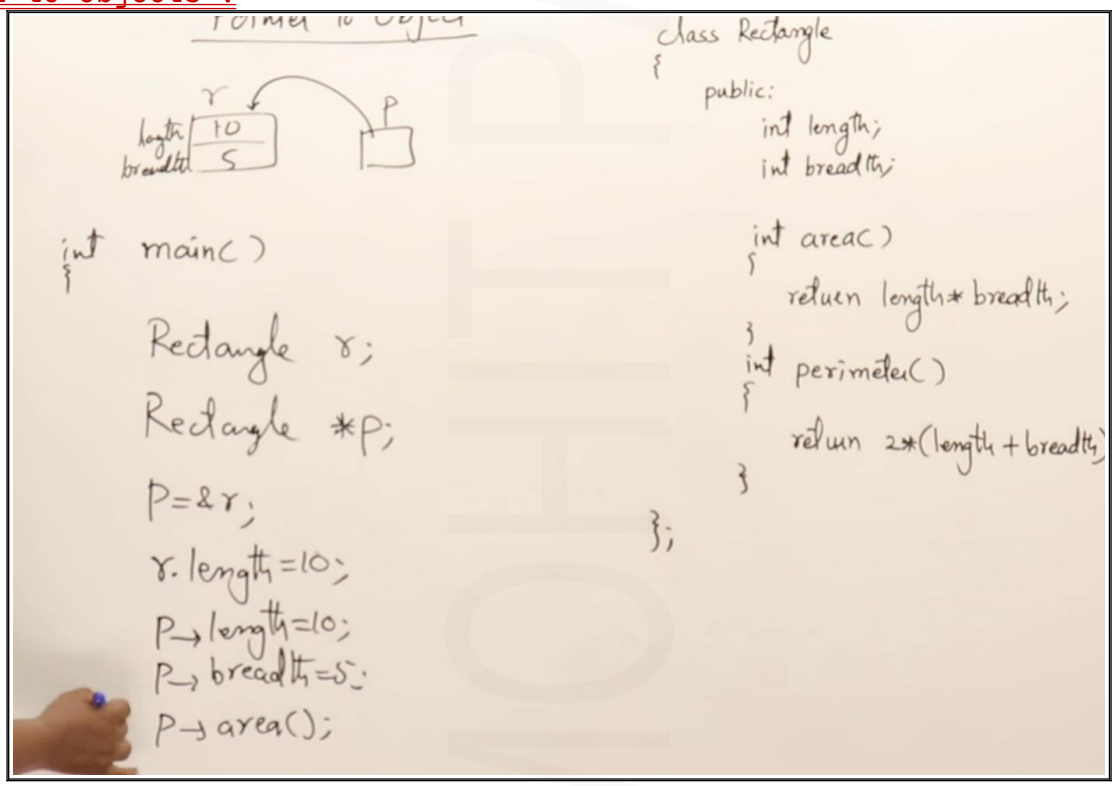


↳



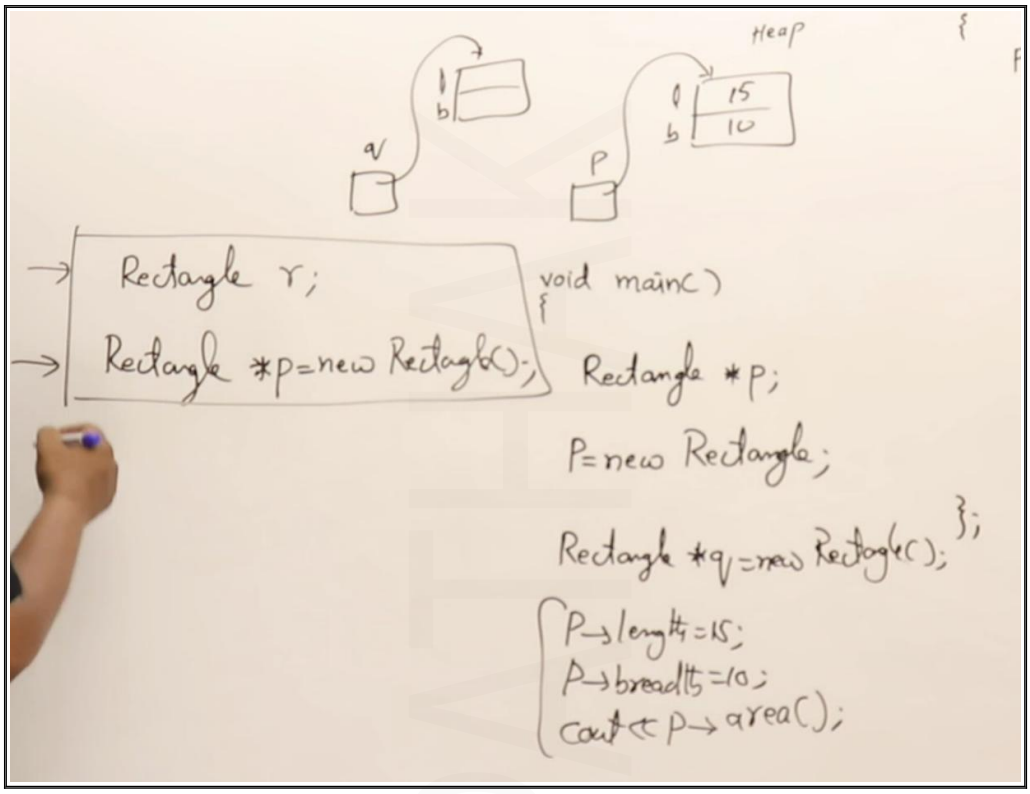
➲ Pointer to objects :

↳



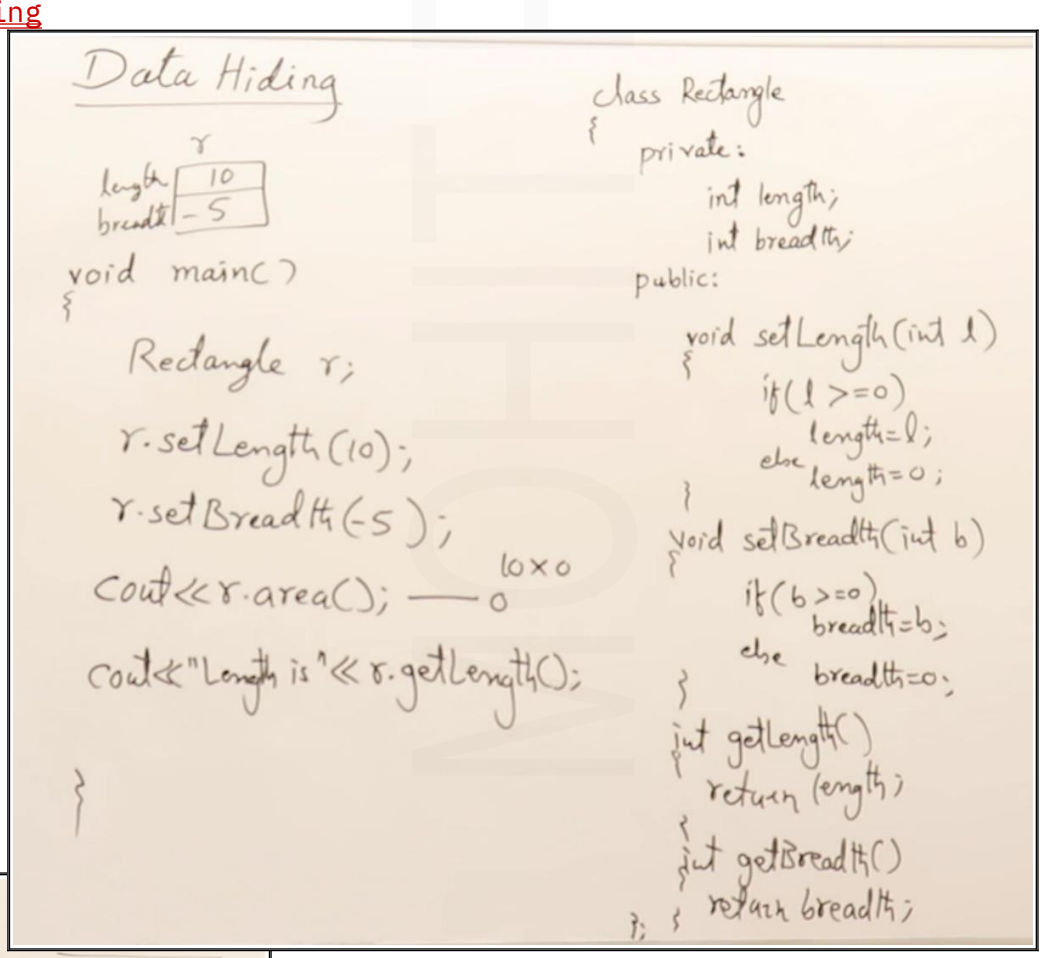
↳ Creating an object in heap

↳



➲ Data Hiding

↳

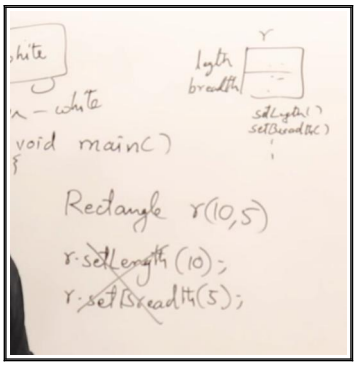


Accesor – getXXX

Mutator – setXXX

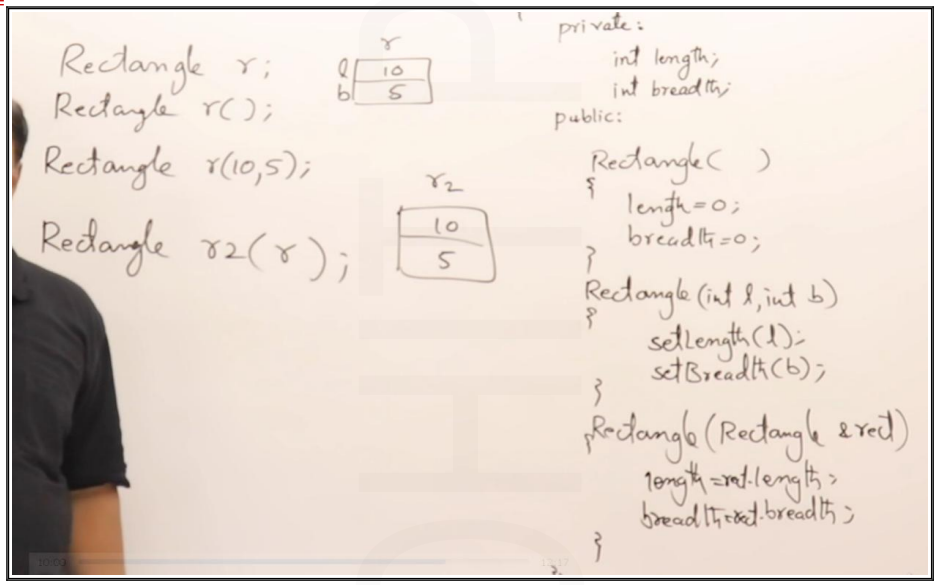
↳

CONSTRUCTORS

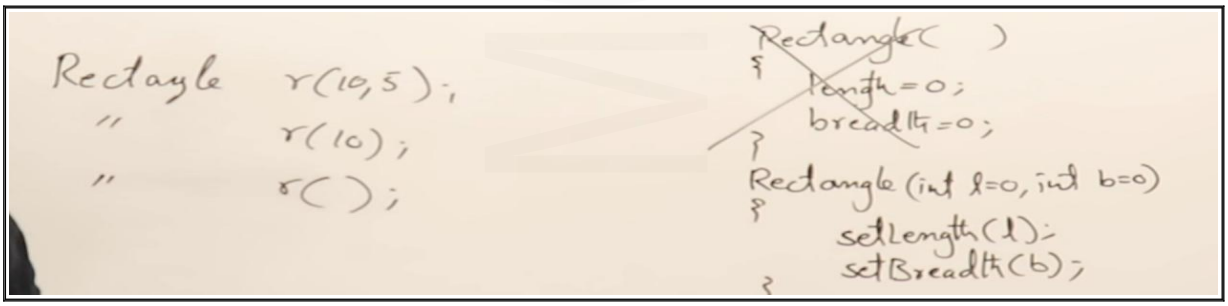
➲ 

➲ Types of constructors :
- ↳ Built-in/System(Provided by the compiler)
  - ↳ Default constructor
- ↳ User defined:
  - ↳ Paramterized
  - ↳ Non-parametrized
  - ↳ copy constructor

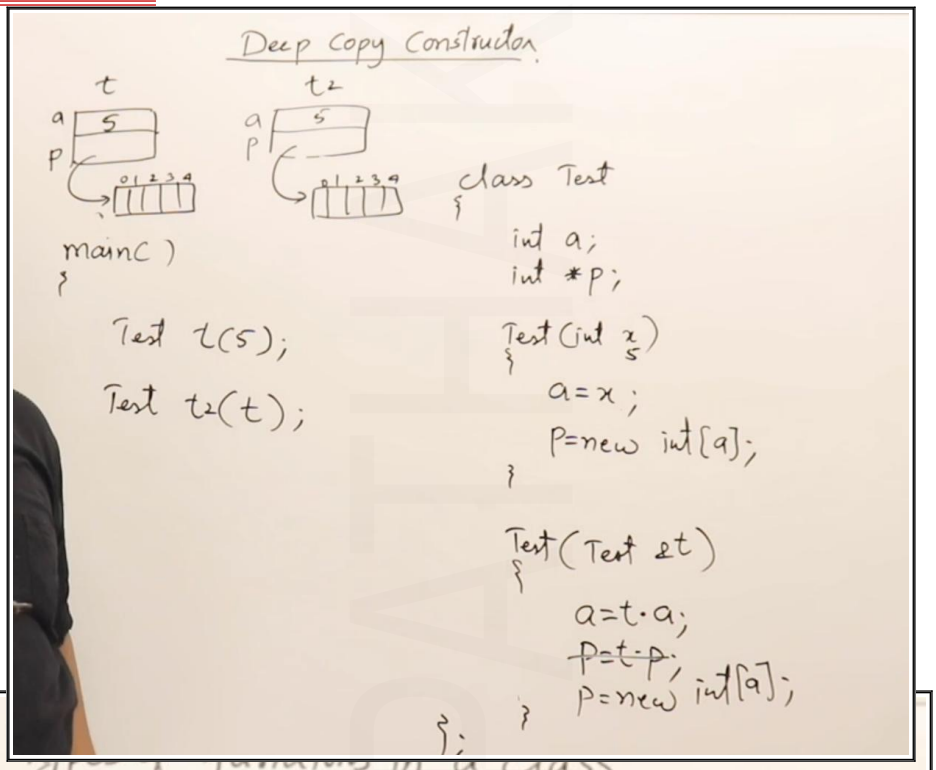
➲ Constructors are functions that have the same name as the class but have no return type.

➲ 

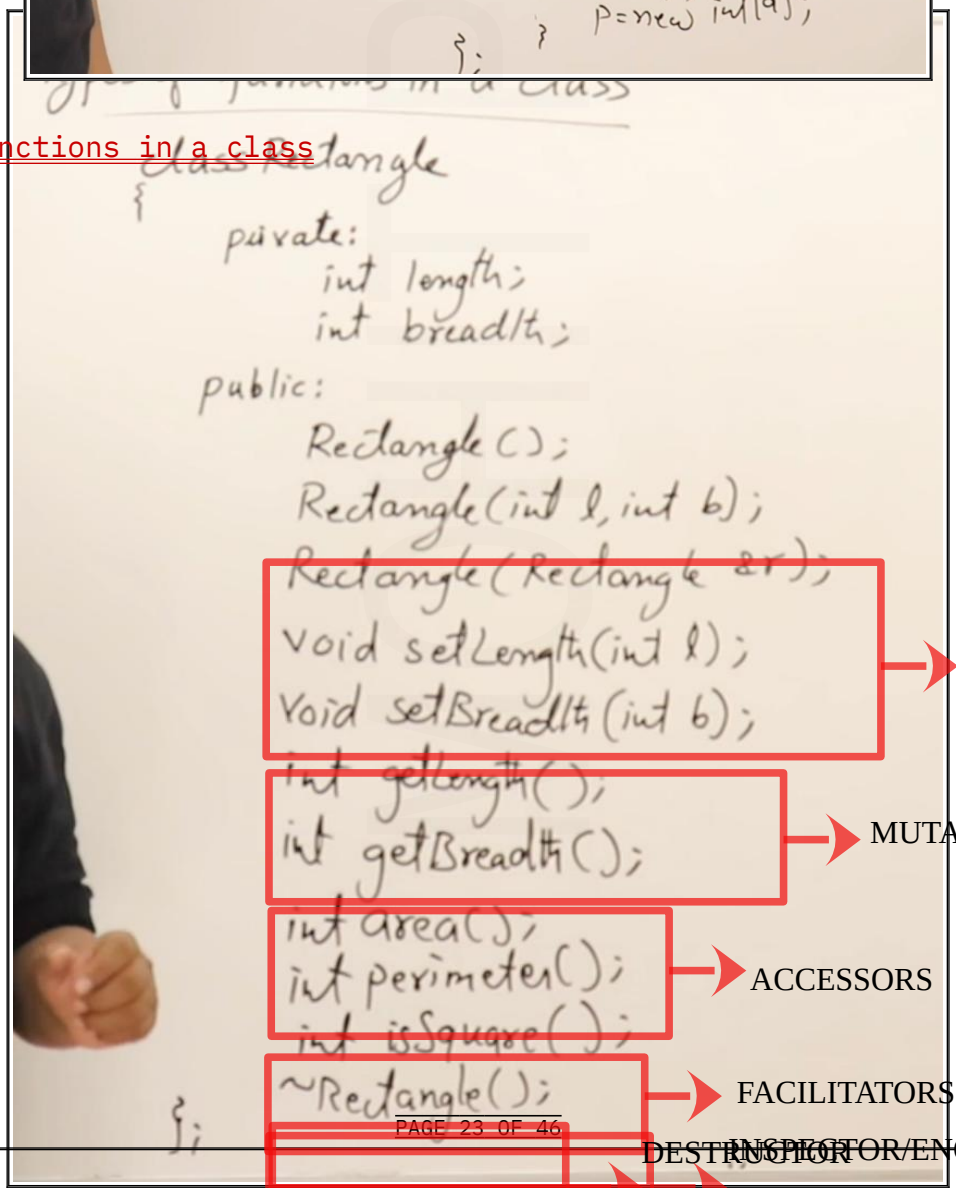
➲ Using default arguments

↳ 

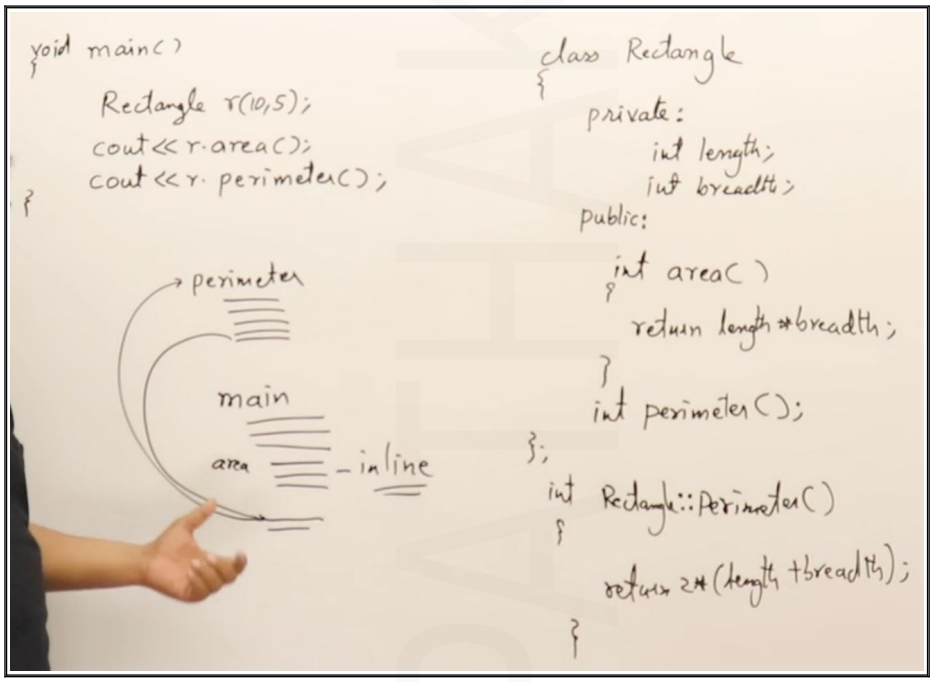
➲ Deep copy constructor : used when there is dma.
↳



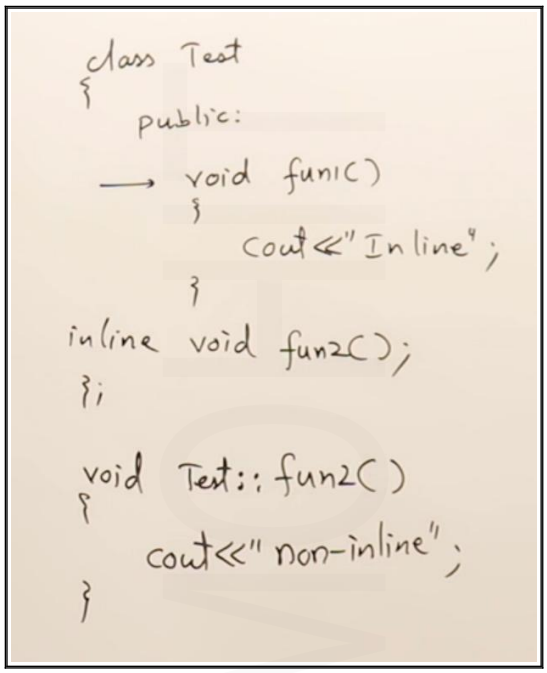
➲ Types of functions in a class
↳



- Methods to write fucntions in a class
  ↳



- Inline function
  ↳



- this
  ↳ used to access the members of the current object
  ↳

```
class Rectangle
{
private:
    int length;
    int breadth;
public:
    Rectangle(int length,int breadth)
    {
        this->length=length;
        this->breadth=breadth;
```

➲ classes vs struct

↳ Bydefault in class all the things are private and in stuct all the thing are public.

OPERATOR OVERLOADING

```
Complex operator+(Complex c)
{
    Complex temp;
    temp.real=real+c.real;
    temp.img=img+c.img;
    return temp;
```

CALL : c=c1+c2;

Friend operator overloading

```
class Complex
{
    private:
        int real;
        int img;
    public:
    friend Complex operator+(Complex c1,complex c2)
    };

Complex operator +(Complex c1,Complex c2)
{
    Complex t;
    t.real = c1.real +c2.real;
    t.img = c1.img +c2.img;
    return t;
}
```

CALL : c=c1+c2;

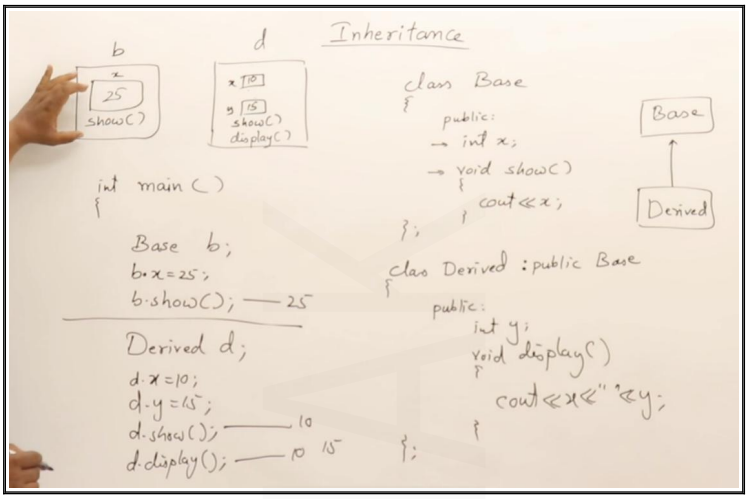
Overloading insertion and exertion operators

```
7)      class Complex
        {
            private:
                int real;
                int img;
            public:
friend ostream & operator<<(ostream &o,Complex &c1)
        };

ostream & operator<<(ostream &o,complex &c1)
{
        o<<c1.real<<"+i"<<c1.img;
        return o;
}
```

CALL : cout<<c;
OR
operator<<(cout,c)

INHERITANCE

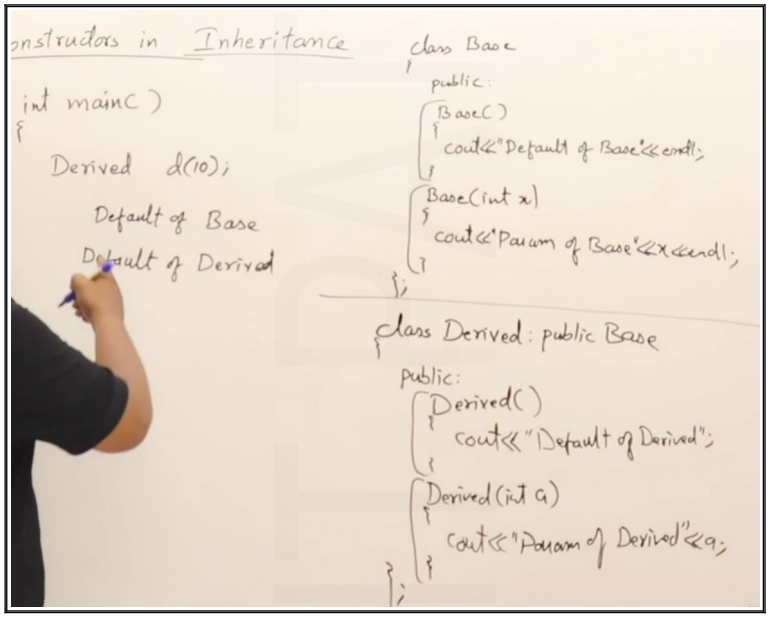
➲



➲ Constructors in Inheritance

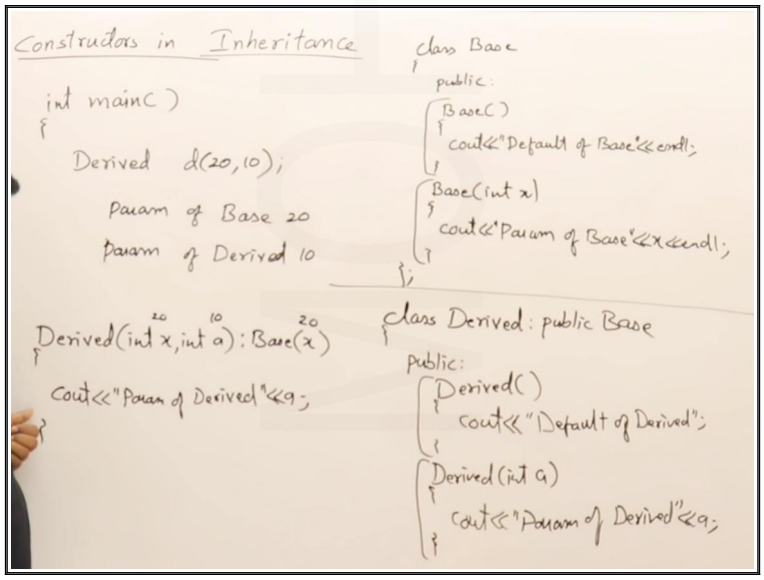
➲ Always the default constructor of base class is executed first.

↳



➲ To execut e the parametrized constructor of base class first

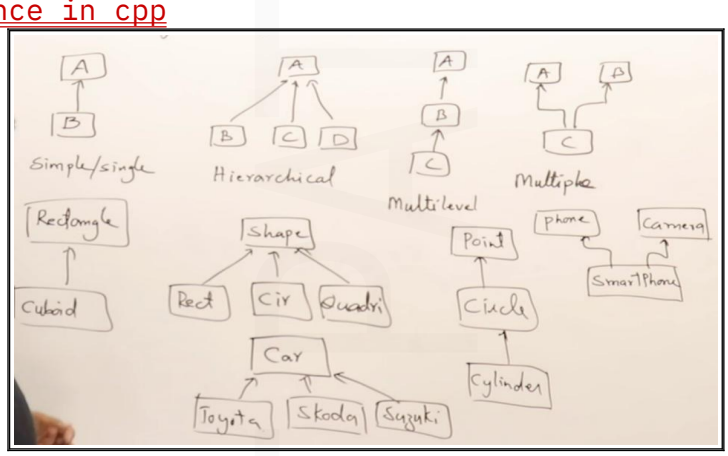
↳



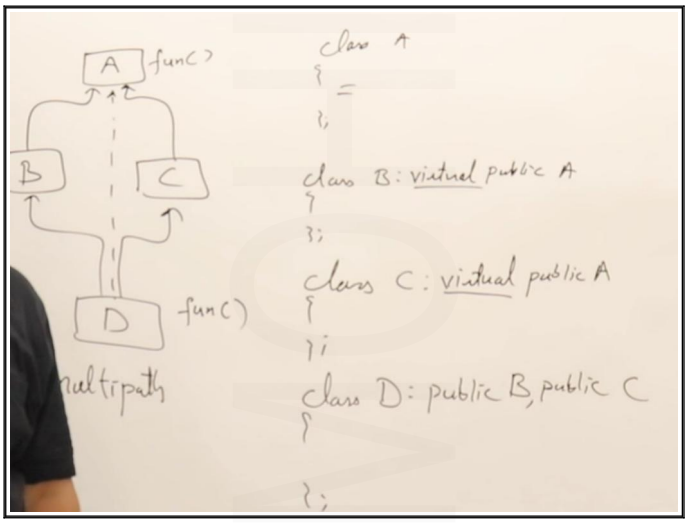
- A class can have a isA and hasA relationships.
- Access specifiers :
  - Private : Accessible only to the parent base class.
  - Protected : Accessible to derived class also.
  - Public : Accessible to everyone
  -

	private	protected	public
inside class	✓	✓	✓
inside Derived class	✗	✓	✓
On Object	✗	✗	✓

- Types of Inheritance in cpp
  -

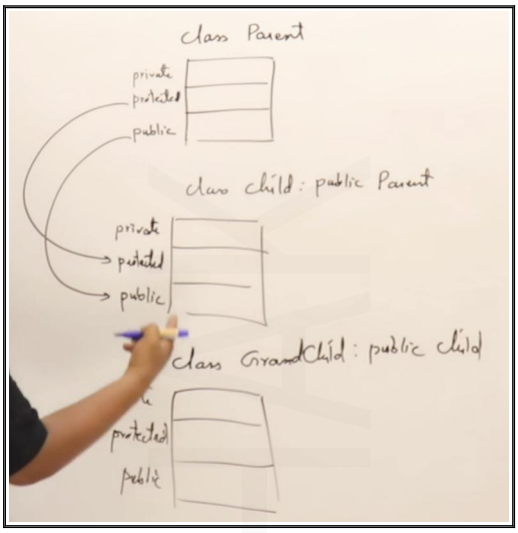


  - Virtual base classes : Used to remove ambiguity in case of Multipath inheritance.
    -



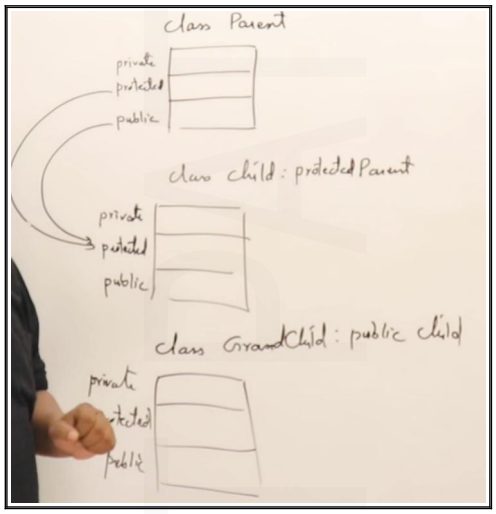
↳ Inheriting publically

↳



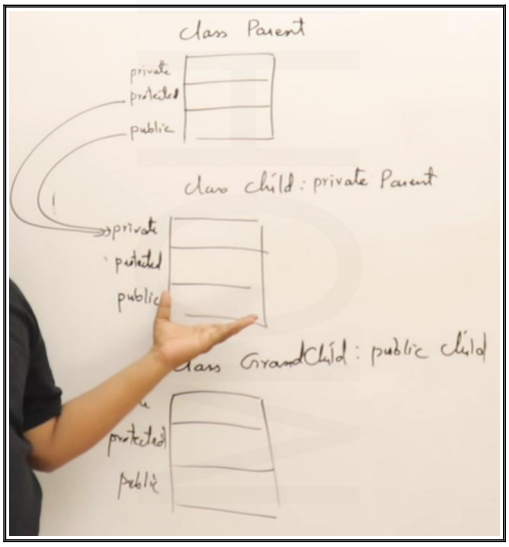
↳ Inheriting protectedly

↳



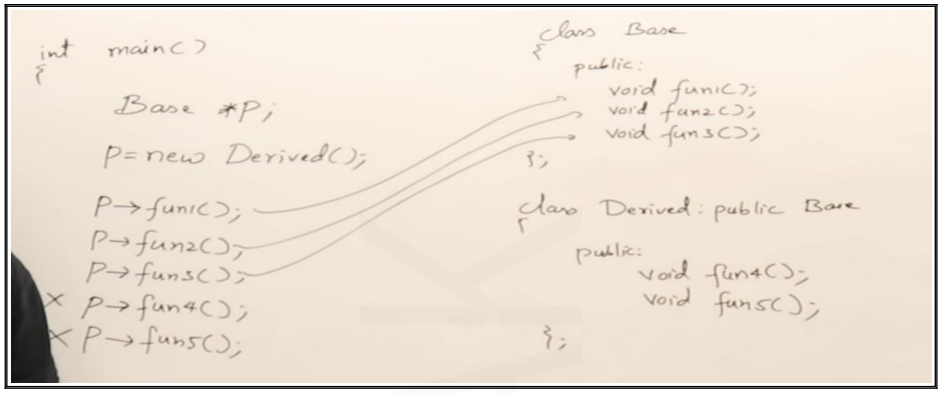
↳ Inheriting privately

↳



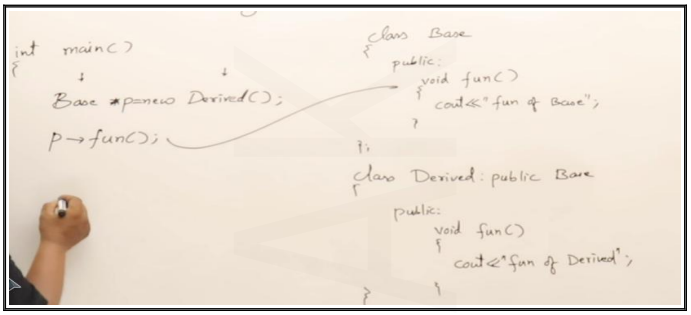
Base class pointer and derived class objects

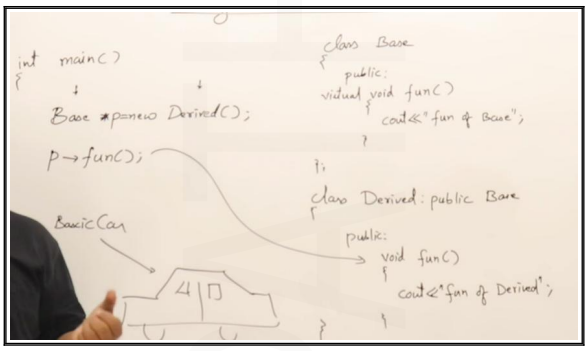
➲



➲ It is not possible to assign the address of a derived class to the base class' pointer.

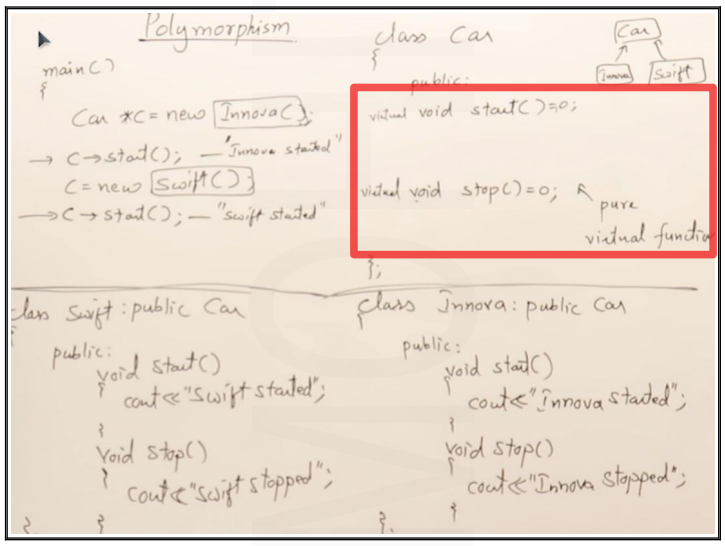
Function overriding

- Function with same prototype if present in derived class is given precedence over the function in base class
- Calling overrided function using base pointer.
  - 

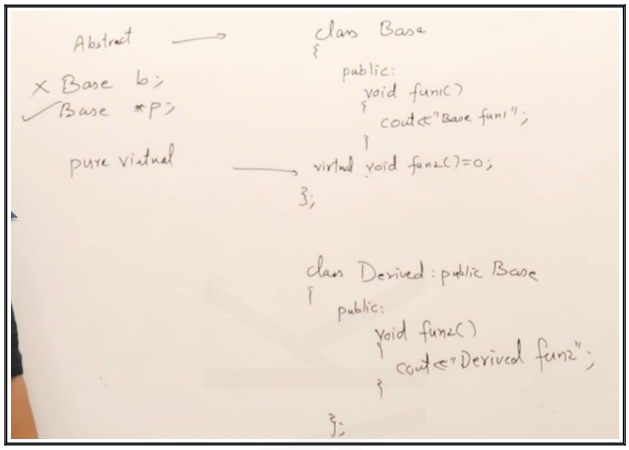
  - Using virtual functions
    - 

    - It is runtime polymorphism.

Polymorphism

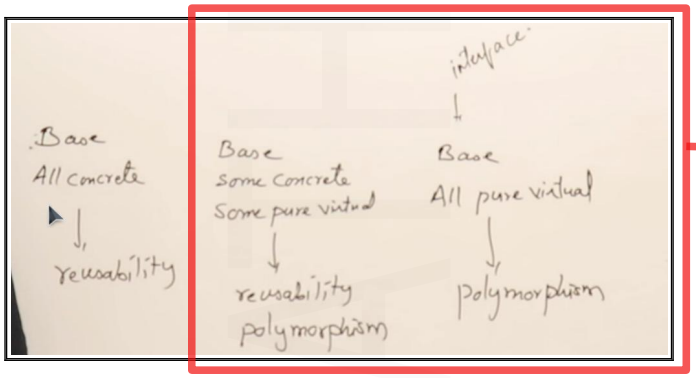
- Pure virtual functions and polymorphism(abstract class)
  - 

↳



↳ Types of classes

↳



ABSTRACT

Friend functions

➲ Friend function

```
class Test
{
    private:
        int a;
    protected:
        int b;
    public:
        int c;
friend void fun();
};

void fun()
{
    Test t;
✓ x t.a=15;
✓ x t.b=b;
✓ t.c=5;
}
```

➲ Friend class

```
class Your;

class My
{
private:int a;
protected: int b;
public: int c;
    friend Your;
};
class Your
{
public:
    My m;
    void fun()
    {
        m.a=10;
        m.b=10;
        m.c=10;
    }
};
```

➲

```
class Test
{
    private:
        int a;
        int b;
    public:
        static int count;

        Test()
        {
            a=10;
            b=10;
            count++;
        }
};

t1                t2
a | 10            a | 10
b | 10            b | 10

count
| 2

main()
{
    Test t1;
    Test t2;
    cout<<t1.count; — 2
    cout<<t2.count; — 2
    cout<<Test::count; — 2

→ int Test::count=0;
```

➲

```
class Test
{
    private:
        int a;
        int b;

    public:
        static int count;

    Test()
    {
        a=10;
        b=10;
        count++;
    }
    static int getcount()
    {
        a++;
        return count;
    }
};
int Test::count=0;

main()
{
    cout<<Test::getcount();-0
    Test t1;
    cout<<t1.getcount();
```

➲ Nested classes

↳

```
class Outer
{
    public:
        ① int a=10;
        ② static int b;

        void func()
        {

        }
        class Inner
        {
            public:
                int x=25;
                void show()
                {
                    cout<<a;
                }
        };
        ③ Inner i;
};
int Outer::b=20;
```

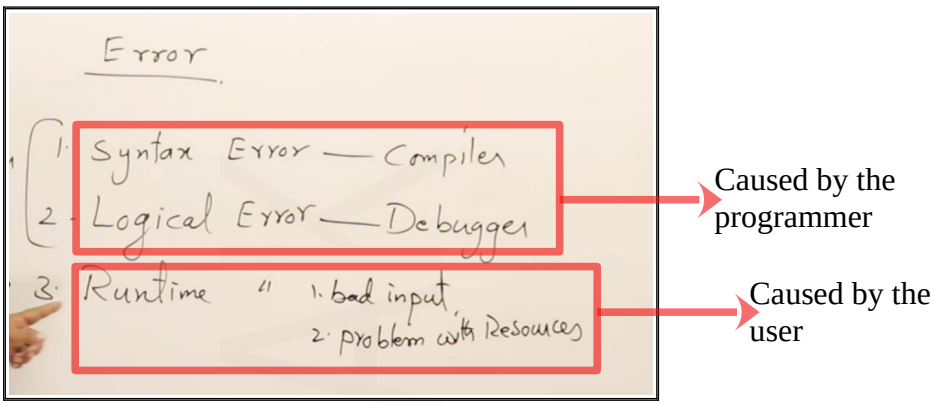
Outer class can access only the public members of the inner class.

Can't be accessed, Only the static members can be accessed

Exception handling

- Errors

  ↳



- We can throw :

  ↳ int, float, double, char, string, class

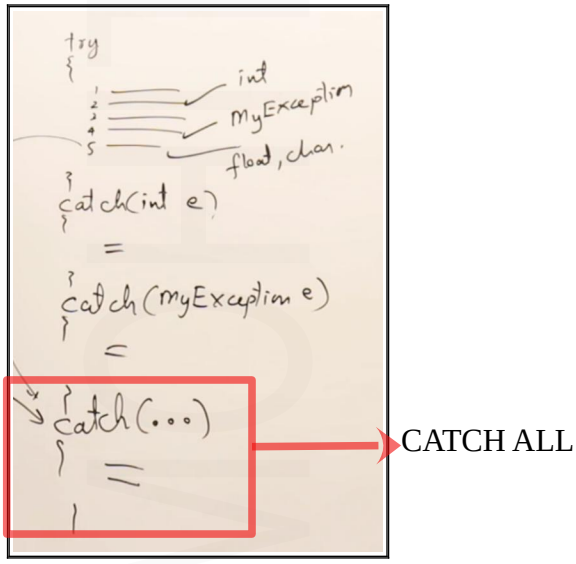
  ↳

```
class MyException : public exception
{
};
int division(int x, int y) throw(int)
{
    if(y==0)
        throw 1;
    return x/y;
}
```

```
class MyException : public exception
{
};
int division(int x, int y) throw(MyException)
{
    if(y==0)
        throw MyException;
    return x/y;
}
```

- Multiple catch blocks

  ↳



  ↳ catch all block must be the last block

➲ There can be nested try..catch blocks

↳

```
try
{
    1 ——
    2 ——
    try
    {
        3 ——
        4 ——
    }
    catch(——)
    {
    }
}
catch( )
{

}
```

↳ First the child catch block must be there before the parent catch block.

↳

```
class MyException1
{
};
class MyException2 : public MyException1
{
};
------------------------------------
try
{
    ———
}
catch(MyException2 e)
{
    ——
}
catch(MyException1 e)
{
    ——
}
```

Templates

➲ To make a function generic we use templates

↳ In case we need to use the function for classes then we need to overwrite the required operators to perform the specific operations.

↳

```
template <class T>
T maximum(T x, T y)
{
    return x>y?x:y;
}

maximum(10,15);
"  (12.5,9.5);
```

↳

```
                    int      double
template<class T, class R>
void add(T x, R y)
{
    cout<< x+y;
}

add(10,12.9);
```

↳ Template class

↳

```
Templates.
template <class T>
class stack
{
    private:
        T s[10];
        int top;
    public:
        void push(T x);
        T pop();
};

template<class T>
void stack<T>::push(T x)
{
    =
}

template<class T>
T stack<T>::pop()
{
    =
}

Stack<int> S;
Stack<float> S2;
```

METHODS

Constants, Preproccessor directive

➲ Constant

↳

```
# define x 10

int main( )
{
 const int x=10;
   ×x++;
   cout<<x;
```

x: 10 (200/201)

➲ Constant integer pointer

↳

```
# define x 10

int main( )
{
    int x =10;
const int *ptr=&x;
  ×++ *ptr;
    cout<<*ptr;
```

x: 10 (200/201)

ptr: 200

➲ Constant pointer to an integer

↳

```
# define x 10

int main( )
{
    int x =10;
int * const ptr=&x;
    int y=20;
  ×ptr=&y;
   ++(*ptr);
```

x: 10 (200/201)

y: 20 (300/301)

ptr: 200

➲ Constant pointer to constant integer

↳

```
# define x 10

int main( )
{
    int x =10;
cont int * const ptr=&x;
    ~~int y=20;~~
  ×ptr=&y;
   ++(*ptr);
```

x: 10 (200/201)

y: 20 (300/301)

ptr: 200

× ++(*ptr)

ptr=&y;

➲ Restricting a function in a class

```
class Demo
{ public:
    int x=10;
    int y=20;
    void Display( ) const
    {
    × x++;
      cout<<x<<" "<<y<<endl;
    }
};
int main( )
{
    Demo d;
    d.Display();
```

// 20

➲ Using constant arguments

↳

```
void fun(const int &x, int &y)
{
    x++;
    cout<<x<<" "<<y;        // 20
}

int main()
{
    int a=10, b=20;
    fun(a, b);
}
```

a/x 10 11 b/y 20

➲ Preprocessor directives

↳

```
#define PI 3.1425        — Symbolic
#define c cout           constants

int main()
{
    cout<<PI;
        3.1425
    cout<<10;
}
```

↳

```
#define SQR(x) (x*x)
#define MSG(x) #x
                "x"

int main()
{
    cout<<SQR(5);
        5*5
    cout<<MSG(Hello);
        "Hello"
```

↳ #ifndef....#endif

↳

```
#ifndef
    #define PI 3.1425
#endif
```

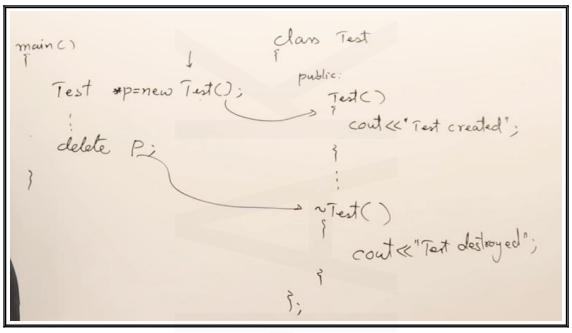
➲ Namespaces

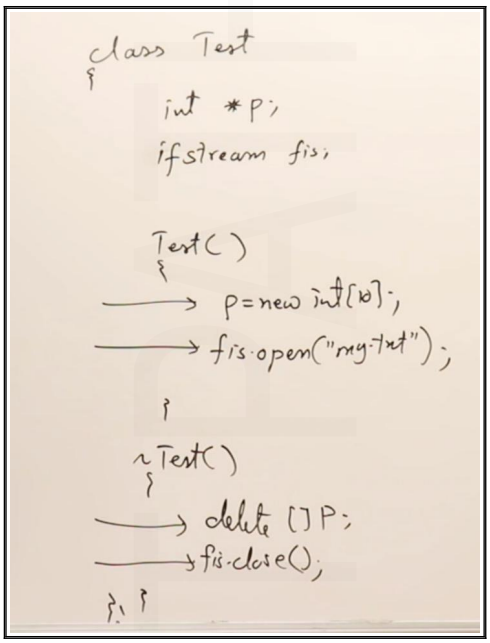
↳

```
namespace First
{
    void fun()
    {
        cout<<"First";
    }
}

namespace Second
{
    void fun()
    {
        cout<<"Second";
    }
}

int main()
{
    First::fun();
}
```

```
using namespace First;
int main()
{
    fun();
    Second::fun();
}
```

Destructors

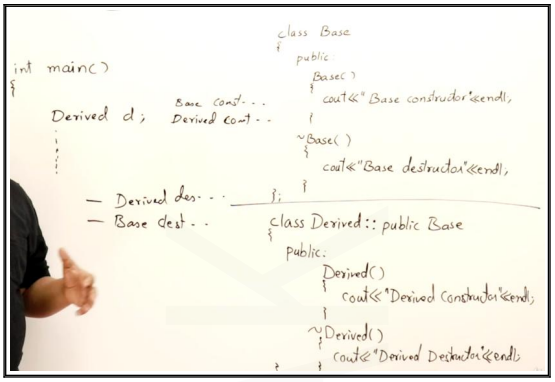
- Destructor
- Destructor in Inheritace
- Virtual Destructor
- 

- 

- The destructor function is not called if the memory is allocated in heap, It is called only when the delete ptr; statement is encountered.
  - ↳
```
void fun()
{
    Demo *p=new Demo();
    delete p;
}
```
  - ↳
```
class Demo
{
    int *p;
public:
    Demo()
    {
        p=new int[10];
        cout<<"Constructor of Demo"<<endl;
    }
    ~Demo()
    {
        delete []p;
        cout<<"Destructor of Demo"<<endl;
    }
```

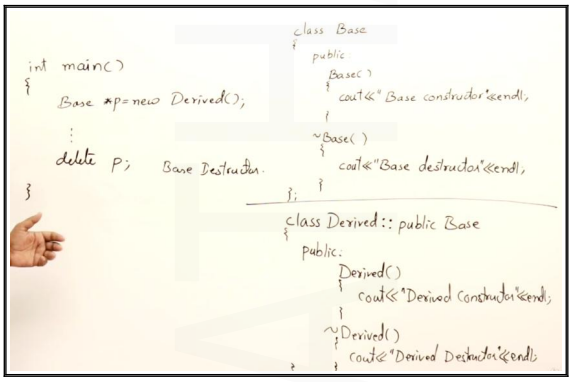
➲ Destructors in Inheritance

↳

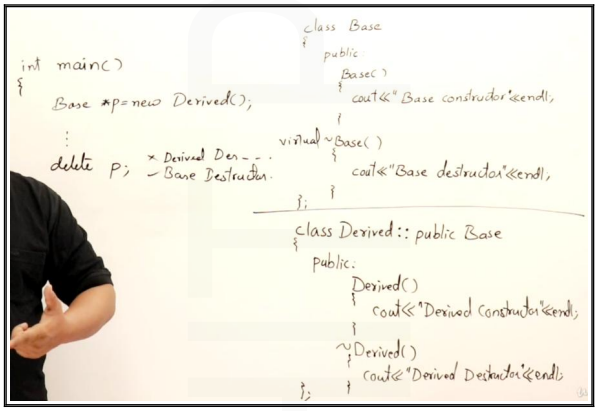


➲ Virtual Destructors

↳

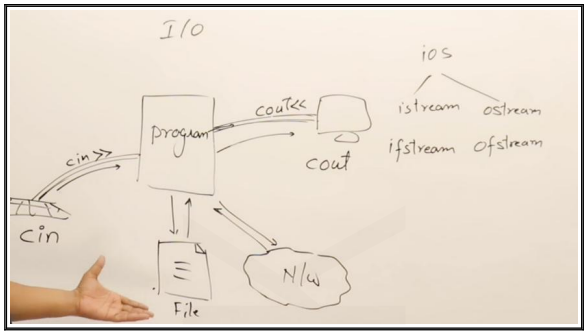


↳



Streams

➲



➲ File handling

↳ Writing to a file

```
#include <fstream>
int main()
{
  ofstream outfile("my.txt");
  outfile << "Hello" << endl;
  outfile << 25 << endl;
  outfile.close();
}
```

My.txt
Hello
25

("my.txt",ios::app) to append

↳

```
1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main()
7  {
8      ofstream ofs("My.txt",ios::trunc);
9      ofs<<"John"<<endl;
10     ofs<<25<<endl;
11     ofs<<"cs"<<endl;
12
13     ofs.close();
14 }
```

Object of class ofstream

➲ Reading from a file

↳

```
#include <fstream>
int main()
{
  ifstream infile;
  infile.open("my.txt");
  string str;
  int x;
  infile >> str;
  infile >> x;
  cout << str << " " << x;
  if(infile.eof()) cout << "end of file reached";
  infile.close();
}
```

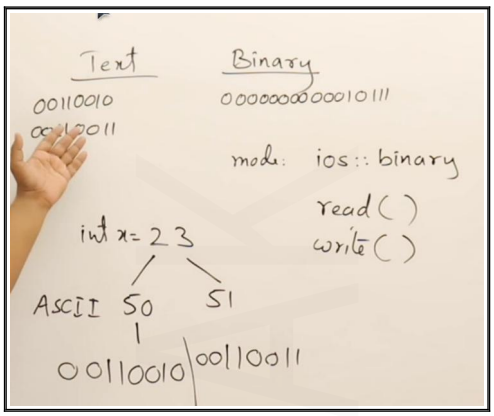
Name of the object

My.txt
Hello
25

infile : gives true when the file is opened
infile.is_open() : gives true if the file is opened

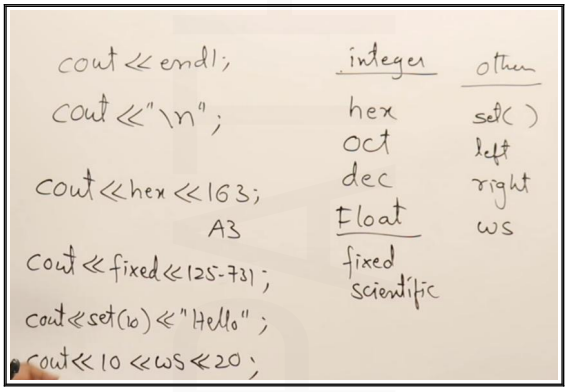
- Serialization : overloading of operators to read or write objects into a file.
- Text vs. Binary files

  ↳

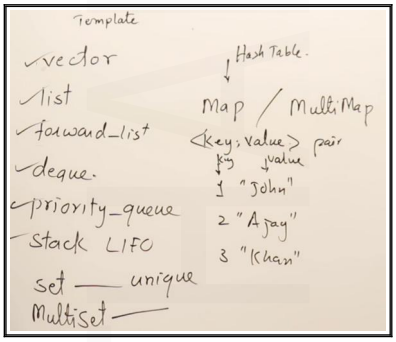


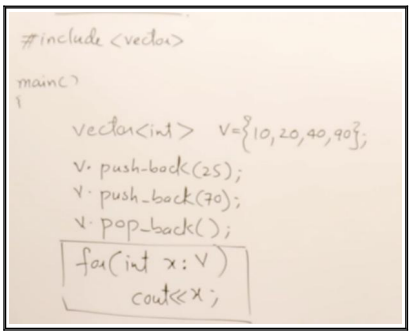
- Manipulators

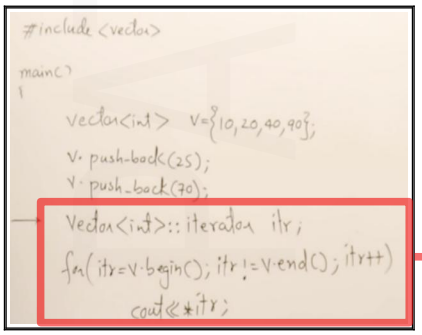
  ↳



STL

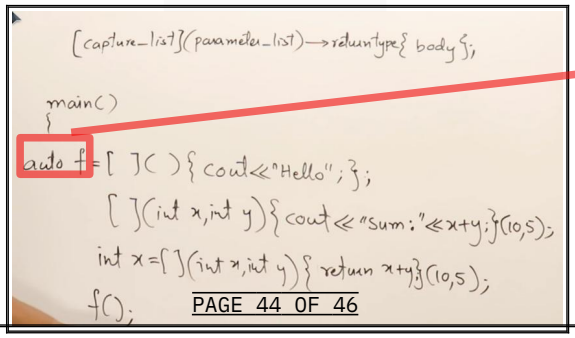
- STL contains :
  - Algorithms
  - Containers
  - Iterators
- Containers
  - 

- 

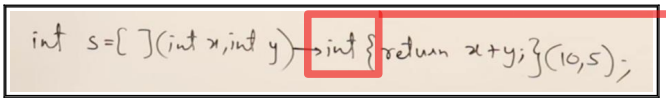


ITERATORS

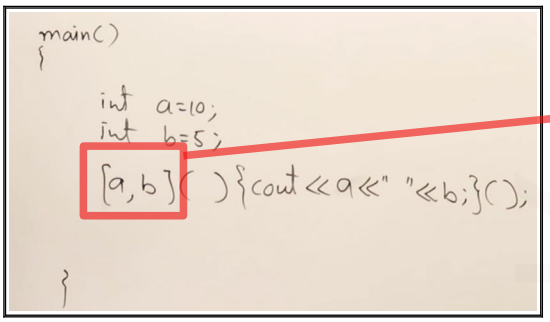
Some keywords

- auto, decltype()
- final
  - Restricts inheritance
  - Only virtual functions can be marked as final and a final marked function cannot be overrided.
- Lambda expression : used to define unnamed functions
  - syntax : [capture_list](parameter_list)→returntype{ body };
  - 

REFERENCE TO THE FUNCTION

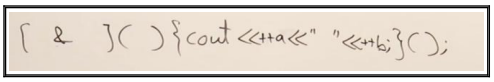
↳ 

RETURN TYPE

↳ 

CAPTURE LIST

↳ We need to capture the variables as pass by reference to be able to modify them

↳ 

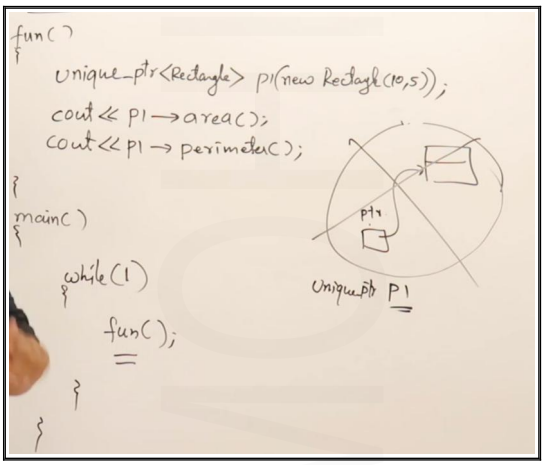
: allows to capture the all local scopes as reference

↳

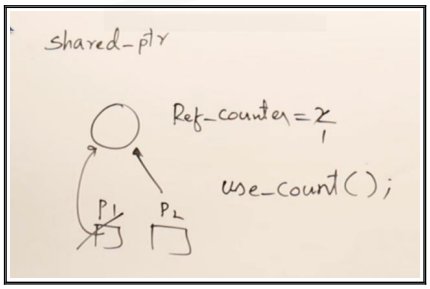
```
template<typename T>
void fun(T p)
{
    p();
}

int main()
{
    int a=10;
    auto f=[&a](){cout<<a++<<endl;};

    fun(f);
}
```

➲ Smart pointers

↳ unique_pointer : a class, A pointer can only point to a single class



↳ shared_pointer :



↳ weak_pointer : same as shared pointer but it does not maintains the reference counter.

➲

```cpp
int main()
{
    unique_ptr<Rectangle> ptr(new Rectangle(10,5));
    cout<<ptr->area()<<endl;

    unique_ptr<Rectangle> ptr2;
    ptr2=move(ptr);
    cout<<ptr2->area();
    cout<<ptr->area();

}
```

```cpp
int main()
{
    shared_ptr<Rectangle> ptr(new Rectangle(10,5));
    cout<<ptr->area()<<endl;

    shared_ptr<Rectangle> ptr2;
    ptr2=ptr;
    cout<<"Ptr2 "<<ptr2->area()<<endl;
    cout<<"Ptr "<<ptr->area()<<endl;
    cout<<ptr.use_count()<<endl;

}
```

➲ Inclass initialization

↳

```cpp
class Test
{
    int x=10;
    int y=13;
public:
    Test(int a,int b)
    {
        x=a;
        y=b;
    }
    Test():Test(1,1)
    {}

};
```

➲ Ellipsis

↳ used to pass multiple arguments to a function

↳

```cpp
int sum(int n, ...)
{
    va_list list;
    va_start(list,n);
    int s=0;
    for(int i=0;i<n;i++)
        s+=va_arg(list,int);
    va_end(list);
    return s;
}
sum(3,10,20,30)
sum(7,5,9,4,2,5,3,7)
```