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:

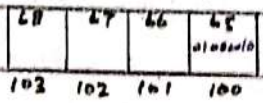
```
int *p = &a;
&p;                    // int **

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
```




```
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
```

Pointer 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)

&

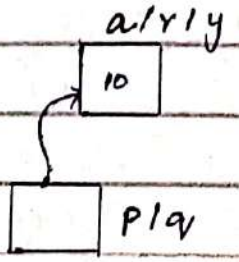
i. Pass by value

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 ( &b[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]
}
```

Over-lapping by memcpy:

```
char msg[] = "0123456789abcdeffgh";
cout << msg << endl;
memcpy ( &msg[4], &msg[2], 7);
cout << msg << endl;
                // 0123456789abcdeffgh
                // 0123234545bcdeffgh
```

Results in Over-lapping.

The answer should have been

// 0123234567 8bcdeffgh

To avoid overlapping use memmove.

#define _CRT_SECURE_NO_WARNINGS

memmove:

```
memmove ( &msg[4], &msg[2], 7);
cout << msg << endl;
                    // 0123234567abcdefgh
```

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 aways are like collection of 1-d aways.

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 << sizeof(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); // 5678

→ Index operator * is a binary operator

f[1] *(f+1)

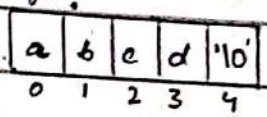
int a = 10;

(&a)[0]; // *(&a+0)

cout << *(&a+0); // 10

Char Arrays:

On backend data is stored as character array.

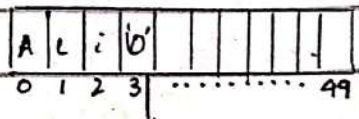



```
string s;
s = "abcd";
cout << s;        // abcd
cout << s[1];     // b

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



automatically store null character.

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

On Backend this loop runs:

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

```
char name[100] = "Ali Hassan";
name[3] = '\0';
cout << name;          // Ali
cout << name;          // Ali
name[3] = 0;
cout << name;          // Ali
name[3] = '0';         // Ali0Hassan
cout << name;
```

2-d char arrays

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

	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];
        // OmerI Saad
```

	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 bsse

```
char name[50];
→ cin >> name;                     // Hello
→ getline(cin, name);              // Hello bsse
→ cin.getline(name, 50);           // Hello bsse
→ 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

72 104

Hello < hello array1 < array2

```
char name[] = "hello";
char name1[] = "Hello";
int result = strcmp (name , name1);
cout << result ;                 // 1
```

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;             End program here. No further
cin >> b;          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:	Output:
1	1
123	abc
1	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

123

// uyu

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)

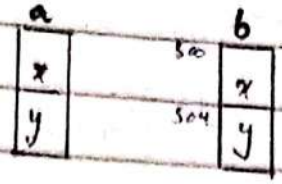
Nouns → Objects
Attribute → Members

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 letter 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};
                x  y        x  y
    cout << a.x << a.y;          // 1020
```

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 = {"Ali Raza", 45, 32876498};
    cout << e1.name << e1.age << e1.salary;
}
```

Employee e1

name
age
salary

consecutive data

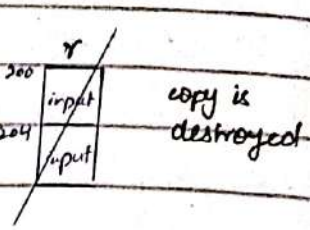
```
// Ali Raza
// 45
// 3.28765e+x
```

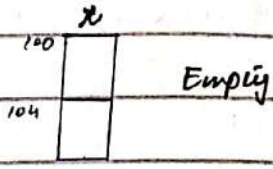
Lecture 04:

```
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:

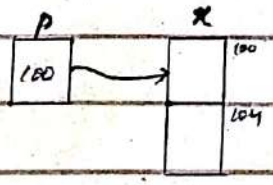
As `int a;`

`&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 :**

```
Rational & test = y;
test.num = 890;
```

y / test

890

→ Alias can't do everything what a pointer does.
Like alias needs to be initialized when declared.

```
inputRational (x);
inputRational (Rational &r)
```

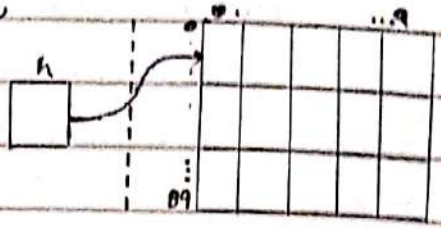
2-d Array:

On Heap:

```
int a = 90; b = 87;
int (*h)[b] = new int [a][b];
```

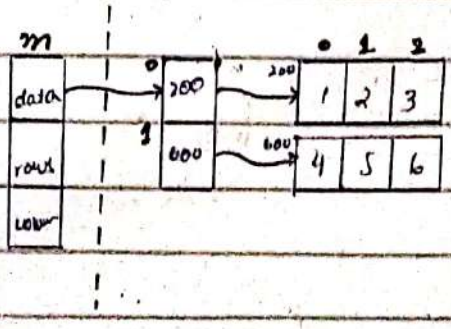
ERROR: Row size (b) can't be a variable.

```
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.

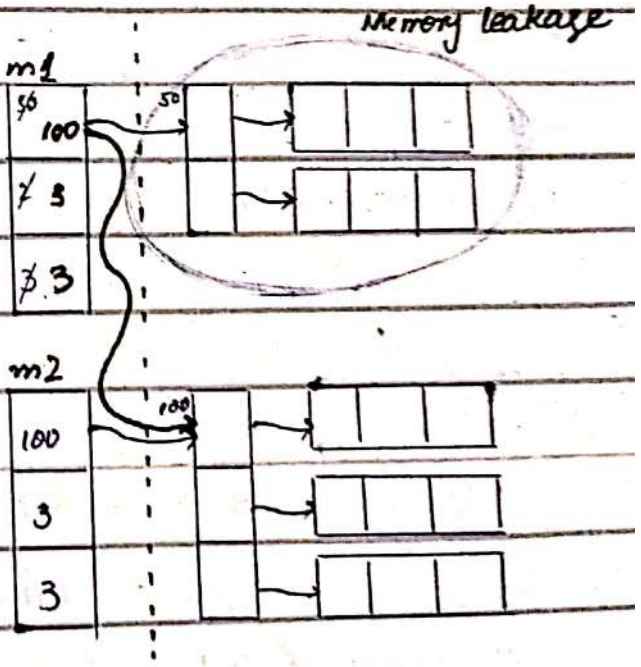
→ Exception Handling:

```
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;
}
```

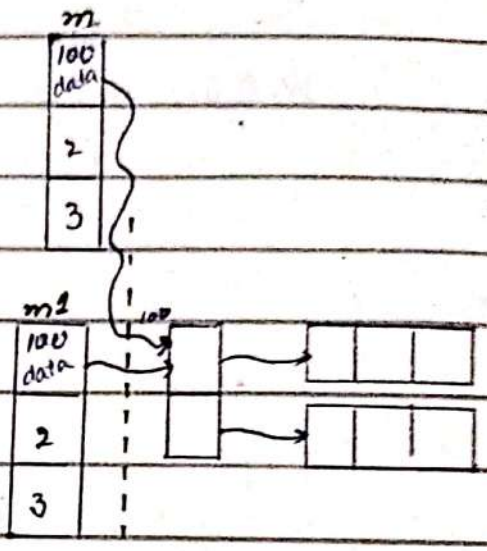
```
int main()
{
    Matrix m1;
    createMatrix(m1, 2, 3);
    Matrix m2;
    createMatrix(m2, 3, 3);
    m1 = m2;
}
```



→ 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;
    create Matrix(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.

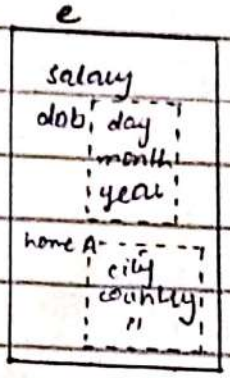
Lecture 05:

```
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.salary = 300.10;
    e.dob.day = 13;
    InputDate(e.dob);
}
```

Employee e

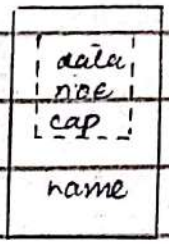



```
Date d[5];
d[1].month = 10;
InputDate(&d[2]);
```

	0	day month year
→	1	day month year
	2	day month year
	3	day month year
	4	day month year

```
d[2] = d[0];    // Members copied
```

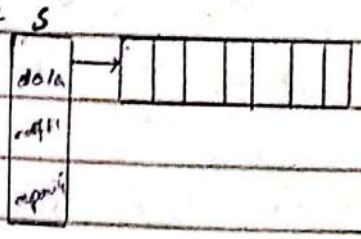
```
Struct Set
{
    int * data;
    int noe;
    int cap;
};

Struct NamedSet
{
    Set nset;
    char name[10];
};
```




```
void InputSet ( Set * s)
{
}

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

```
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] << " , " ;
}
```

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;
  cin >> temp;
  if (Temp == BOIL_POINT)
  {
  } ...
}
```

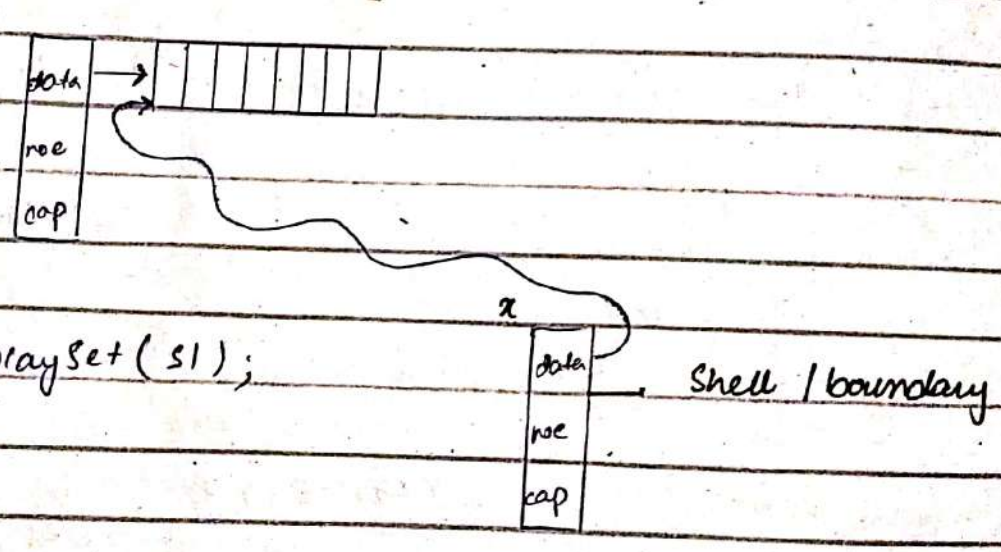
if not assigned 0, it will be automatically assigned value 101.

```
BOIL_PONT = 75;        // ERROR : BOIL_POINT is constant identifier.
```

```
enum Months { 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)
{
}
```

Arrays with Struct pass by Value :



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

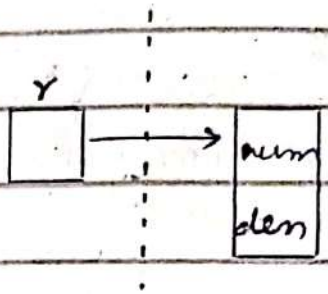
```
displayset ( createClone(s1));

set createClone ( set st)
{
    int * temp = new int [st.capacity];
    for (int i=0; i< st.noe; i++)
        temp [i] = st.data [i];
    st.data = temp;
    return st;
}
```

Aways inside 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;
```



Array:

```
Rational *p = new Rational[5];
p[0].num = 75;
delete[] p;
```

Lecture 01:

OOP

Object : Set of attributes with interrelated 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 : Unclear / 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 combined of different types of object.

→ **Association:** Object is associated with different types of object.

→ **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;        // For t1 parameterized constructor
    t1.printTwelveHourFormat();   will be called.
    t2.printTwelveHourFormat(); // For t2 default constructor
}                                 will be called.
```

↳ Delegate constructor

```
Time(int h, int m, int s) : Time()   → It will execute first
{
    setTime(h,m,s);
}

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 initialise 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.

```
~Array()
{
    if (data)
        delete [] data;
    data = nullptr;
    capacity = 0;
}
```

Call destructor:

```
this -> ~Array();
```

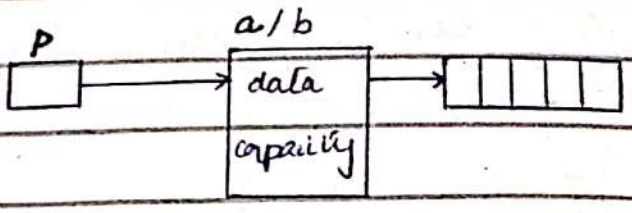
Sequence:

```
int a, b, c;
```

Constructor executes for a, b, c

Destructor executes for c, b, a

Pointer and Alias :



Array * p = &a ;

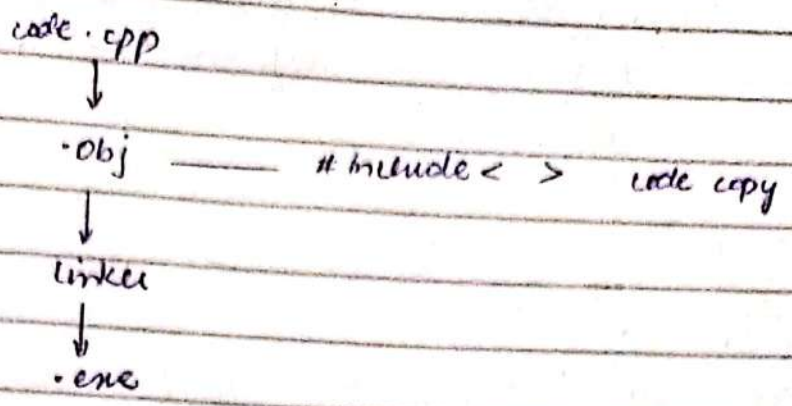
(*p) . getSet (1) = 32;

p -> 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:____ " → path of file

define

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

File.h / String.h

File name constant

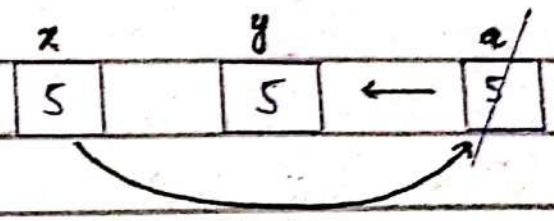
```
# ifndef  STRING_H
# define  STRING_H
   :
   :
# endif
```

ifndef → if not defined

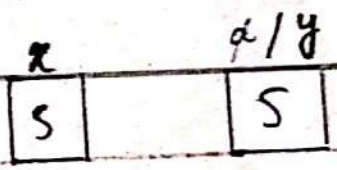
Lecture 11:

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 :-

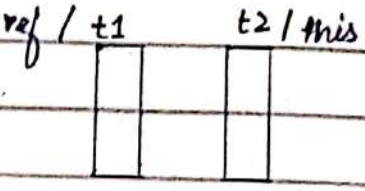
→ Default copy constructor does shallow copy.

```
className ( const className & )
Time ( const Time &ref)
{
    memcpy ( this , &ref , sizeof(Time));
}
```

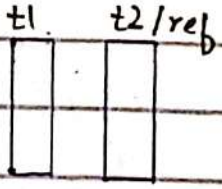
```
Time (Time ref)
{
}
```

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;
    const int c = 11 ;          // 1st stage
    const int y[3] = {1,2,3};   // 2st stage for Array

    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.
    }
};
```

direct initialization

Member initialization list

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

will execute first as defined in class

garbage

→ Member initialization list is prefered 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

→ Global function by default external linkage.

oop.h	oop.cpp
	if included here then its existance count.

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

→ Global variable

a.cpp	main.cpp

int var = 10; (global variable)	extern var; → Must Include
	int main()
	{
	cout<< var; // 10
	}

static int var = 10;	cout<<var; // Error

a.cpp	b.cpp	main.cpp

static int var = 10;	int var = 100	int main()
		{
		cout<< var; // 100
		}

→ Class

```
class MyMath
{
  public:
    int pow ( int b, int c)
    {
    }
};

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

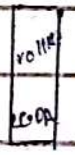
m → Size 1 byte (minimum memory)

→ Static — class level information

```
class MyMath
{
  public:
  static int pow()          // this pointer not exists.
  {
  }
};

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

a) data member

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

b) function

With functions can't use static and const together.

Mid Term

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

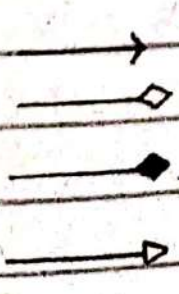
Object have their own lifetime

Shade / weak aggregation

e.g. Car and Driver

UML Notations:

Unified modeling language



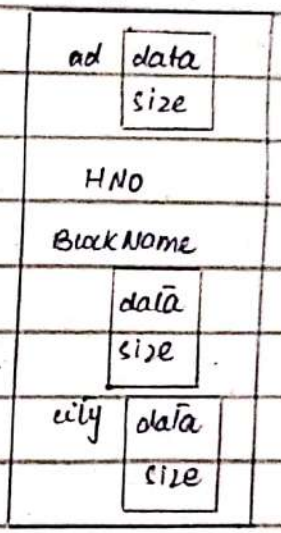
Aggregation:

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

Composition:

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

```
class Address
{
    string ad;
    int HNo;
    string BlockName;
    string city;
};

class Host
{
    Address guests;
};
```



→ Guests initialized first.

→ Destructor for ... guests then Host.

```
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:

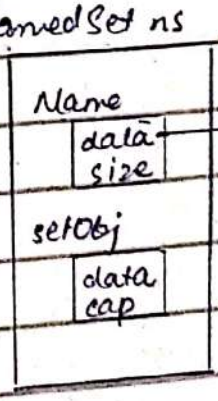
- For backward compatability
- Default access modifier

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);
}
```

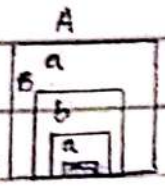


Prototype — Forward Declaration

```
class B;
class A
{   B * ptr;      → 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;
};
```

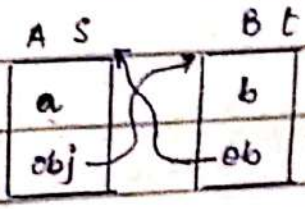
```
class B
{   int b;
    A * ob;
};
```

Possible

```
int main()
{   A s;
    B t;
}
```

t.ob = &s

s.obj = &t



Lecture 16:

Cyclic Reference

A.h

```
#include "B.h"

class A
{
  A(int)
}
```

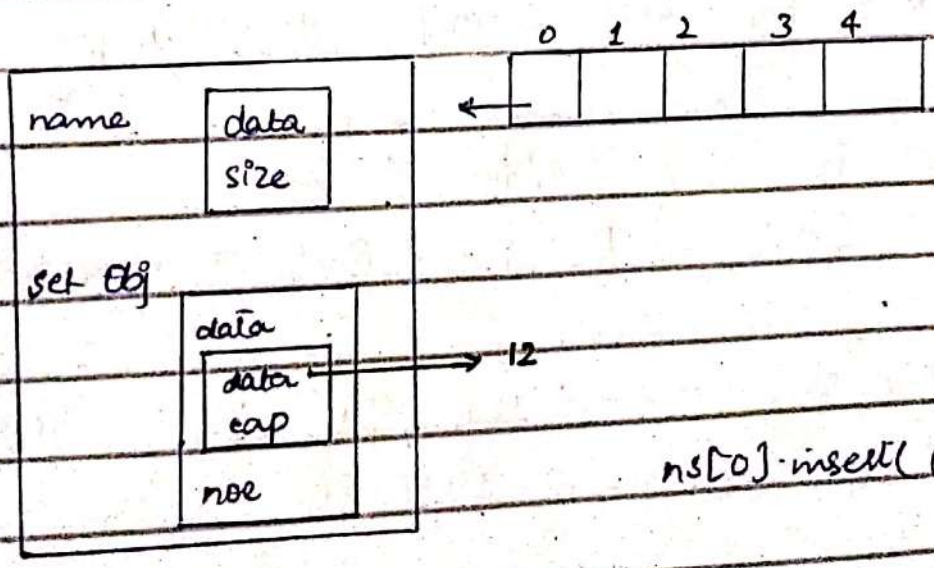
B.h

```
#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];
```




```
ns[0].insert(12);
```

```
Time *t = new Time;
    delete t;
NamedSet *p = new NamedSet[5];
    delete [] p;
```

```
class A
{
  int i,j;
public:
  A(int a)
  {
  }
  A(int a, int b)
  {
  }
  A()
  {
  }
};
```

A x[3]

x	
0	i j
1	i j
2	i j

Construction 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
```

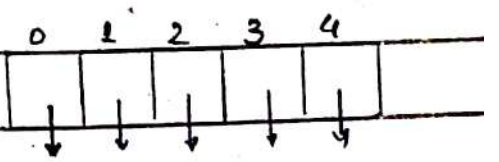
→ This kind of initialization is preferred only on Stack.
On Heap it is not so practical.

```
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.

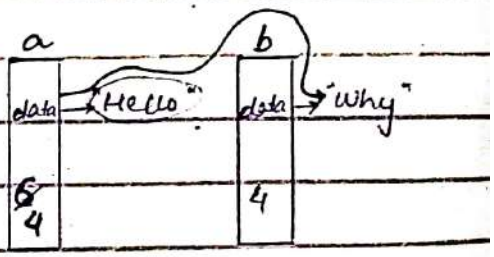
[] index operator — binary

! 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.

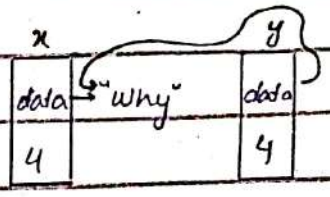
```
String a("Hello");
String b("Why");
a = b;   Assignment
```



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.operator[](i) = 'c'

Backend: a.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 -> ~String();
    if (!ref.data)
        return * this;
    size = ref.size;
    data = 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 ** qpool → 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

++s

s.operator++()

Set& operator++()
{
for(int i=0; i<noe; i++)
data[i]++;
return *this;
}

Postfix

s++

s.operator++(int)

Set operator++(int)
{
Set x(*this);
for(int i=0; i<noe; 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 unaccessible / can't be modified.

So in the case of unaccessible 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();
```


```
ostream & x = cout;
x << "Hello";
```

↓ Can do this by creating an alias only.

Can't create a copy

Like

```
ostream x = 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

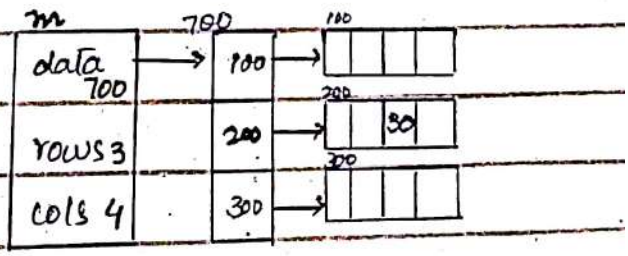
cin>>a // returns alias

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
    exit(0);
}
```

Not practical

→ Function call operator

operator()

S("Hello");

↳ function call object / functor

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 optimisation.
However compiler can't handle ambiguity.

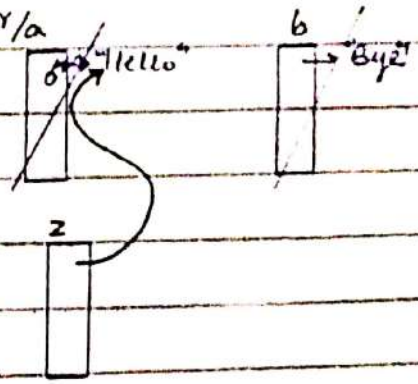
```
String f()
{
  String a("Hello"), b("Bye");
  int x = 10;
  if (x > 7)
      return a;
  else
      return b;
}
String z = f();
```

Here compiler can't decide which 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 :

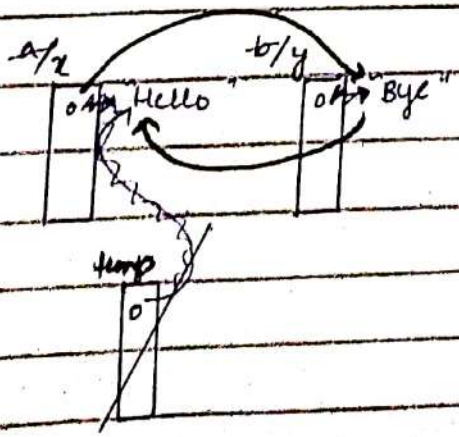
```
String z;
z = f();
```

```
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 :-

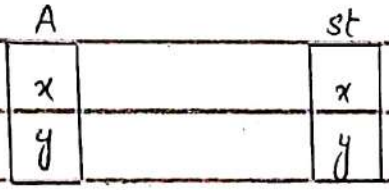
An algorithm that not more than one object of a class can be created.

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

A *p = (A*) new int[2];
p->x = 30;

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

Factory Method

```
Singleton * p = Singleton :: createObject();
Singleton a(*p);

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

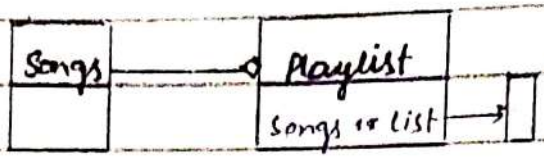
```
class A
{
    int x;
    int y;
    static A st;
}
```




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

Aggregation :- (has a)

- parent - child relationship
- Independant parent and child
- Shared aggregation



Composition :- (part of)

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

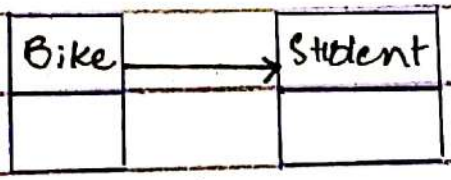

```
Class Subject
{   String Title;
    String Code;
}
```

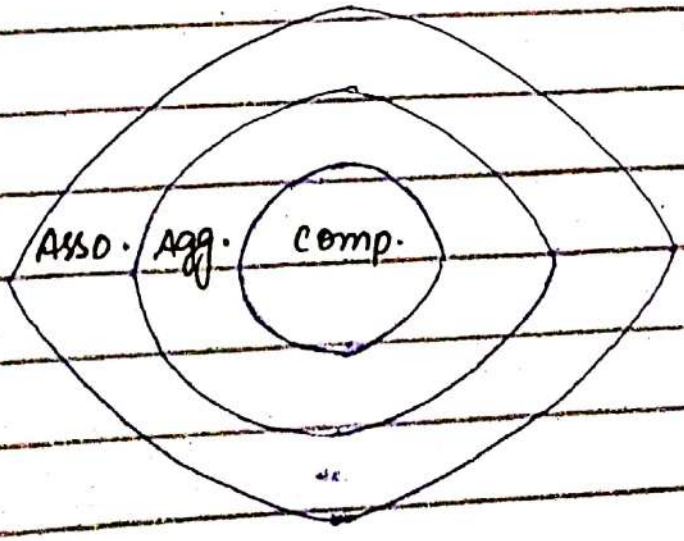
Association :- (knows a)

- Two-way relationship

```
class Student
{   Bike * b = nullptr;
public:
Bike * getBike()
{   return b;
}
```

```
void setBike(Bike *p)
{
    b = p;
    b -> setStudent(this);
}
}
class Bike
{
    Student *s = 0;
    public:
    void setPrice(Student *x)
    {
        s = x;
    }
}
```





Dependency :- (uses a)

---->

```
class Date
{
}

String getDateInFormat1()
{
}
```

Lecture 21

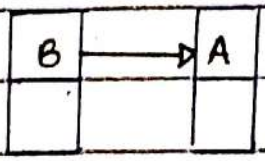
Inheritance :- (is a)

- 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 / derive parent / base

→ Child inherits parent.

```
B obj;
obj.h();
obj.f();
```

B obj

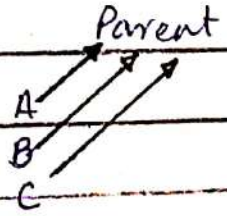
a	constructor first
b	destructor first

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

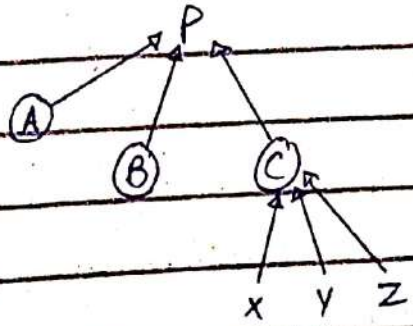
→ We can't use friend classes to reuse functionality of a class with additional features because we will have to write all the wrappers again.

Generalization :-

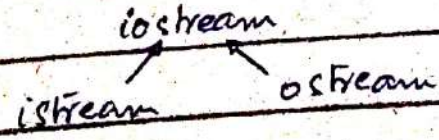
- inheritance



(Common Parent)



→ Direct Parent (C) of X, Y, Z.

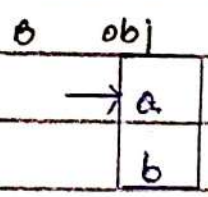


default Access Modifier

→ Public, (public, private, protected) Access Modifiers

```
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   // Logically wrong but implementation fine
{
    string name;
    string bGroup;
    Address adss;   // Right way
};
```

```
class B : public A

B obj;
obj.f();
```



→ Base class can refer to derive class object.

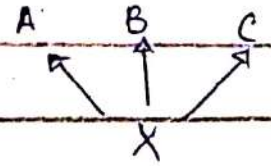
→ Strong class base

→ Strong object derive

→ Never manipulate derive class objects through base class.

Construction 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 main()
{
    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() {}
};
```

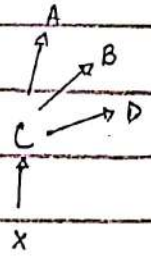
```
struct B : public A
{
    B() : A()  {}
    B(int x) : A(x) {}
};
```

or

```
{
    using A::A;   // inherits constructor
    B()  {}
    B(int x) {}
};
```

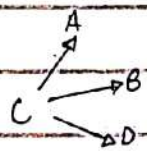
→ constructors are not inherited.

→ **Inheritance Pattern:**



C is direct parent of X. Through C A, B, D are also parents of X.

→ Multiple inheritance:

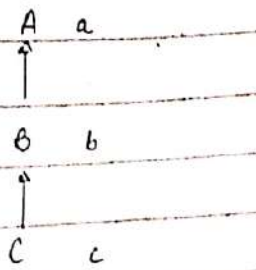


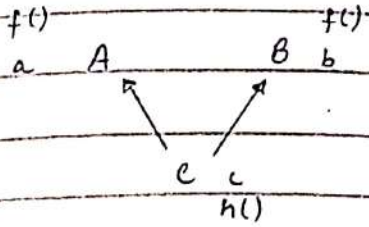
A, B, D are direct parents of C.

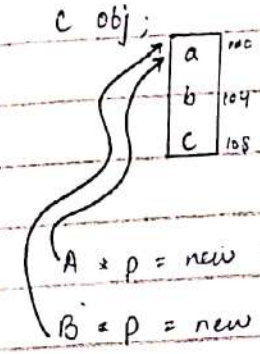
class C: public A, public B, public D

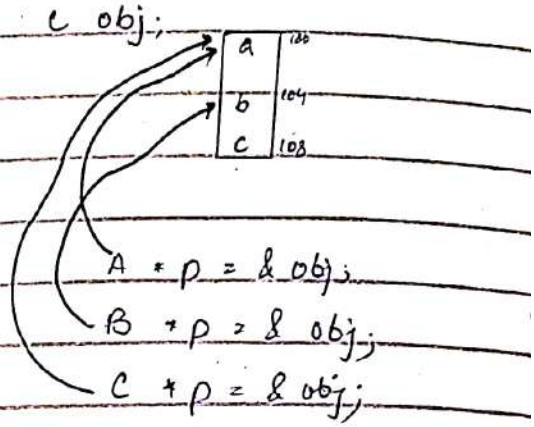
Constructor in order;

A, B, D, C





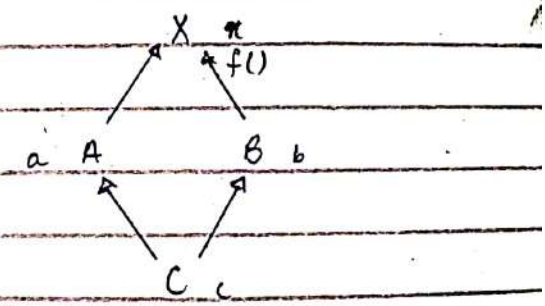




We will have to call function which are same explicitly.

C obj.A::f();

→ Diamond Inheritance :



class C : public A : public B

C obj;

(can't decide the path to take)

obj.x; // Error

obj.f(); // Error

Lecture 23

Polymorphism :-

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 nop = 0;
        int select = 0;
public:
```

```
void addPrinter (Printer * p)
{
    plist[nop] = p;
    nop++;
}

Print()
{
    plist[select] → print();
}
}

PrinterList pl;          // Global

int main()
{
    pl.addPrinter(new ABCPrint);
    pl.addPrinter(new XYZPrint);
}

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

→ We can achieve polymorphism by using virtual.

A f()
↑
B f()

pB → A::f();
pB → f();

pa → f();
pa → B::f(); // Error

Run-time 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 → x();
p → 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 = new B;
p -> f();        // 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.

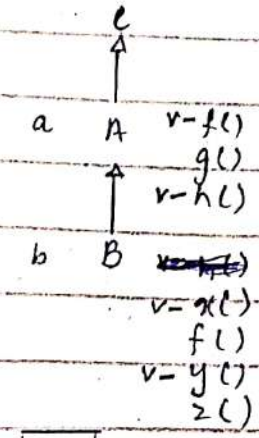
→ A function pointer should have same signature and return type of the function it is pointing to.

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

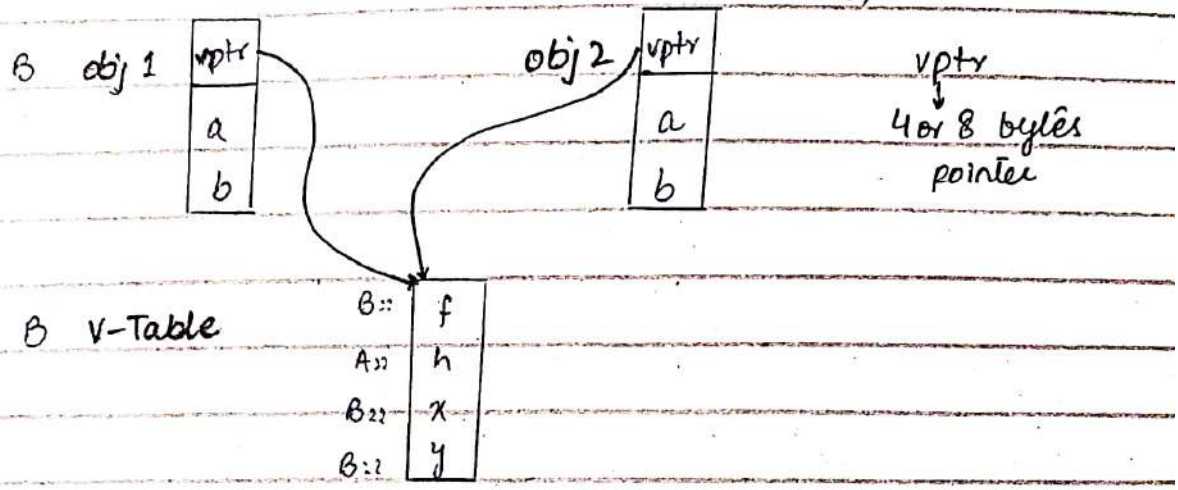
Lecture 24

V-Table (Virtual Table)

- Array of Function Pointers
- One v-table for a class containing atleast one virtual function.



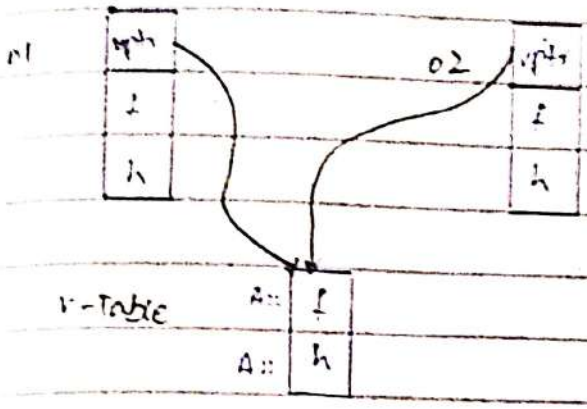
No v-table for C:




```
Now (A * p)
{
    p → h();        // A::h()
}
```

*(int *)p // Address of v-table

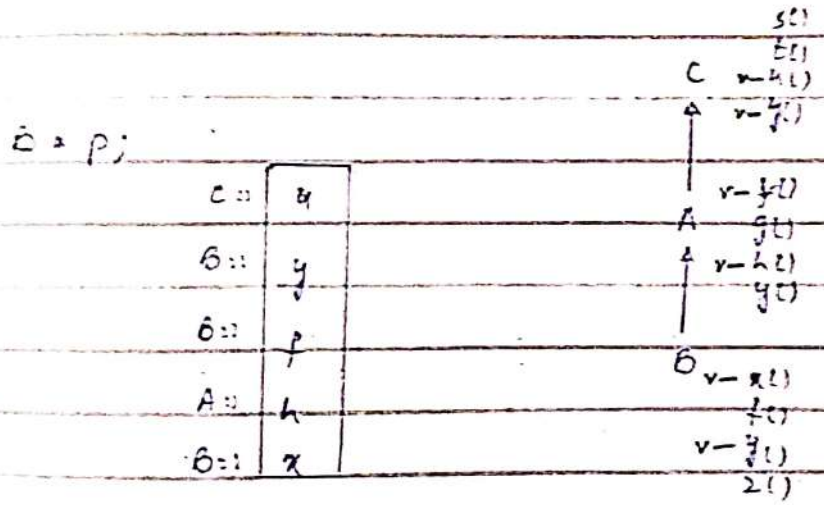
A o1, o2;



o1.f(); // No polymorphism

A *p = &o1;

p → f(); // Polymorphism occurs



→ v-Table for every class is made.

class A
{}

sizeof(A); // Byte allignment / Padding
Power of 2

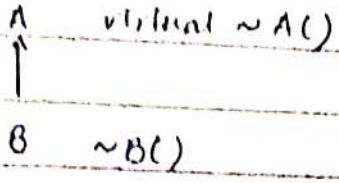
a.
intellisense


```
struct A                          struct B
{                                 {
    int a;                            int b;
    virtual void g() {}               virtual void y() {}
    virtual void h() {}               virtual void z() {}
}                                 }

int main()
{   A obj;
    B * p = (B*) &obj;
    p -> y();                     // A::g
}
```

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

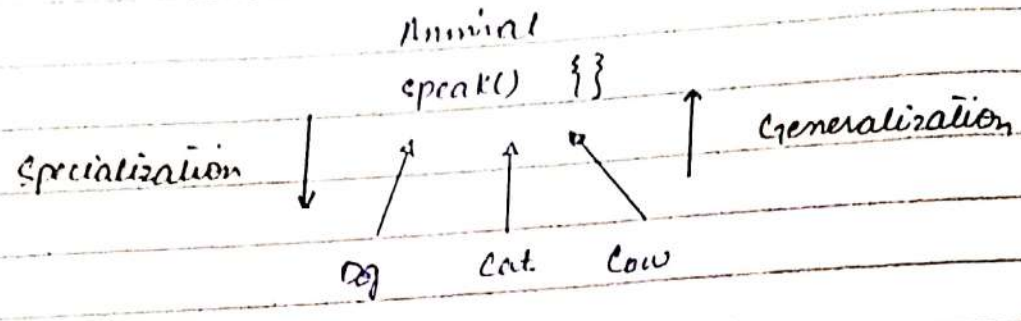
→ In case of constructor/destructor no polymorphic call.

→ Using virtual with destructor, all destructors 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.

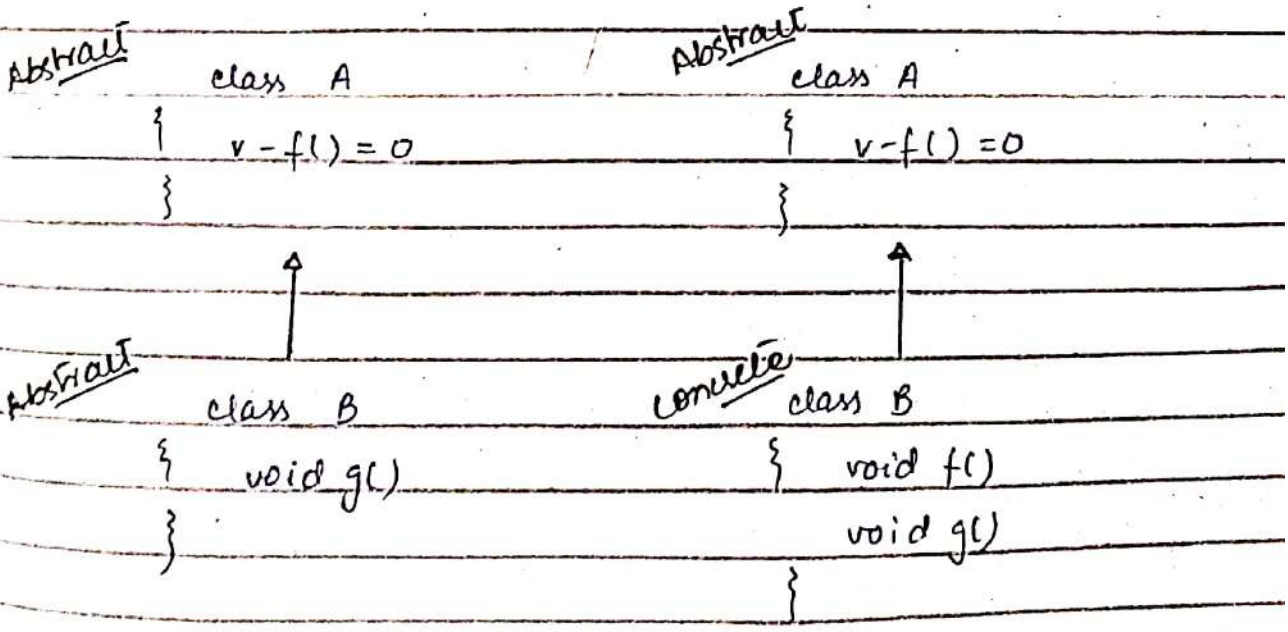
(Programming by contract)

→ Concrete class:
- Object can be made.

→ Abstract class: (Mixture)
- 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)
{
  // pure virtual functions only
}
```



→ Destructor :

If made pure virtual then other classes must write a destructor.

So, you have to provide destructor definition of base class globally.

→ Virtual Constructor :

(Only logically)

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

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

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

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

A * p = A / B / C

→ If class A has pure virtual function

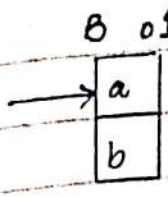
virtual A * clone() = 0;

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

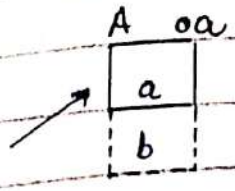
```
new A;          // Syntax Error
A obj;          // Syntax Error
```

```
char x[16];
A *p = (A *)x;      // Runtime Error
p->g();
```

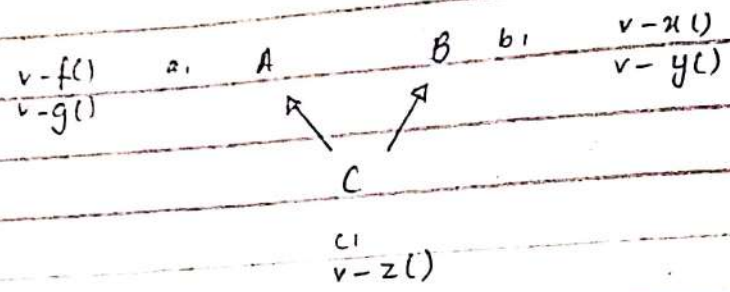
```
B o1;
A * p = &o1;
```

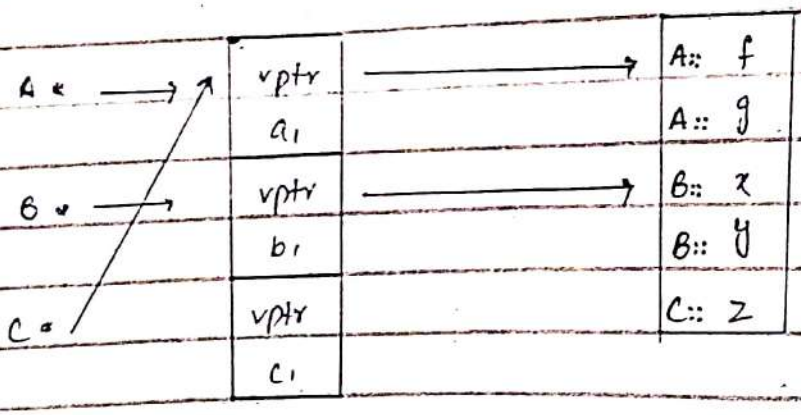



```
A oa;
B * pB = (B*) &oa;
pB -> b;   // Error
```

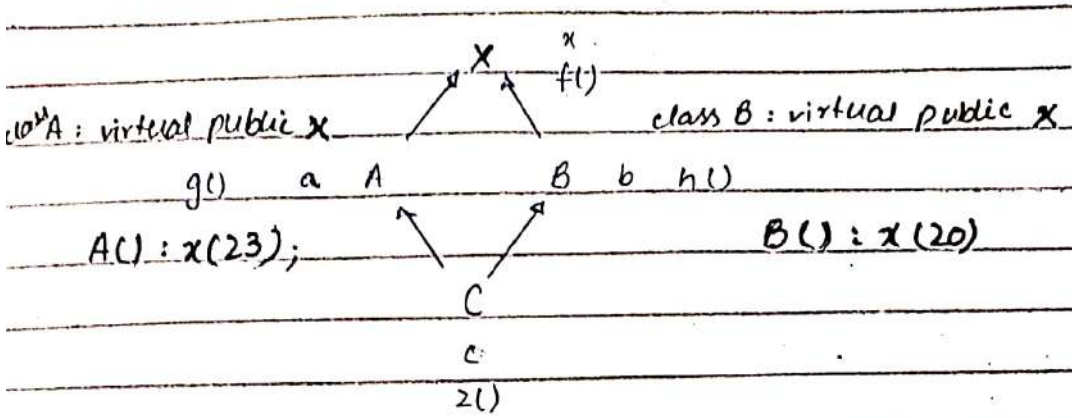


Multiple Inheritance :





Virtual Inheritance :



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

→ Before Virtual Inheritance :

x
a
x
b
c

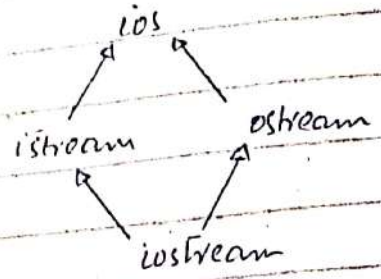
size = 20 bytes

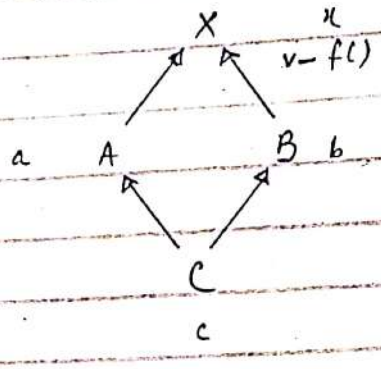
→ After virtual Inheritance :

class A : virtual public X
class B : virtual public X

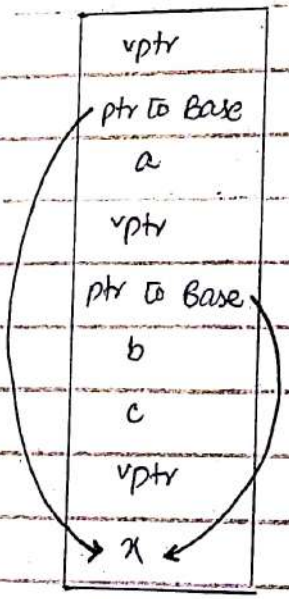
x
a
b
c

Lecture 26





→ layout on compiler



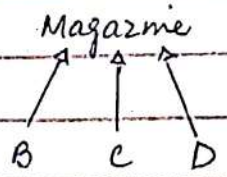
→ First constructor of virtual classes then Non-virtual classes will execute.

Type casting :

- const cast
- static cast
- dynamic cast

→ dynamic_cast< >( )

Only for polymorphic types



Media
~~Magazine~~ **m

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 = new B;

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

✓ B *q = static_cast<B*>(p); // p knows its static type
B *q = 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/function directly below until semicolon.

```
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;

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

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:
    //    //
};
```

// Backend:
class Array <int>
class Array <float>

```
Array <int> a(5);
Array <int> b(5);
    a = b;          // Shallow copy
Array <float> c(5);
    c = b;          // Error: type difference

Array <int> * p = &a;
Array <int> &q = a;
```

→ Copy Constructor:

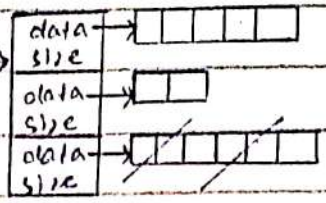
```
Array (Array <T> & ref)
```

Nested

```
Array <Array<int>> a;

class Array<Array<int>>
{
    Array <int> * data
```

```
class Array <int>
{
    int * data
```

Array<int> * data →



(2-d Array)

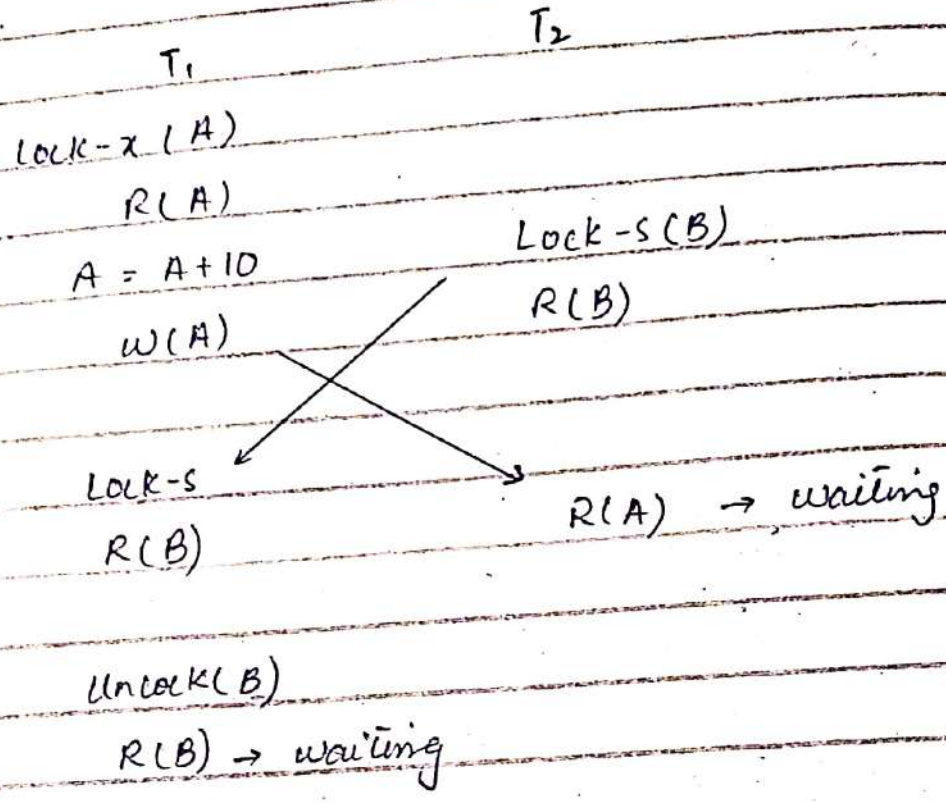
```
a[0][1];

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