

## Lecture 01:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "String.h"
int main()
{}
```

double y = 90;

Datatype of 90 is int.

By default datatype of integral values is int.

float g = 10.5;

Datatype of 10.5 is double.

By default datatype of floating point values is double.

(int) 90 ;

Type casting

y % 2;

Error: y is a double. No modulus operator on double.

void \* f = &a;

Void Pointer / Generic Pointer

It can hold address of any identifier.

\* f;

Error: It can't be dereferenced.

## Data type of pointers:

```
int *p = &a;  
&p;           // int **
```

```
int **q = &p;  
&q;           // int ***
```

```
const int a = 90;  
&a;           // const int *
```

```
const int *p = &a;  
&p;           // const int **
```

## Constant int in pointer:

```
int a = 10;  
const int *p = &a;
```

```
const int b = 90;  
int *p = &b;      Error;
```

p is a non-constant pointer  
to an int non-constant.

int a = 64;

&a; // 100

&a+1; // 104

cout << (long long int) (&a+1); → To view the address

68	67	66	65
103	102	101	100

int a = 16961;

cout << \* (char \*) &a; // A

cout << (int) \* (char \*) &a; // 65

cout << (int) \* ((char \*) &a+1); // 66

cout << \* ((char \*) &a+1); // B

int a = 16961;

char \* p = (char \*) &a;

cout << p[0]; // A

cout << p[1]; // B

## Pointee reading:

const int \* = int const \* // pointer to const int

int \*\* // pointer to pointer to int

int \*\* const // const pointer to pointer to int

int \* const \* // pointer to const pointer to int

int const \*\* // pointer to pointer to const int

int \* const \* const

// const pointer to const pointer to int

## Alias (By Reference)

&

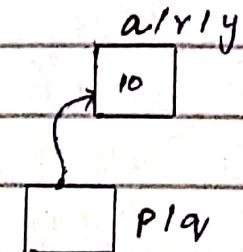
- i. Pass by value
- ii. Pass by reference
  - Pointer
  - Alias (nickname)

- Alias behaves like a constant pointer.
- Alias should be initialized at time of declaration.

```
int a = 10;
```

```
int &r = a;
```

```
int &y = r;
```



```
int *p = &a;
```

```
int * &q = p;
```

alias of pointer

```
**q;
```

Error

```
cout << &a << &r;
```

same address

- Alias is made for easier syntax than pointer.

# define \_CRT\_SECURE\_NO\_WARNINGS

memcpy

memcpy( void \* dst, const void \* src, size\_t n )

memcpy — memory copy

dst — destination

src — source

int a[4] = { 10, 20, 30, 40 }; // 16 bytes

int b[6] = { 5, 6, 7, 8, 9, 10 }; // 24 bytes

memcpy(a, b, 16);

for (int i=0; i<4; i++)

cout << a[i] << ":"; // 5:6:7:8:

memcpy(a, b, 8);

for (int i=0; i<4; i++)

cout << a[i] << ":"; // 5:6:30:40:

memcpy(b, a, 16);

for (int i=0; i<6; i++)

cout << b[i] << ":"; // 10:20:30:40:9:10:

memcpy( &b[2], a, 16 );

// 5:6:10:20:30:40:

creating own function for memory copy:

```
int a[4] = {10, 20, 30, 40};  
int b[6] = {5, 6, 7, 8, 9, 10};
```

```
myMemcpy (Rb[2], a, 16);
```

```
void myMemcpy (void * dest, const void * src, int bytes)  
{  
    char * d = (char *) dest;  
    char * s = (char *) src;  
    for (int i=0; i< bytes; i++)  
        d[i] = s[i]  
}
```

Overlapping by memcpy:

```
char msg[] = "0123456789abcdefg";  
cout << msg << endl;  
memcpy (&msg[4], 8msg[2], 7);  
cout << msg << endl;  
// 0123456789abcdefg  
// 01232345454bcdefgh
```

Results in Overlapping.

The answer should have been

```
// 0123'2345678bcdefgh
```

To avoid overlapping use memmove.

#define \_CRT\_SECURE\_NO\_WARNINGS

memmove:

memmove ( &msg[4], &msg[2], 7);

cout << msg << endl;

// 01232345678bcdeffgh

Creating own function for memory move.

```
void myMemMove(void *dest, const void *src, int bytes)
```

char \* d = (char \*) dest;

char \* s = (char \*) src;

char \* temp = new char [bytes];

memcpy (temp, s, bytes);

for (int i=0; i< bytes; i++)

d[i] = temp[i];

delete [] temp;

}

delete [] temp;

cout << temp;

Some random address.

\* temp;

might result in run-time  
Error.

```
const int a = 90;
```

```
const int *p = &a;
```

```
+ (int *) &a = 763;
```

```
cout << *p << " " << a;
```

```
}
```

// 763: 90

↓  
compiler blindly  
put a = 90 where he sees  
a in the program.

## Lecture 02:

Always:

```
int b[5];
```

(b is an array of size 5 of type  
int non-constant.)

→ 2-d arrays are like collection of 1-d arrays.

row size — columns

```
int a[5][4];
```

(a is an array of row size  
4 of type int non-constant)

```
int c[3][4][5];
```

(c is an array of rowsize 4x5  
of type int non-constant)

`int * x[5];` (`x` is an array of size 5 of type  
non-constant pointer to an int  
non-constant)

`int (*x)[5];` (`x` is a pointer of row size 5 to an  
int non-constant)  
`cout << size(x);` // 4

→ Array name is its base address.

→ 2-d array behave like a `int**`.

`int a[5][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};`

`void f(int (*h)[4], int rows)`  
{  
}

`int * p = &a[0][0];` `int *`

`cout << (long long int) &a[0][0];` - base address  
let // 232

same [   
     `cout << (long long int) a[1];` // 248  
     `cout << (long long int) &a[1];` // 248  
     `cout << (long long int) &a[1][0];` // 248

`232 + (1 * 4 + 0) * 4` - 248

```
void printArray(int *a, int size)
{
    for (int i=0; i<size; i++)
        cout << a[i];
}
```

```
int main()
```

```
{
```

```
    printArray(a[0], 20);
```

It can print 2-d array

```
    printArray(a[1], 4);
```

// SL78

→ Index operator \* is a binary operator

f[1]      \*(f+1)

```
int a = 10;
```

```
(&a)[0];      // *(&a + 0)
```

```
cout << *(&a + 0);      // 10
```

## Char Arrays:

```
string s;  
s = "abcd";  
cout << s; // abcd  
cout << s[1]; // b
```

On backend data is stored as character array.

a	b	c	d	'\0'
0	1	2	3	4

```
char s[6] = "Hello";
```

Primitive data types - base data types (int, char, float)

String is not primitive data type.

```
int a = 24; // 4 bytes  
char s[6] = "Ahsan"; // 6 bytes
```

```
char s[5] = "Ahsan"; Error: No space to store  
null character '\0'.
```

```
char s[3] = {'A', 'B', 'C'}; no need of null character
```

```
char name[100] = "Ali";  
cout << name; // Ali
```

'\0' - null character

integral value = 0

A	l	i	'\0'	.....	49
0	1	2	3	.....	49

automatically stores null character.

```
int i = 0  
while (name[i] != '\0')  
{  
    cout << name[i];  
    i++;  
}
```

On Backend this loop runs.

char name[1] = "Ali Hassan";      Array of size 11 will be  
made which can't be changed afterwards.

char name[100] = "Ali Hassan";

name[3] = 'O';

cout << name;      // Ali

cout << name;      // Ali

name[3] = 'O';

cout << name;      // Ali

name[3] = 'O';      // Ali O Hassan

cout << name;

## 2-d char arrays

students      size of names  
char name[40][100] = {{"Ali", "Ahsan"}};

	0	1	2	3	4	5	.....	99
0	A	l	i	'\0'				
1	A	h	s	a	n	'\0'		
2								
:								
39								

cout << name[1];      char \*      // Ahsan

& name[0][0];      char \*

name;      char (\*)[100]

In case of character array, cout statement also  
dereferences the address.

```
char name[3][5] = {"Ali", "Omer", "Saad"};
```

```
name[1][4] = 'I';
```

```
cout << name[1];
```

```
// OmerISaad
```

	0	1	2	3	4
0	A	L	i	'\0'	
1	O	m	e	r	'\0'
2	S	a	a	d	'\0'

Because in memory all data is stored as 1-d array. (Row-Major Order)

A	L	i	'\0'	O	m	e	r	'\0'	S	a	a	d	'\0'
I													

→ 2-d array behaves like a double pointer. int \*\*

```
name[1][2];
```

```
name[1] → address (1*5)
```

```
name[1][2] → address (1*5+2)
```

```
cout << (name[1])[2] << endl; // e
```

```
cout << name[1][2]; // e
```

But char \*\*p = name; - Error

Hello base

```
char name[50];
```

```
→ cin >> name; // Hello
```

```
→ getline (cin, name); // Hello base
```

```
→ cin.getline (name, 50); // Hello base
```

```
→ cin.getline (name, 50, 's'); // Hello b
```

## strcpy - string copy

```
char name[50] = "abcd";  
strcpy(name, "Hello How");  
cout << name;
```

// Hello How

## strcmp - string comparison

```
char name[] = "hello";  
char name1[] = "hello";  
int result = strcmp(name, name1);  
cout << result;
```

// 0  
In case of equal.

```
char name[] = "Hello";  
char name1[] = "hello";  
int result = strcmp(name, name1);  
cout << result;
```

// -1  
Negative value when

Hello < hello

array1 < array2

```
char name[] = "hello";  
char name1[] = "Hello";  
int result = strcmp(name, name1);  
cout << result;
```

Positive value when

hello > Hello

array1 > array2

strcat — string concatenation

```
char name[50] = "Omer";
```

```
strcat(name, " Khalid");
```

```
cout << name;
```

// Omer Khalid

Own string concat function:

```
StringConcat(name, " Khalid");
```

```
void StringConcat(char * dest, const char * src)
```

```
{
```

```
int i = 0;
```

```
while (dest[i] != '\0')
```

```
    i++;
```

```
int j = 0;
```

```
while (src[j] != '\0')
```

```
{
```

```
    dest[i] = src[j];
```

```
    i++;
```

```
    j++;
```

```
}
```

```
cout << dest;
```

```
}
```

// Omer Khalid

strlen — string length

```
char name[50] = "Omer";
```

```
cout << strlen(name);
```

// 4

## Corrupt stream

some flag on

int a;

cin >> a;

// hdfg

int b;

cin >> b;

End program here. No further  
input is taken.

string s;

cin >> s;

int a;

cin >> a;

// hdf

int b;

cin >> b;

cout << "qrs";

// hdf

qrs

## cin.good()

int a;

cout << cin.good();

cout << endl;

cin >> a;

// 123

// abc

cout << cin.good();

Output:

1

123

1

1

abc

0

## Solution for corrupt stream

```
int a;
```

```
cin >> a;
```

```
if (!cin - good())  
{
```

```
    cin.clear();
```

→ flag off

```
    while (cin.get() != '\n')
```

```
{  
}
```

```
    cin >> a;
```

```
}
```

```
cout << a;
```

// 123

// 454

123

675

675

## Lecture 03:

### struct

User / Programmer defined data type (UDT)

A C language structure

// To calculate Distance b/w two points

int x1, y1; // 1 Point

int x2, y2; // 2 Point

cin >> x1, y1;

cin >> x2, y2;

These Points let a, b can be stored in a user-defined data type (UDT)

Nouns → Objects

Attribute → Members

We all are human type objects.

struct Point

{

int x;

int y;

// x and y are members.

}

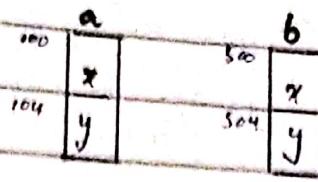
int main()

{

Point a, b; // a and b are two objects of  
data type Point.

Size: 8 bytes

As  $x$  and  $y$  are of type int.



→ Struct data type first letters should be capital  
e.g. Point.

### Assigning Values:

$a.x = 10;$

$a.y = 19;$

`cout << a.x << ":" << a.y;` // 10 : 19

(.) dot operator → binary operator

Member selection operator      L.H.S — object

R.H.S — member

`int x = 90;`      // Different variable  $x$ .

`cout << a;`      Error: UDT

`cin >> a;`      Error: UDT

### Initialization:

{  
Point a = {10, 20}, b = {1, 2};

`cout << a.x << a.y;`      // 10 20

, we can also do

$a.x = 90;$

## In Functions.

```
void printPoint ( Point a )
{
    cout << a.x << ":" << a.y;           // 90 : 20
}
printPoint ( a );                         // Pass by value ( copy )
```

int a, b; - Error: a, b are also variables  
of UDT.

a = b; // Sequential Assignment only if  
data type of both attributes / members  
of a and b is same.

```
cout << a.x << ":" << a.y;           // 1 : 2
```

```
double callDistance ( Point a, Point b )
```

```
{
    return sqrt( (b.x - a.x) * (b.x - a.x) + (b.y - a.y) * (b.y - a.y) );
}
```

```
cout << callDistance ( a, b );
```

```

struct Employee
{
    char name[50];
    int age;
    double salary;
};

```

```
int main()
```

Employee e1

name	consecutive
age	data
salary	

```
Employee e1 = {"Ali Raza", 45, 32876498};
```

```
cout << e1.name << e1.age << e1.salary;
```

// Ali Raza

45

3.28765e+000

## Lecture 04:

```
struct Rational
```

```
{
    int numerator; // num
    int denominator; // den
};
```

```
Rational addRational(Rational a, Rational b)
```

```
{ Rational res;
```

```
res.num = a.num * b.den + a.den * b.num;
```

```
res.den = a.den * b.den;
```

```
return res;
```

```
int main()
```

```
Rational x = {2, 3};
```

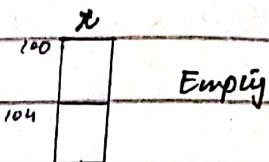
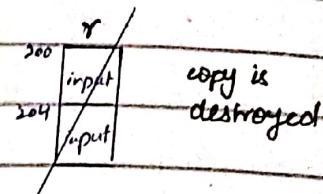
```
Rational y = {10, 20};
```

```
Rational result = addRational(x, y);
```

```
cout << result.num << "/" << result.den; // 70/60
```

As in functions, when we pass a struct object by value, a copy is made.

```
void inputRational ( Rational r )  
{  
    cout << "Enter numerator: ";  
    cin >> r.num;  
    cout << "Enter denominator: ";  
    cin >> r.den;  
}  
inputRational ( x );
```



## Passing by Reference

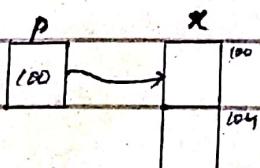
### → Pointers:

```
As      int a;  
       &a;      // int *
```

In previous example of Rational

```
&x;      // Rational *
```

Rational \* p = &x;



\*p.num = 901;      ERROR :

Precedence of (\*) is higher than (\*). So it treats p as an object but p is a pointer.

(\*p).num = 901;

\*p = y ;      // x = y : Assigning value of y members to x.

printRational (\*p);      // printRational (x)

- `inputRational (&x);`

```
void inputRational (Rational *r)
{
    cout << "Enter numerator : ";
    cin >> (*r).num;           // cin >> r -> num;
    cout << "Enter denominator : ";
    cin >> (*r).den;          // cin >> r -> den;
}
```

(→) Arrow operator : binary operator

L.H.S → object address

R.H.S → member

→ Alias :

y / test

`Rational & test = y;`

890

`test.num = 890;`

→ Alias can't do everything what a pointer does.

Like alias needs to be initialized when declared.

`inputRational (x);`

`inputRational (Rational &r)`

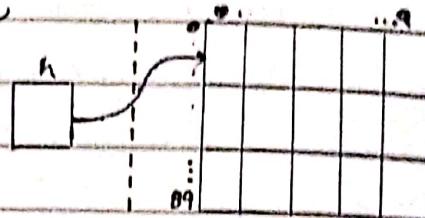
## 2-d Array:

On Heaps

int a = 90; b = 87;

int (\*h)[b] = new int [a][b];      Error: Row size (b)  
can't be a variable.

int (\*h)[10] = new int [a][10];



Another method:

struct Matrix

{ int \* data;

int rows;

int columns;

};

void createMatrix (Matrix &m, int r, int c)

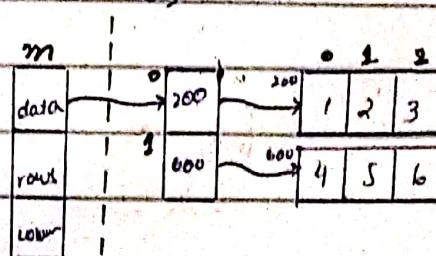
{ m.rows = r;

m.columns = c;

m.data = new int \*[m.rows];

for (int i=0; i<r; i++)

m.data[i] = new int [m.columns];



Also deallocate memory at the end by using a loop.

```

void inputMatrix (Matrix *m)
{
    for (int i = 0; i < m->rows; i++)
    {
        for (int j = 0; j < m->columns; j++)
        {
            cout << i << ": " << j;
            cin >> m->data[i][j];
        }
    }
}

int main()
{
    Matrix m1;
    createMatrix (&m1, 2, 3);
    inputMatrix (&m1);
}

```

### getSetElement Function:

```

int & getSetElement (Matrix &m, int r, int c)
{
    return m.data[r][c];
}

int main()
{
    Matrix m1;
    createMatrix (&m1, 2, 3);
    getSetElement (m1, 1, 2) = 89;
}

```

With this function, we can also perform

## Bounds Checking & Exception Handling

### → Bound checking.

```
int & getSetElement (Matrix &m, int r, int c)
{
    if (r >= 0 && r < m.rows && c >= 0 && c < m.columns)
        return m.data[r][c];
}
```

Error: what if this condition is false. Function has to return something.

### → Exception Handling.

```
int & getSetElement (Matrix &m, int r, int c)
{
    if (r >= 0 && r < m.rows && c >= 0 && c < m.columns)
        return m.data[r][c];
    cout << "Invalid Bound";
    exit (0); // terminate a program
}
```

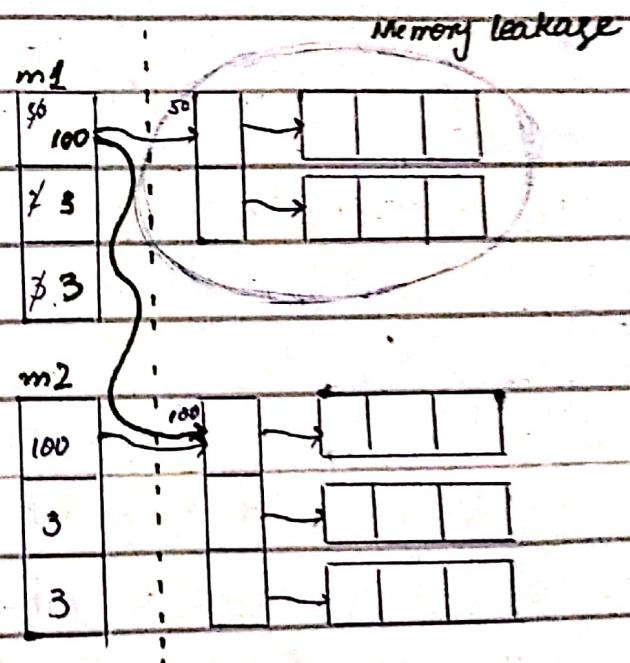
```
int main()
```

```
{
    Matrix m1;
    createMatrix (m1, 2, 3);
    for (int i = 0; i < m1.rows; i++)
    {
        for (int j = 0; j < m1.columns; j++)
        {
            cout << i << ":" << j;
            cin >> getSetElement (m1, i, j);
        }
    }
}
```

## Deallocation:

```
int main()
{
    Matrix m1;
    ...
    for (int i = 0; i < 2; i++)
        delete [] m1.data[i];
    delete [] m1.data;
}
```

```
int main()
{
    Matrix m1;
    createMatrix(m1, 2, 3);
    Matrix m2;
    createMatrix(m2, 3, 3);
    m1 = m2;
}
```

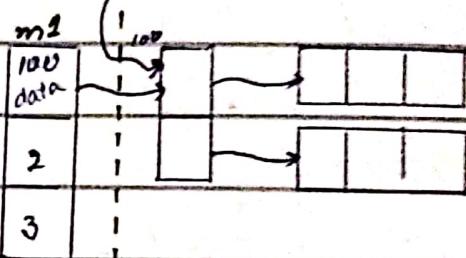


→ This results in memory leakage and also the data is inconsistent now that memory is shared.

Just like the previous function.

```
void inputMatrix(matrix m)
{
    for (int i=0; i<m.rows; i++)
    {
        for (int j=0; j<m.columns; j++)
        {
            cout << i << ":" << j;
            cin >> m.data[i][j];
        }
    }
}

int main()
{
    Matrix m1;
    createMatrix(m1, 2, 3);
    inputMatrix(m1);
}
```



If we pass a matrix object by value, only copy of its members is made. It points to the same address.  
No copy of array is made.

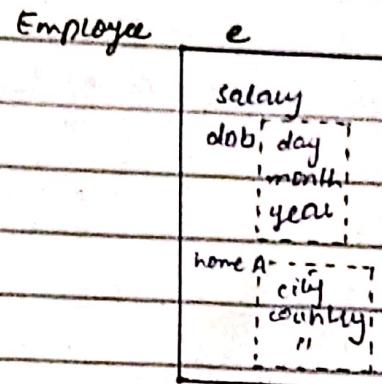
## Lecture 05:

```
Struct Date
{
    int day;
    int month;
    int year;
};
```

```
Struct Address
{
    char city[30];
    char country[50];
    int streetNo;
    char block[30];
}, char colony[100];
```

```
Struct Employee
{
    double salary;
    Date dob;
    Address homeA;
    Address officeA;
};
```

```
int main()
{
    Employee e;
```



$$e \cdot \text{Salary} = 300.10;$$

$$e \cdot \text{dob} \cdot \text{day} = 13;$$

Input Date( e.dob );

}

Date d[5];

day	month	year
1	day	month
2	day	month
3	day	month
4	day	month

$$d[1] \cdot \text{month} = 10;$$

Input Date ( &d[2]);

$$d[2] = d[0]; \quad 4 \text{ Members copied}$$

```

struct Set {
    int * data;
    int noe;
    int cap;
};

```

```

void InputSet ( Set * s )
{
}

```

```

void InputSet ( NamedSet * s )
{
    cin >> s->name;
    InputSet ( &( s->nset ) );
}

```

```

struct NamedSet {
    Set nset;
    char name[10];
};

```

	data
	noe
	cap
name	

```

struct Point {
    int x;
    int y;
};

```

```

struct Line {
    Point start;
    Point end;
};

```

```

double calDistance ( Line ln )
{
}

```

```

return sqrt = ( ln.end.x - ln.start.x ) * ( ln.end.x - ln.start.x )
            + ( ln.end.y - ln.start.y ) * ( ln.end.y - ln.start.y );
}

```

## Lab Task A.3

```
struct Set
```

```
{  
    int * data;  
    int noOfElements;  
    int capacity;  
};
```

```
void createSet (Set *s, int capacity)
```

```
{  
    s->capacity = capacity;  
    s->data = new int [capacity];  
}
```

```
int isMember (Set &s, int ele)
```

```
{  
    int i=0;  
    while (i < s.noOfElements && s.data[i] != ele)  
        i++;
```

```
    return (i < s.noOfElements ? i : -1);  
}
```

```
bool insertElement (Set *s, int elem)
```

```
{  
    if (s->noOfElements == s->capacity)  
        return false;
```

```
    int ind = isMember (*s, elem);
```

```
    if (ind != -1)
```

```
        return false;
```

```
    s->data [s->noOfElements] = elem;
```

```
    s->noOfElements ++;
```

```
    return true;
```

```
}
```

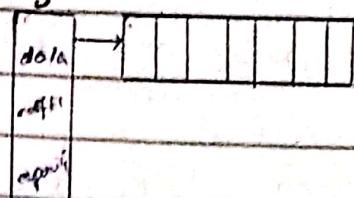
```
void displaySet (Set s)
```

```
{  
    for (int i=0 ; i < s.noOfElements ; i++)
```

```
        cout << s.data[i] << ", ";
```

```
}
```

Set S



## Resizing Array

```
int isMember (Set &s, int ele)
{
    int i=0;
    while (i < s.noOfElements && s.data[i] != ele)
        i++;
    return (i < s.noOfElements ? i : -1);
}

void resize (Set &s)
{
    Set s2;
    s.capacity = s.capacity * 2;
    createSet (&s2, s.capacity);
    for (int i=0; i < s.noOfElements; i++)
        s2.data[i] = s.data[i];
    freeSet (&s);
    s.data = s2.data;
}

bool insertElement (Set *s, int elem)
{
    if (*s->noOfElements == *s->capacity)
        resize (*s);
    int ind = isMember (*s, elem);
    if (ind != -1)
        return false;
    (*s->data[*s->noOfElements]) = elem;
    (*s->noOfElements)++;
    return true;
}
```

## Lecture 06:

Practice - 03

### Enumeration

Numbering (Sequence)

To improve readability

```
enum Colors { RED, GREEN, BLUE };
```

RED, GREEN, BLUE → constant identifiers.

Assigned values

0, 1, 2 automatically.

```
enum TempStatus { BOIL_POINT = 100, FREEZE_POINT = 0 };
```

```
{ int main()
```

```
    int temp;
```

↓  
if not assigned 0, it

will be automatically  
assigned value 101.

```
    cin >> temp;
```

```
}
```

```
{
```

BOIL\_POINT = 75;

// ERROR : BOIL\_POINT is constant  
identifier.

```
enum Month { JAN, FEB, MAR, ... };
```

```
int main()
```

```
{ const int daysOfMonth[] = { 31, 28, 31, ... };
```

```
cout << daysOfMonth[JAN];
```

Month m;

m = JAN;

m = 0;

// ERROR : Only data stored in enum.

```
enum GameStatus { WIN, LOOSE, DRAW, IN-PROGRESS };
```

```
GameStatus check()
```

```
{
```

```
GameStatus st;
```

```
if (-)
```

```
return st;
```

```
}
```

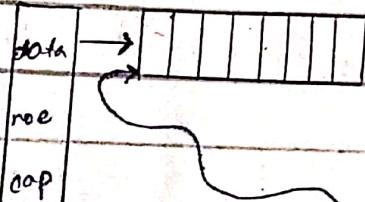
```
while (check() == IN-PROGRESS)
```

```
{
```

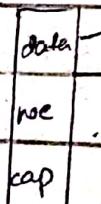
```
}
```

Always with struct pass by value.

s1



x



```
displaySet(s1);
```

Shell / boundary

Shallow copy

Shallow clone

Member wise copy

Bit wise copy

→ Bit-wise copy : Behaviour

```
memcpy(&x, &s1, sizeof(set));
```

If we want to give no right to modify the array, we have to create a clone.

## Cloning (identical copy)

deep clone

displayset ( createClone ( s1 ));

Set createClone ( Set st )

```
int * temp = new int [st.capacity];
for (int i=0; i < st.size(); i++)
    temp [i] = st.data [i];
st.data = temp;
return st;
```

Always middle shell are automatically passed by Value.

struct Student

{ char s[10];

int marks[5];

float CGPA;

};

int main()

{ f (s);

};

f ( Student x )

" passed by value "

};

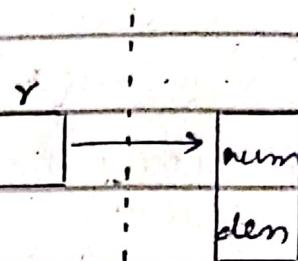
};

## Struct on Heap:

Rational r;

Rational \*r = new Rational;

delete r;



Away:

Rational \*p = new Rational [5];

p[0].num = 75;

delete [] p;

## Lecture 07:

### OODP

**Object :** Set of attributes with integrated functionalities.

**Identity :** To identify objects of same type.

**State of object :** Should always be valid.

- Abstraction
- Encapsulation
- Information Hiding
- Aggregation / Composition / Association
- Inheritance
- Polymorphism

- Templates
- Exception Handling
- File Streams

- Abstraction: Unleash / Details hidden
  - Functional Abstraction  
Hidden how function works.
  - Data Abstraction  
Hidden information and how it is stored.

- Encapsulation: Data and its functionality combined.

- **composition:** Object is composed of different types of objects.
- **Association:** Object is associated with different types of objects.
- **Polymorphism:** Responding differently on the same message by different objects.

class Time

{ private :

int hours;

int minutes;

int seconds;

public :

void inputTime()

{ cout << "Enter hours: ";

cin >> hours;

cout << "Enter minutes: ";

cin >> minutes;

cout << "Enter seconds: ";

cin >> seconds;

}

void printTwelveHourFormat()

{ cout << (hours > 12 ? hours % 12 : hours) << ":";

cout << minutes << ":" << seconds;

cout << (hours >= 12 ? " PM" : " AM");

}

} ;

```
int main()
```

```
{  
    Time t1 = {13, 20, 50};
```

→ Error

```
    t1.hours = 49;
```

→ Error

```
Time t2;
```

```
t2.inputTime();
```

// Now can take input

## Wrapper Functions /

## Getter Setter Functions

To input values in such objects.

```
public:
```

```
void setHours ( int h )
```

```
{ if ( h >= 0 && h <= 23 )
```

```
    hours = h;
```

```
}
```

```
int getHours ()
```

```
{ return hours;
```

```
}
```

```
int main()
```

```
{
```

```
    Time t1;
```

```
    t1.setHours ( 13 );
```

```
}
```

→ Never in invalid state.

## Lecture 08:

Access Modifiers

- ↳ private
- ↳ public

Default access modifier

struct → public

class → private

Constructor :

used for

- initialization
- Resource deallocation

↳ Default constructor

called for every object

public :

Time()

{ hours = 0;

minutes = 0;

seconds = 0;

}

↳ Parameterized constructor

Time( int h, int m, int s )

{

setTime();

}

|| Bug : as it will assign garbage  
if input is not correct.

```
int main()
```

```
{
```

```
Time t1(1, 2, 3), t2;
```

```
t1.printTwelveHourFormat();
```

// For t1 parameterized constructor  
will be called.

```
t2.printTwelveHourFormat();
```

// For t2 default constructor  
will be called.

```
}
```

## ↳ Delegate constructor

```
Time(int h, int m, int s) : Time()
```

```
{  
    setTime(h, m, s);  
}
```

It will execute first

```
int main()
```

```
{  
    Time t(22, -4, 50);
```

```
    t.printTwelveHourFormat();  
}
```

Common Mistake:-

```
Time y();
```

Function Declaration

```
Time y{};
```

constructor

```
Time t { 22, 30, 45 };
```

we can also use it to initialize array on heap.

```
int * p = new arr [3] { 10, 20, 30 };
```

## Lecture 09:

### ↳ Direct initialization:

Time (int h = 12, int min = 30, int sec = 10)

{

}

→ Not good approach

If you don't make a constructor, empty constructor will execute automatically.

But if you make atleast one constructor, it doesn't do this on its own.

### ↳ Resource Allocation:

class Array

{

int \* data;

int capacity;

public:

Array (int cap = 0)

{     capacity = 0;

    data = nullptr;

    if (cap <= 0)

        return;

    capacity = cap;

    data = new int [capacity];

}

## ↳ Resource deallocation:

### Destructor :

Automatically executes at the end.

But you can call it and use it anytime

$\sim \text{Array}()$

{

if (data)

    delete [] data;

    data = nullptr;

    capacity = 0;

}

### Call destructor :

this  $\rightarrow$   $\sim \text{Array}()$  ;

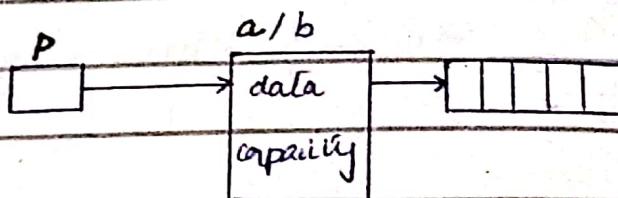
### Sequence:

int a, b, c;

Constructor executes for a, b, c

Destructor executes for c, b, a

## Pointer and Alias :



Array \*  $p = \&a$  ;

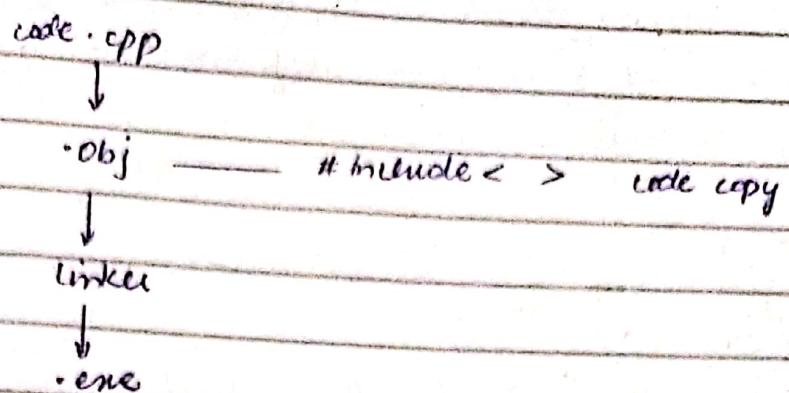
( $*p$ ) . getSet(1) = 32;

$p \rightarrow$  getSet(1) = 32;

Alias :

Array &  $b = a$ ;

## Lecture 10:



Link — Linker

Creating own include files:

MyMath.h // only declarations  
int pow(int, int);  
double sqrt(int);  
void table ( int ,int );

MyMath.cpp // Definitions  
#include "MyMath.h"

...  
...

Driver / Main.cpp

#include "MyMath.h"  
int main()  
{  
}

}

Never include in Main.cpp the MyMath.cpp as it can result as double included definitions.

- Declarations can be included more than once but it increases the size of your .cpp file.
- A class can't be included twice.
- Definitions should be included only once.

Local → Scope within that function.

Static → within one .cpp file.

Global → within the project (multiple files)

#include " "      searches in current project then in include.

#include < >      searches in include file

#include " C:\11\_11\ "      → path of file

#define

when one file.h is included don't include it again.

File.h / String.h

File name constant

#ifndef STRING\_H #

#define STRING\_H #

:

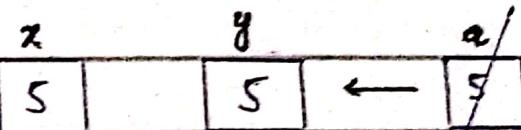
#endif

∴ ifndef → if not defined

## Lecture 11:

### RVO - Return Value Optimization

```
int f(int a)
{
    return a;
}
```



```
int main()
{
    int x = 5;
    int y;
    y = f(x);
}
```

```
int main()
```

```
{  
    int x = 5;  
    int y = f(x);  
}
```



## Copy Constructor :-

→ Default copy constructor does shallow copy.

```
ClassName( const ClassName & )
```

```
    Time( const Time & ref )
```

```
{
```

```
    memory( this, &ref, size of( Time ) );
```

```
}
```

```
Time( Time ref )
```

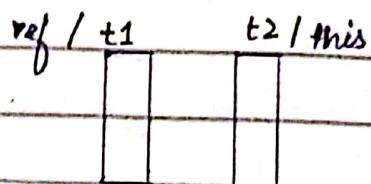
```
{
```

```
}
```

Syntax Error : Infinite recursion

```
Time t1;
```

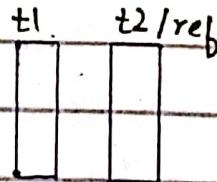
```
Time t2( t1 );
```



```
void f( Time x )
```

```
{
```

```
    t1 = f( t2 );
```



## String

```
String( const String & ref ) : String()
```

```
{ size = ref.size;
```

```
if( ! ref.data )
```

```
    return;
```

```
data = new char[ size ];
```

```
for( int i=0 ; i<size ; i++ )
```

```
{ data[i] = ref.data[i];
```

```
}
```

```
size = ref.size;
```

```
}
```

## Lecture 12:

### Using const

```
class Test
```

```
{
```

```
    int a;
```

```
    int b;
```

Direct initialization const int c = 11; // 1st stage

const int y[3] = {1,2,3}; // 2nd stage for array

Member initialization list

```
Test(): c(10), a(50), b(6) // 2nd stage
```

```
{  
}  
    test(int x, int y, int z) : a(x), b(y), c(z) // Last stage  
    {  
        // Values can't be initialized here.  
        // Only assignment can be done.  
    }  
};
```

will execute first as defined in class

b(y), a(b), c(b)

↑ garbage

→ Member initialization list is preferred over direct initialization.

```
int main()
```

```
{  
    Test t(45); // Test(&t, 45) // Test *  
    const Test s; // &s // const Test *
```

// last stage for initialization  
is Body of constructor

```
s.a = 90;
```

Error: constant object

```
{
```

```
int main()
{
    const Test t;
    t.a = 90;           Error: const object
    Test s;
    s.a = 90;           // No error
}
```

necessary for const class

```
void f() const
{
    const Test t;
    t.a = 90;           Error: const object
    this->a = 28;       Error: const object
}
```

```
int main()
{
    const Test s;
    s.f();             // f(&s)
```

Non-constant className \* → this non-constant  
constant className \* → this constant

→ this (className \*) - exception of constant inside  
constructor and destructor.

```
void f()
```

```
{
```

```
void f() const
```

```
{
```

```
}
```

→ can co-exist (Signature difference)

```
class
```

```
String
```

```
{
```

```
    char * const data;
```

```
    const int size;
```

```
    data[0] = 9;
```

```
    data = new char[10]; Error: const +
```

```
};
```

Alias:

```
int &x = 85;
```

Error: temporary data

```
const int &x = 85;
```

```
cout << x; // 85
```

x

85

## Lecture 13:

### Using static

Global lifetime  
Local linkage / file scope

→ Internal function by default external linkage.

a.h

a.cpp

if included here

then its existence count

a.cpp	b.cpp	main.cpp
void f();		int main() {
void f()	void f()	f();      // Error: Not Found
{	{	}
}		must include prototype before.
↓ // Error: Multiple definitions Found		
void f()	static void f()	
{	{	
↓ can co-exist		
		int main()
		{
	static void g()	g();      // Error: Not Found
	{	g() has file scope.
		}

## → Global variable

a.cpp

```
int var = 10;  
global  
variable
```

main.cpp

```
extern var; → Must include  
int main()  
{  
    cout << var; // 10  
}
```

static int var = 10;

cout << var; // Error

a.cpp

```
static int var = 10;
```

b.cpp

```
int var = 100
```

main.cpp

```
int main()  
{  
    cout << var; // 100  
}
```

## → Class

```
class MyMath  
{  
public:
```

```
    int pow( int b, int c)
```

```
    {
```

```
}
```

```
};
```

```
int main()
```

```
{  
    MyMath m;
```

```
    m.pow(2, 3);
```

```
{
```

$m \rightarrow$  Size 1 byte (minimum memory)

→ static — class level information

```
class MyMath  
public:  
    static int pow()  
};  
};
```

```
int main()  
{  
    MyMath m;  
    m.pow(2,3);  
}
```

// call

or

```
MyMath::pow(2,3);  
}
```

// call

```
class Student
```

```
public:
```

```
int rollNo;
```

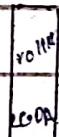
```
float CGPA;
```

```
static int x;
```

```
static char college[20];
```

```
};
```

Student



college



```
int Student::x;
```

```
char Student::college[20];
```

] Must declare globally if  
not then linker error

```
int main()
```

```
{  
    Student::college; // call
```

```
    Student s1;
```

```
}
```

Initialization:

```
int Student :: x = 10;  
char Student :: college[20] = "PUCIT";
```

class Student

{ static int x;

public:

static void Student :: setX(int val)

x = val;

}

};

class Car

{ static int id;

int carId;

Car()

{ carId = id;

id++;

}

};

int Car :: id = 1;

→ constant with static

•) data member

static const char college[20];

char Student :: college[20] = "PUCIT"; → necessary

•) function

With functions can't use static and const together

learn

nick

## Lecture 14:

### Relationships:

#### Reusability

- Association
- Aggregation
- composition
- Inheritance
- Dependancy
- Polymorphism

→ whole part relationship (one-way)

#### Association:

Class is associated with another class.

#### Composition:

Death relationship

Strong aggregation relationship

e.g. Car and Wheels

#### Aggregation

Object have their own lifetime

Shade / weak aggregation

e.g. Car and Driver

### UML Notations:

Unified modeling language

→ Association

↔ Aggregation

→ composition

→ Inheritance

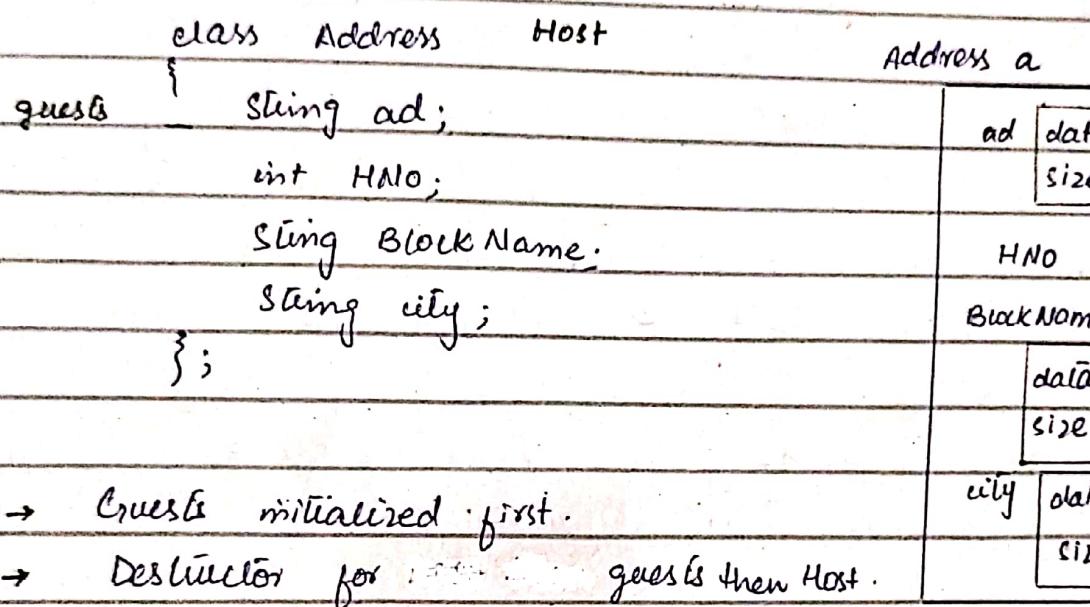
---> Dependancy

## Aggregation:

Parts are implemented as pointer member variables in the whole class.

## Composition:

Parts are implemented as non-pointer member variables in the whole class.



Address(s) : ad("abc"), HNo(12), BlockName("D"), city("L")  
 {  
 }  
 initialized first

a.ad.data ; — Error  
 private

Address(const Address& ref) : ad(ref.ad), BlockName(ref.BlockName),  
city(ref.city)  
 (copy constructor for string)

## Lecture 15:

use for class over struct:

- For backward compatibility
- Default access modifier

C++ is a strongly typed language.

string ("abc") . display();

unnamed object (created and destroyed in this statement).

cascading call (side by side call)

string ("OOP") . concatenate ("Hello") . concatenate ("END") . display();

NamedSet (int cap = 0, string s = "A") : setObj(cap), Name(s)

{

}

int main()

{

NamedSet ns;

ns.insert(13);

}

void insert (int ele)

{ setObj.insert(ele); }

}

NamedSet ns

Name	data	size
setObj	A4	13

Name	data	size
setObj	13	4

## Prototype      Forward Declaration

class B;

class A

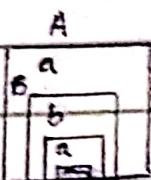
```
{    B * pa;    → No error but can't dereference  
    B obj;      → Error (don't know constructor for B)  
    int data;  
};
```

class A

```
{    int a;  
    B obj;  
};
```

class B

```
{    int b;  
    A ob;  
};
```



Not Possible

class A

```
{    int a;  
    B * obj;  
};
```

class B

```
{    int b;  
    A + ob;  
};
```

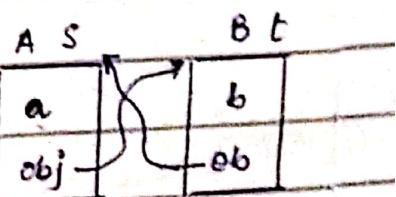
Possible

int main()

```
{    A s;  
    B t;  
}
```

$$t \cdot ob = \&s$$

$$s \cdot obj = \&t$$



## Lecture 16:

### Cyclic Reference

A.h

#include "B.h"

class A

```
{ A(int)  
}
```

B.h

#include "A.h"

class B

```
{ A obj;
```

B(): obj(12)

```
{  
}
```

B(const int & ref): obj()

```
{  
}
```

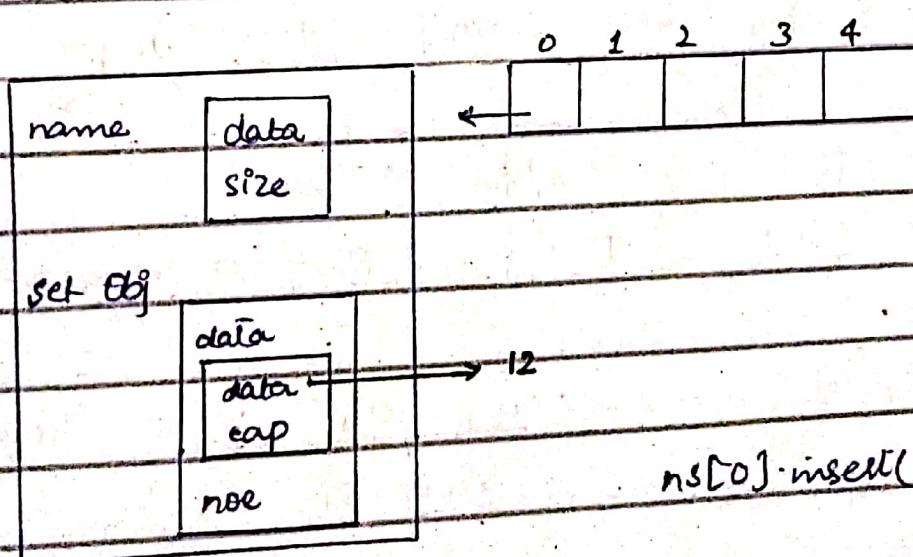
```
}
```

B o1;

B o2(o1);

### Array of Objects:

NamedSet ns[5];



Time + t = new Time;

delete t;

NamedSet \* p = new NamedSet [S];

delete [] p;

class A

{ int i,j;

public:

A (int a)

{ }

A (int a, int b)

{ }

{ }

A ()

{ }

{ }

}

A x[3] x

0 ij

1 ij

2 ij

constructor using nameless object:

A x[3] = { A(12), , A(30) }; → Error

A x[3] = { A(12), A(30) }; // Default for third one

A x[3] = { (1,2) }; → Error

A x [3] = { 12,13 }; // Only for one argument

→ This kind of initialization is preferred only on Stack.

On Heap it is not so practical.

A \* p = new A[3] { A(12), A(13), A(1,2) };

A \* p = new A[n] → cannot work

A y = A(12);

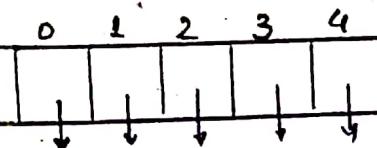
A y = 12;    // Default constructor for y that receives an int.

Double - pointer:

NamedSet \*\*p = new NamedSet \* [5];

p[1] = new NamedSet;

delete p[1];



## Lecture 17:

### Operator Overloading:

For user-defined data types

- Arity of operator — It takes how many operands
- Can't change operator arity, precedence & associativity.

[] index operator — binary

! not operator — unary

Default → constructor, destructor, copy constructor;

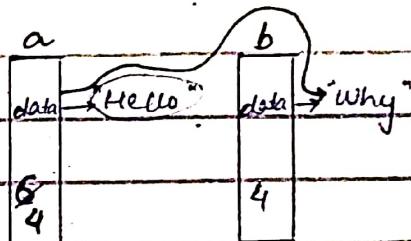
Available assignment operator

- Just like copy constructor does shallow copy, assignment operator can produce some unwanted results due to memory leakage.

String a("Hello");

String b("why");

a = b; Assignment



copy constructor

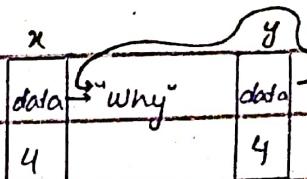
String x("why");

String y(x);

String x = a;

At the time of declaration

→ It does not result in memory leakage.



calling object  $a = b$  receiving object  
operator

$a[i] = 'c'$

$a \cdot \text{operator}[i] = 'c'$

Backend:  $a \cdot \text{operator} = (b)$

can also write

className & operator (const className &)

→ Copy Assignment Operator:

String & operator = (const String & ref)

if ( $this == \&ref$ ) // To resolve  $a=a$

return  $*this;$

$this \rightarrow \sim String();$

if (!ref.data)

return  $*this;$

$size = ref.size;$

$data = \text{new char}[size];$

for (int i=0; i<size; i++)

$data[i] = ref.data[i];$

return  $*this;$

}

// className & To resolve cascading assignment ( $a=b=c$ )

By doing this we can simply do.

→ In copy constructor

String (const String & ref) : String()

{  
 $*this = ref;$

}

## Lecture 18:

`cout << flush << "start";`

↳ immediately prints output on console.

`<< '\n'` preferred

`<< endl` slow as it is `<< '\n' << flush`

→ In watch we can write and check expressions.

`(void *) & str[0]` // address

Question `** q, pool`

→ we created a double pointer instead of pointer to Question as

it takes less space in memory (only that of a pointer), and it makes it easier

to move object.

## Operator Overloading?

`!s`

`s.operator !()` → no parameter as it is unary

→ logic of `IsEmpty()`

`Set s, s2;`

`++s + s2;`

`s++ + s2;`

## → Prefix

$\text{++s}$

$s.\text{operator}++()$

## Postfix

$s++$

$s.\text{operator}++(\text{int})$

## Set & operator++()

{

`for(int i=0; i<n; i++)`

`data[i]++;`

`return *this;`

}

## Set operator++(int)

{

`Set x(*this);`

`for(int i=0; i<n; i++)`

`data[i]++;`

`return x;`

}

↳ Always return by value cause  
we are returning a temporary.

## → Unary

$-m$

$+m$

## → IOStream :

→ ostream

`cout << s;`

`cout.operator << (s);`

↳ But cout class is inaccessible / can't be modified.

So in the case of inaccessible type or primitive type  
create global functions.

→ primitive

23 + s

Error

s + 23

no error

→ Global functions don't have access to private data.

### String.h

ostream & operator<< ( ostream & os, const string & s);

### Global

ostream & operator<< ( ostream & os, const string & s);

{

if (!s)

return os;

int i = 0;

while (s[i] != '\0')

{ cout << s[i];

i++;

}

return os;

}

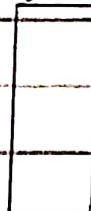
] or

cout << &s[0];

] or

s.display();

os / cout



ostream & x = cout;

x << "Hello";

↓ Can do this by creating an alias only.

Can't create a copy

Like

ostream n = cout; Error

can also access different functions of cout with cout.

## → istream

istream & operator >> (istream & is, string & s)

```
{  
    s.input();  
    return is;  
}
```

istream / ostream & return type

because in case of

int a, b;

cin >> a >> b;  
→

left to right

cin >> a // returns alias

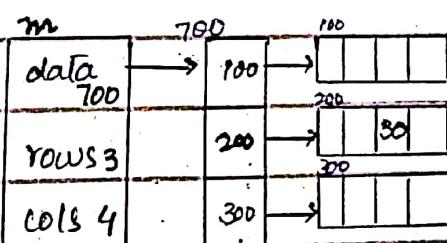
operator + (int, int)      Error

// Can't change behavior of primitive data types.  
One must be user-defined.

Matrix m(3,4);

m[1][2] = 30;

200



int \* operator [] (int ind)

{ if (ind >= 0 && ind < rows)

// no bounds checking

return data[ind];

As it checks only rows

}

Not practical

→ Function call operator

operator()

s("Hello");

↳ function call object / function

m(1,2) = 30;

m.operator()(1,2)

int & Matrix::operator()(int i, int j)

{ if((i>=0 && i<rows) && (j>=0 && j<cols))

return data[i][j];

exit(0);

}

Another way:

class Matrix

{ Array &;

int \* data;

int rows;

int cols;

Matrix(int r, int c)

{

rows = r;

cols = c;

data = new Array[rows];

for(int i = 0; i < rows; i++)

data[i].resize(cols);

}

}

```
Array & Matrix:: operator[](int i)
{
    if(i >= 0 && i < rows)
        return data[i];
    exit(0);
}
```

## → Type cast operator:

operator type() → also the return type

A. obj;

int x = (int) obj;

```
class A
{
public:
    explicit operator int()
    {
        return 13;
    }
}
```

→ If explicit is not written function can be called both explicitly and implicitly.

returning a value either true or false

while (str)

```
explicit operator bool() const
{
    if(data == nullptr || data[0] == '\0')
        return false;
    return true;
}
```

## Lecture 19:

Operators that can't be overloaded

? :

sizeof

:

\*

Operators that can't be globally defined

=

( ) typecast

[ ]

→ Operator &&

Alias of only a temporary / r value.

int && r = 23;

int a = 10;

int && b = a; // Error (L-value)

→ At compile time, compiler takes decision  
of return value optimization.

However compiler can't handle ambiguity.

String f()

{ String a("Hello"), b("Bye");

int x = 10,

if (x > 7)

return a;

else

return b;

}

String z = f();

Move implicit copy constructor  
value to return

→ Move Constructor:

Transfer resources.

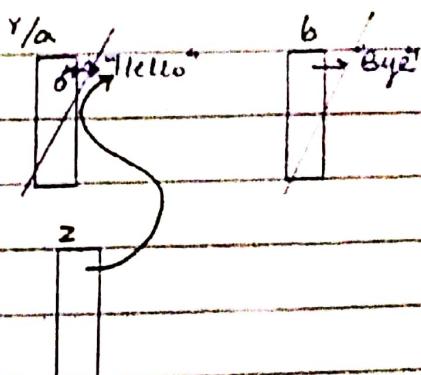
String (String && r)

data = r.data;

size = r.size;

r.data = nullptr;

}



String f()

{ String a("Hello");

return a;

}

String z = f();

// a will be given name z. Or if move constructor available  
then move otherwise copy constructor.

## → Move assignment Operator :

String z;

$z = f();$

String & operator = (const String && r)

{  
    this → ~String();

    data = r.data;

    size = r.size;

    r.data = nullptr;

    return \*this;

}

Creating an optimized Swap.

int main()

{  
    String x("Hello"), y("Bye");

    mySwap(x, y);

}

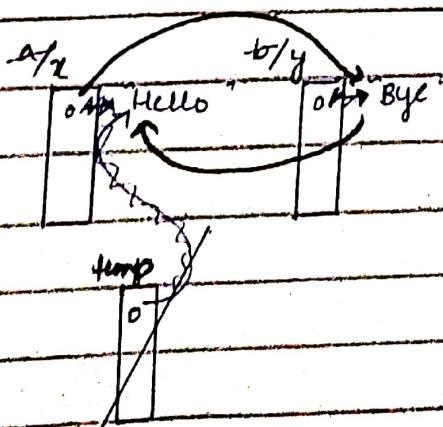
void mySwap(String &a, String &b)

{  
    String temp = move(a);

    a = move(b);

    b = move(temp);

}



## → One Argument constructor / Type cast constructor Conversion constructor.

class A

```
{ int x;
```

public:

A()

```
{}
```

A(int)

```
{}
```

```
}
```

int main()

```
{
```

A obj;

obj = 36;

// obj = A(36)

class A

```
{ a
```

class B

```
{ b
```

a=b

Preference:

1. Assignment operator
2. Type cast operator
3. Type cast constructor

cout << (1,2,3,4);

114

## → Restriction

`String (const String &) = delete;`

`String& operator=(const String &) = delete;`

`f(int i)`  
  `{ }`

// Only only int

`f(float) = delete;`

`String() = default;`

`String(char *)`  
  `{ }`

## → Use of explicit

`explicit A(int)`  
  `{ }`

`int main()`  
  `{`

`A obj;`

`obj = 68; // Error`

`obj = A(8); // need to call explicitly`

→ Ambiguous statement for compiler if explicit removed from same preference operators.

→ After defining any constructor, default constructors are removed : So, we can do

`A() = default;`

## Lecture 20:

### Friend Functions and classes :-

Friendship is always granted not taken.  
But because of this encapsulation and abstraction is destroyed.

class String

{

friend void f();

friend class A;

friend void A::g();

}

void f()

{

class A

{ friend class String;

void g();

}

→ Friend function of a class is only one way.

→ Friendship in classes can be two ways.

Two way friendship in classes can access each other data even private.

In C++, operator overloading is done by making static functions. It has access to all the private data. But it doesn't use the concept of friends.

## Singleton Pattern :-

An algorithm that not more than one object of a class can be created.

```
class A
{
    int x;
    int y;
    A()
    {
        // Object can't be made
    }
public:
    void f();
    void g();
}
```

$A * p = (A *) \text{new } \text{int}[2];$

$p \rightarrow x = 30;$

```
class Singleton
{
    int x, y;
    static Singleton * ptr;
    Singleton()
    {
        // Object can't be made
    }
    ~Singleton()
    {
        // Object can't be made
    }
public:
    Factory
    {
        Static Singleton * create Object()
        {
            if (!ptr)
                ptr = new Singleton();
            return ptr;
        }
    }
}
```

Singleton \* p = Singleton::createObject();  
Singleton a(+p);

```
static void removeObject()
{
    if (ptr)
        delete ptr;
    ptr = nullptr;
}
```

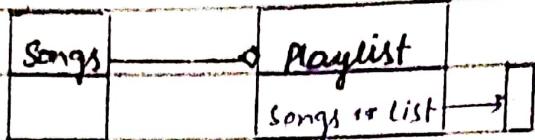
```
class A
{
    int x;
    int y;
    static A st;
}
```

A	st
x	x
y	y

```
A obj;
obj.st.x = 30;
obj.st.y = 40;
```

## Aggregation :- (has a)

- parent-child relationship
- Independent parent and child
- Shared aggregation



## Composition :- (part of)

- Exclusive relationship
- Whole-part relationship
- Dependent
- One-way



Class Subject

{ String Title;

String Code;

## Associations (knows a)

- Two-way relationship

class Student

{ Bike \* b = nullptr;

public:

Bike \* getBike()

{ return b;

```
void setBike( Bike * p )
```

```
{
```

```
b = p;
```

```
b -> setStudent( thru );
```

```
}
```

```
}
```

```
class Bike
```

```
{
```

```
Student * s = 0;
```

```
public,
```

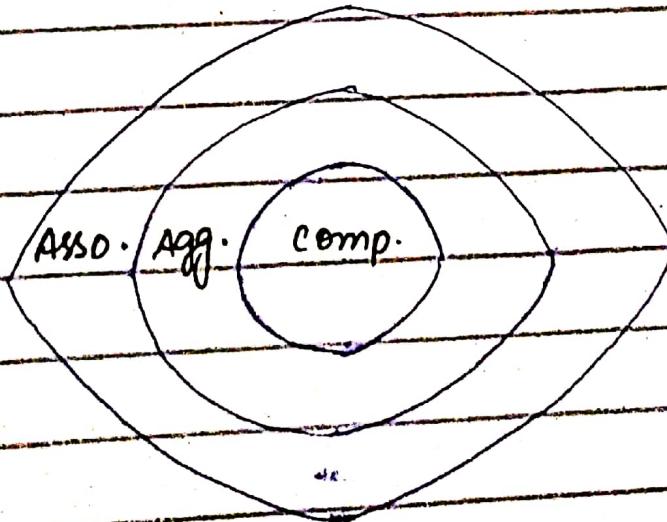
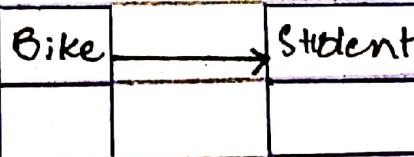
```
void setPrice( Student * x )
```

```
{
```

```
s = x;
```

```
}
```

```
}
```



## Dependency :-

(uses a)

```
class Date  
{  
}
```

```
String getInformation()  
{  
}
```

## Lecture 21

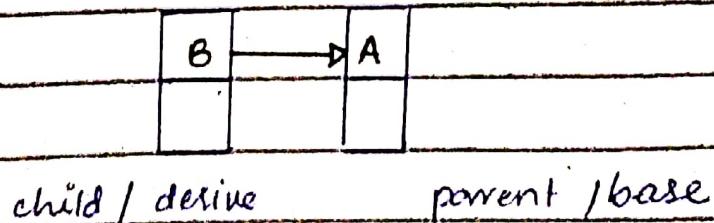
### Inheritance :- (is a)

- One-way relationship
- Uni-directional

A class required as it is with additional functionality.

```
class B : class A  
{  
    int b;  
public:  
    void h()  
};
```

class A  
int a;  
public:  
void f()  
{  
}



→ Child inherits parent.

B obj;

obj.h();

obj.f();

B obj

a - constructor first

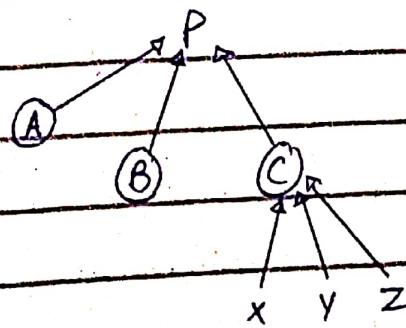
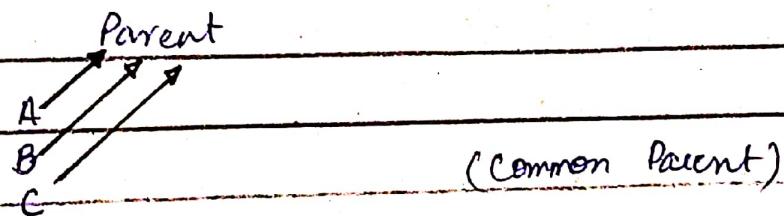
b - destructor first

- Constructor / initialization of parent first.
- Destructor for child first.

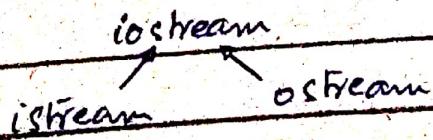
⇒ We can't use friend classes to reuse functionality of a class with additional features because we will have to write all the wrappers again.

## Generalization :-

inheritance



→ Direct Parent (C) of X, Y, Z.



## default Access Modifier

→ Public, (public, private, protected)

class Student : public Person

{ string rollNo;

}

class Address

{ string ads;

string city;

string country;

}

class Teacher : public Person

{ string specialization;

double salary;

}

class Person : public Address

{ string name; // Logically wrong but implementation fine

string bGroup;

Address add; // Right way

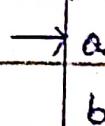
}

class B : public A

B obj;

obj.f();

B obj



→ Base class can refer to derive class object.

Strong class base

Strong object derive

→ Never manipulate derive class objects through base class.

Constructor explicit of base class.

class B : public A

{

public:

B()

{

}

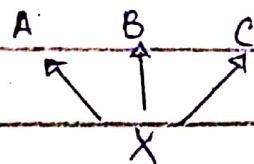
# default

B(): A(13)

{

}

}



A \* p = xobj;

⇒ B(const B & ref) : A(ref) # inher.

⇒ B (const B & ref) : obj(ref obj) # comp.

Function is first checked in derive class.

Hiding:

Redefining a function in derive class hides all the function in base class with same name even with different prototypes.

B obj;

obj: A => X();

## Assignment:

```
B& operator=(const B& ref)
{
    A::operator=(ref);
    b=ref.b;
}
```

## Visibility:

```
class B:public A
{
    int b;
public,
using A::x; // If written in private part
}                                         can't use this function in
                                            derive.
```

B obj;

A \* p = & obj;

p->f(); // A

obj.f(); // B

## Constructor:

```
NamedSet(const Set & ref): Set(ref)
{
}
```

## Lecture 22

→ private : All data of that class becomes private'

```
class B : private A
{
    int b;
    B()
}

void f()
{
}
```

```
class A
{
    int a;
    A()
}

void g()
{
}
```

```
int main()
{
    B obj;
    obj.g(); // Error : not accessible
    A * p = new B; // Not allowed
```

```
A * p = & obj;
p->g(); // Can be done like this
```

→ Protected :

Available within class and down the hierarchy (in child classes also, if any).

↳ without inheritance relationship between classes there is no difference between private and protected.

B : protected //

{ int min()

{ A \* p = new B;

p → f(); // Possible

B obj;

obj.g(); // Error

	Private	Public	Protected
Private	X	private	private
Public	X	public	protected
Protected	X	protected	protected

## → Constructor Inheritance :

struct A

{ int a;

A() {}

A(int y) {}

void f()

void g()

struct B : public A

B() : A() {}

B(int z) : A(z) {}

or

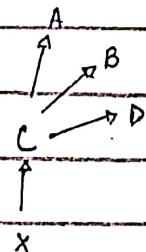
{ using A::A; // Inherited constructor

B() {}

B(int x) {}

constructors are not inherited.

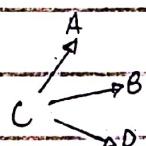
### Inheritance Pattern:



C is direct parent of X. Through C

A, B, D are also parents of X.

### → Multiple inheritance:

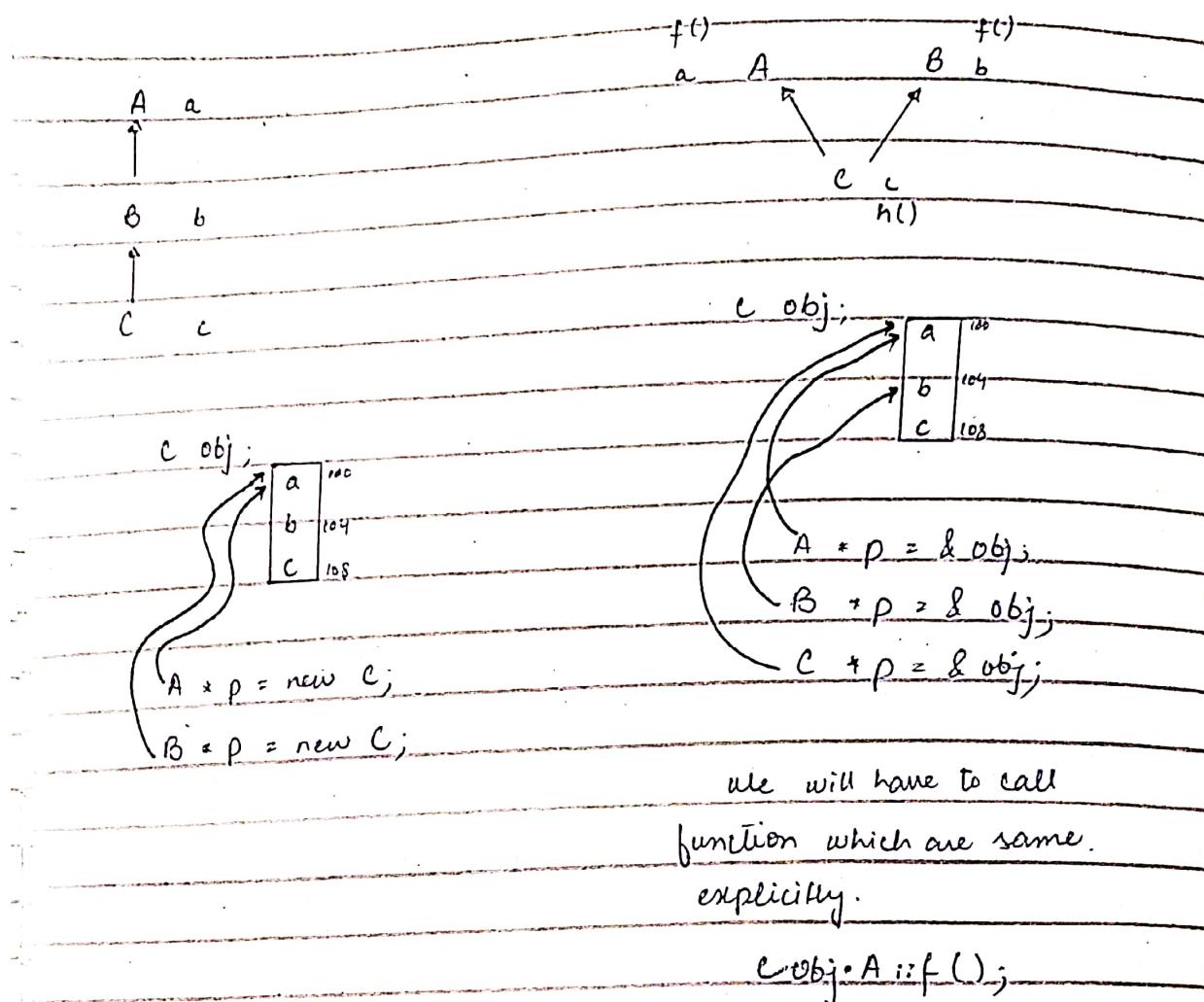


A, B, D are direct parents of C.

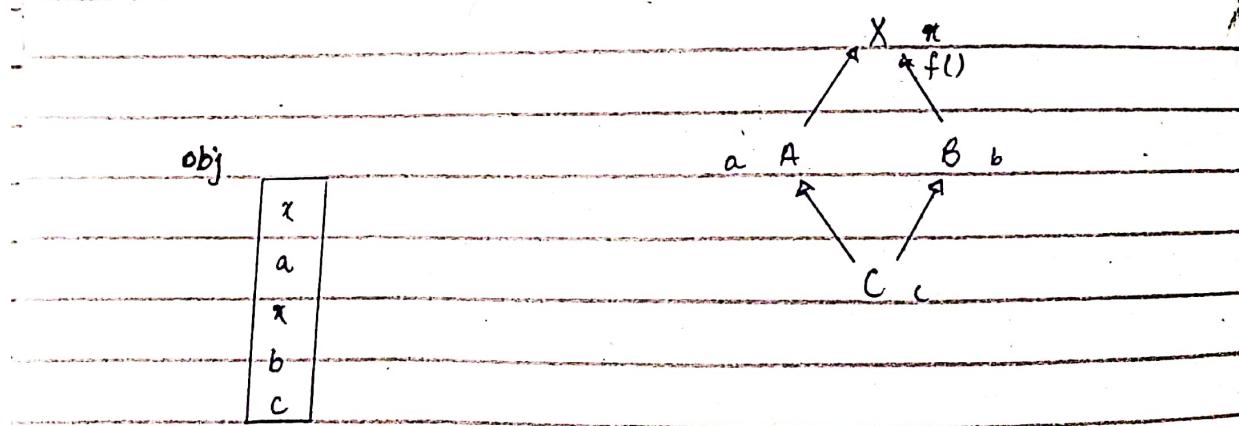
class C: public A, public B, public D

Constructor in order;

A, B, D, C



### → Diamond Inheritance :



class C : public A : public B

$c\ obj;$

(can't decide the  
path to take)

$obj.\ a;$  // Error  
 $obj.\ f();$  // Error

## Lecture 23

### Polymorphism 2 -

Windows

```
class Printer  
{  
public:  
    virtual void print()  
};  
{
```

```
class ABCPrinter : public Printer  
{  
public:  
    void print()  
};  
{
```

Printer \* p = new ABCPrinter;

p → print();      || print() of Windows

Windows

```
class PrinterList  
{  
    Printer ** plist = new Printer *[10];  
    int np = 0;  
    int select = 0;  
  
public:
```

```
void addPointe ( Pointe * p)
{
    plist[nop] = p;
    nop++;
}
```

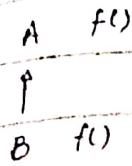
```
Pointe()
{
    plist[select] → point();
}
{
```

PointeList pl;                  // Global

```
int main()
{
    pl.addPointe (new ABCPoint);
    pl.addPointe (new XYZPoint);
}
```

```
void printPointeList():
{
    for (int i=0 ; i < nop ; i++)
    {
        cout << plist [i] → getName();
    }
}
```

→ We can achieve polymorphism by using virtual.



$pB \rightarrow A :: f();$

$pB \rightarrow f();$

$pa \rightarrow f();$

$pa \rightarrow B :: f();$  // Error

### Runtime Polymorphism:

using Virtual // Out of compiler's understanding

### Compile time Polymorphism:

calling functions with different parameters

$f(1);$

$f(1, 2);$

$f(1, 2, 3);$

### → Runtime Polymorphism:

A virtual  $f()$

B  $f()$

$x()$

$y()$

A  $a;$

B  $b;$

$A * p = \&b;$

$p \rightarrow x();$

$p \rightarrow f();$

|

Latest Implementation

→ Compiler only check static type

```
class A
{
    virtual void g()
}
```

```
class B
{
    void g()          // overwrite ; Automatically virtual
}
```

→ If signature matches of two virtual functions then  
their return type must be same. Otherwise syntax error.

```
class A
{
    virtual void g()
}
```

```
class B
{
    void g()
    int g(int);
}
```

```
class A
{
    void f()
}
```

```
class B
{
    virtual void f()
}
```

```
class C
{
    void f()
}
```

$A * p = \text{new } B;$   
 $r \rightarrow f(); \quad // \quad A = f()$

## Function Pointers -

```
int main()
{
    void (*p)(void);
    p = f;
    p();           // f()
}
```

→ 4 or 8 bytes pointer depending on machine size.

So Jump of 4 or 8 bytes.

→ A function pointer should have same signature and return type of the function it is pointing to.

```
int main()
{
    void (*p[3])(void);
    p[0] = f;
    p[1] = x;
    p[0]();           // f()
}
```

# Lecture 24

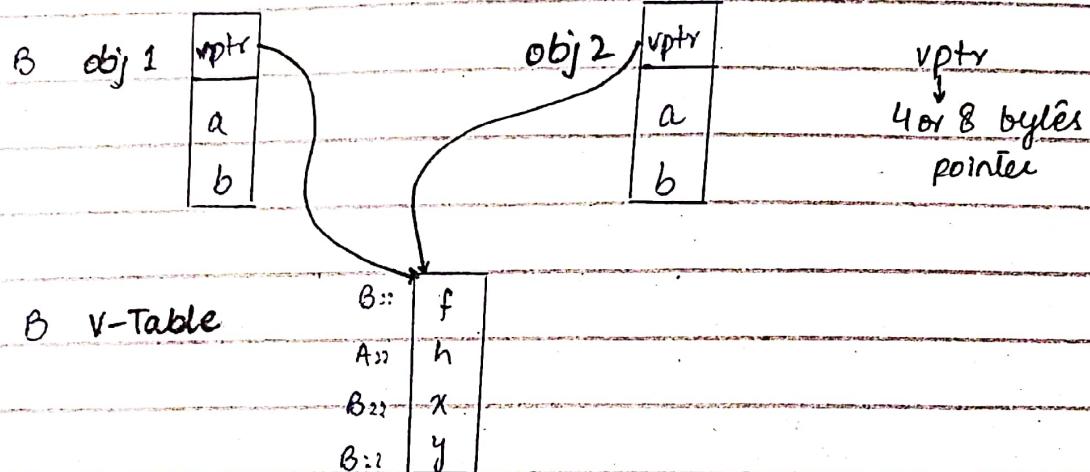
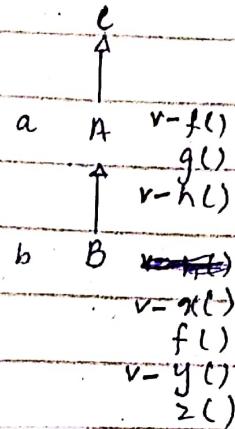
## V-Table

## (Virtual Table)

→ Array of Function Pointers

→ One v-table for a class containing atleast one virtual function.

No v-table for C.

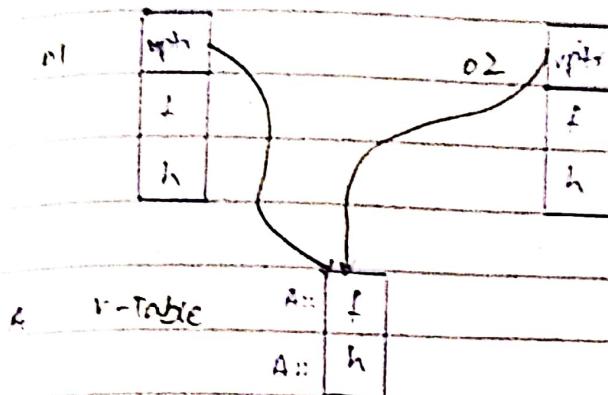


Now ( $A + p$ )

{      $p \rightarrow h();$      //  $A::h()$

$\star(\text{int} + p)$      // Address of v-table

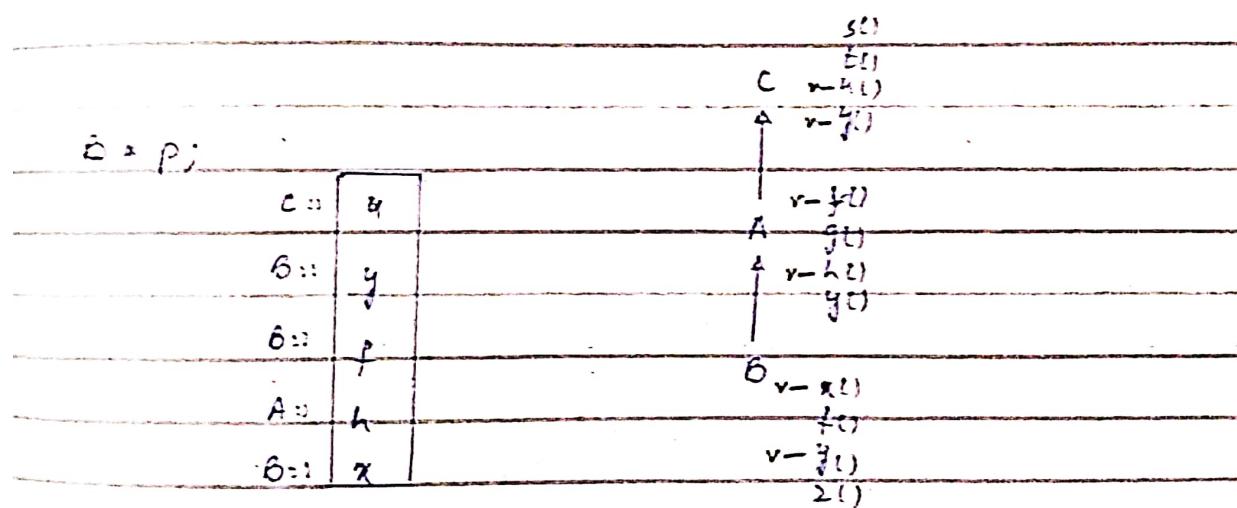
A = o1, o2;



o1.f(); // No polymorphism

A + p = & o1;

p → f(); // Polymorphism occurs



→ v-table for every class is made.

class A

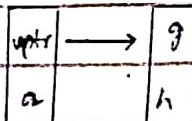
{ }

size of (A) :      // Byte alignment / Padding  
Power of 2

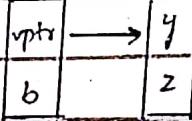
a.

intellisense

A



B



struct A

{  
    int a;  
    virtual void g() {}  
    virtual void h() {}  
}

struct B

{  
    int b;  
    virtual void y() {}  
    virtual void z() {}  
}

int main()

{  
    A obj;  
    B \* p = (B \*) & obj;

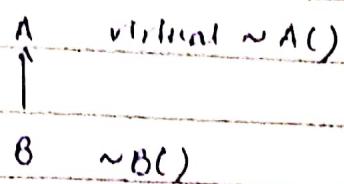
p → y();      // A::g

}

A obj;

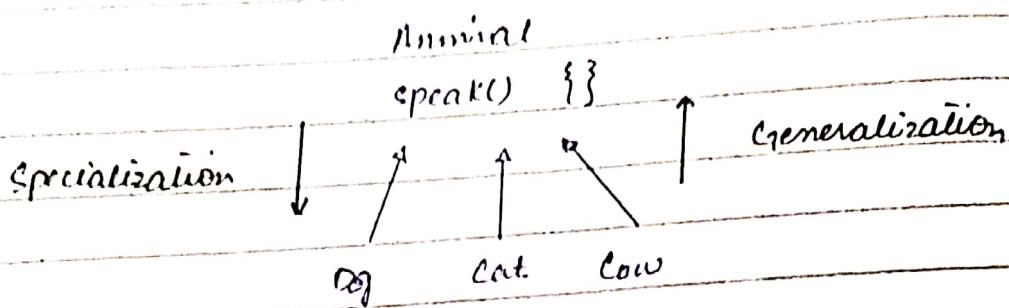
+ (int \*)((void \*\*) obj + 1);      // A::a

- In case of constructor/destructor no polymorphic call.
- Using virtual with destructor, all destructions of derived classes become virtual.



## Lecture 25

void print() override  
// if function not virtual before,  
make it virtual.



### Pure-virtual Function

virtual print() = 0;

1. Pure virtual function definition can't be given in class. Can be defined globally.
2. An object can't be made of a class that consists of atleast 1 pure virtual function. Treated as an Abstract class.
3. Every derived class must implement these functions.  
(Programming by contract)

Concrete class:

- Object can be made.

Abstract class: (Mixine)

- Object can not be made.
- Implemented not inherited.
- Some pure virtual functions and some non-pure.

Pure Interface: (Pure)

- Only pure virtual functions.

Interface ABC

(Always public)

{}  
11 pure virtual functions only

Abstract  
class A  
{}  
v-f() = 0  
{}

Abstract  
class A  
{}  
v-f() = 0  
{}

Abstract  
class B  
{}  
void g()  
{}

Concrete  
class B  
{}  
void f()  
void g()  
{}

→ Destructor:  
If made pure virtual then other classes  
must write a destructor.  
So, you have to provide destructor  
definition of base class logically.

→ virtual Constructors:  
(Only logically)

```
1 class A
    {
        virtual A* clone();
        {
            return new A(+this);
        }
    }

2 class B
    {
        virtual A* clone()
        {
            return new B(+this);
        }
    }

3 class C
    {
        virtual A* clone()
        {
            return new C(+this);
        }
    }

3 void f(A* p)
    {
        A * x = p->clone(); // polymorphic
    }
```

$$A * P = \begin{matrix} A \\ B \\ C \end{matrix}$$

If class A has pure virtual function

virtual A \* clone() = 0;

then no object can be made, so no definition of creating new object is provided of class A.

new A; // Syntax Error

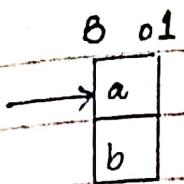
A obj; // Syntax Error

char x[16];

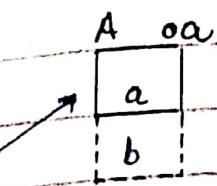
A \* p = (A \*)x; // Runtime Error

P → g();

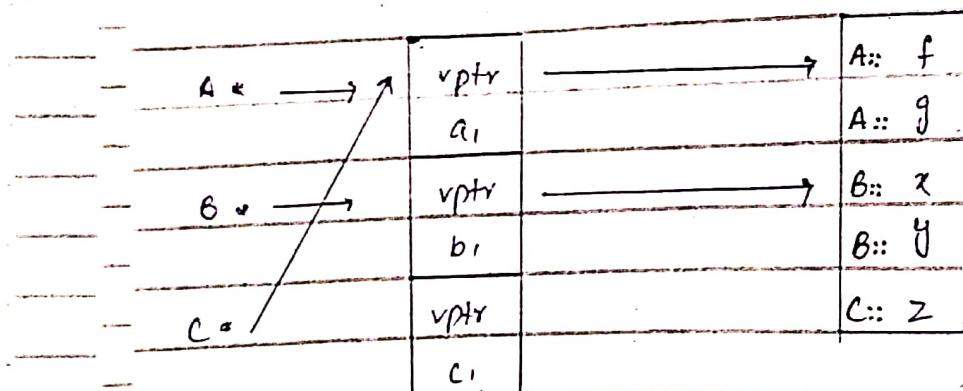
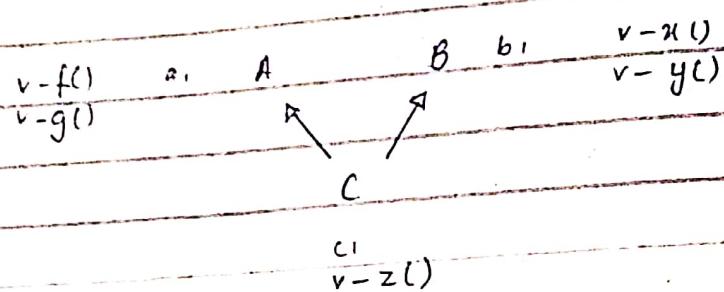
$B \text{ o1};$   
 $A + P = B \text{ o1};$



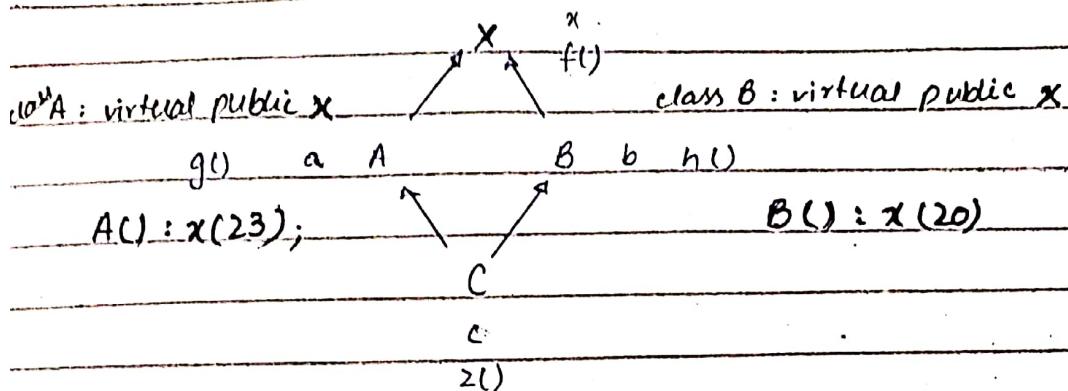
$A \text{ oa};$   
 $B + PB = (B \times) \& \text{ oa};$   
 $PB \rightarrow b; \quad \text{1 EMO}$



## Multiple Inheritance :



## Virtual Inheritance :



- e decides which constructor to run.
- Nothing to do with vptr / v-table.

→ Before Virtual Inheritance :

x	
a	
x	
b	
c	

size = 20 bytes

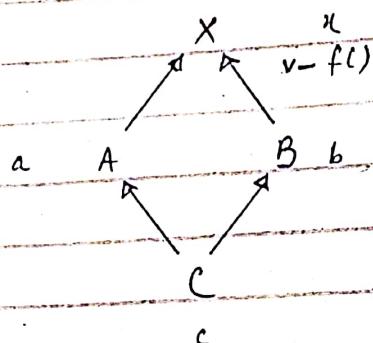
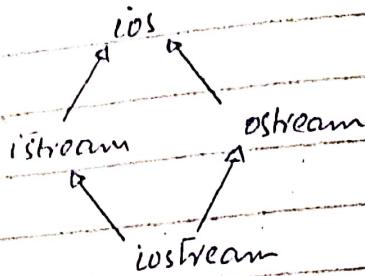
→ After Virtual Inheritance :

class A : virtual public X

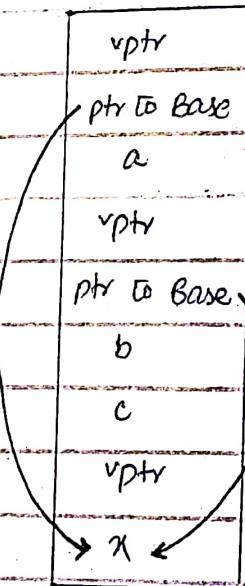
class B : virtual public X

x	
a	
b	
c	

## Lecture 26



→ layout on compiler



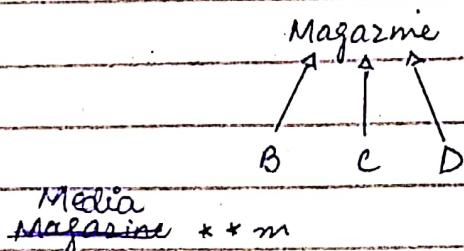
→ First construction of Virtual classes then  
Non-virtual classes will execute.

## Type Casting :

- const cast
- static cast
- dynamic cast

→ `dynamic_cast<>( )`

only for polymorphic types



`Magazine * q = dynamic_cast<Magazine*>(m[i])`

// If true then assign to pointer

otherwise places null.

→ Upcast is allowed.

→ Downcast is not allowed, generally.

→ Downcast only for polymorphic classes.

- ✓  $A * p = \text{new } B;$



$B * q = p;$  // Shouldn't do it.  
bcz. p doesn't know its  
dynamic type.

- ✓  $B * q = \text{static\_cast} < B * > (p);$  // p knows its static type  
 $B * q = \text{dynamic\_cast} < B * > (p);$  // Shouldn't do it.

→ const cast

removes constness of an identifier temporarily.

## RTTI

(Runtime Type Identification)

→ For polymorphic usage:

```
int a = 10;
```

```
cout << typeid(a).name(); // int
```

→ RTTI is enabled by default.

→ Used for dynamic cast.

→ Can also disable RTTI.

class Z  
{ }

class X  
{ }

Z \* p = new X; // Type of p: Z

class Z  
{ virtual void f()  
}  
}

class X  
{ }

Z \* p = new X; // Type of p: X

## Templates :-

a mold

→ Capital, single letter e.g. T

template < typename T >

→ visibility of a template is in the class directly  
below upto semi colon.

template < typename T >

void mySwap (T &a, T &b)

{

T temp;

// //

}

→ Function is generated only once for each type.  
in object file (.cpp) (At compile time)

mySwap < int > (x, y);

T replaced by int

mySwap < float > (a, b);

T replaced by float

mySwap(x, y);

// Implicitly detects type  
from variables.

```
int x = 4;  
float y = 10;
```

my swap(x, y); // Error: don't know which function to call

my swap<int>(x, y); ↴ // Error: because type doesn't match with function parameter type.

```
template <typename T>  
int linearSearch(T *arr, int size, T key)  
{  
    int i = 0;  
    while (i < size && arr[i] != key)  
        i++;  
    return i == size ? -1 : i;  
}
```

```
int a[] = {10, 20, 30, 40, 50};
```

```
cout << linearSearch(a, 5, n);
```

```
int a[] = {10, 20, 30, 40, 50};
```

```
float x = 20.5;
```

```
cout << linearSearch<int>(a, 5, x);
```

// works. Truncates the float part.

// template <typename T>

class Array

{  
    T \* data;

    int size = 0;

public:

    // //

}

Array <int> a(5);

Array <int> b(5);

a = b;

// Backend:

class Array <int>

class Array <float>

Array <float> c(5);

c = b;

// Shallow copy

// Error: type difference

Array <int> \* p = & a;

Array <int> & q = a;

→ Copy constructor:

Array (Array <T> & ref)

## Nested

Array <Array <int>> a;

class Array<Array <int>>

{

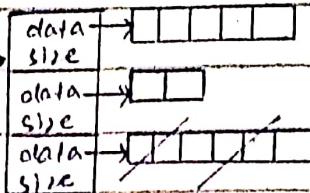
    Array <int> \* data

class Array <int>

{

    int \* data

`Array<int> * data`



(2-d Array)

`a[0][1];`

`a.resize(3);`

`a[0].resize(5);`

`a[1].resize(2);`

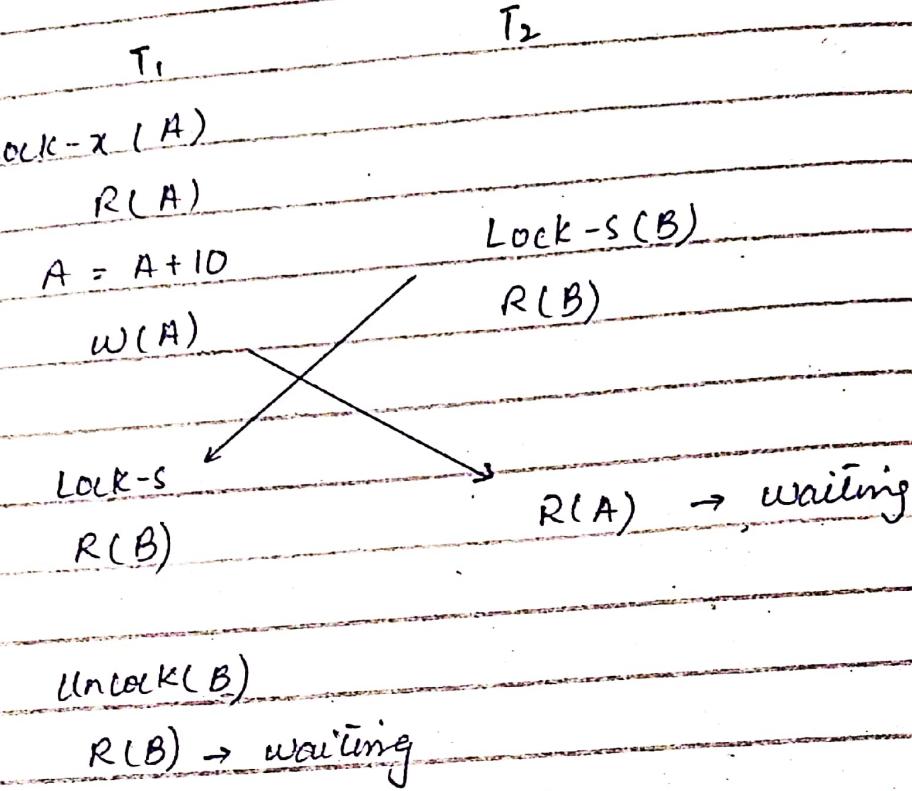
`a[2].resize(6);`

↓

Jack Array: Each row size is different.

`a[2].~Array();`

Deadlock : lock for infinite time.



- Prevention.
- Detection
- Avoidance

Performed by DBMS

End