3/8/23 C → Structured programming Language

C++ → Object Oriented Programming Language

Datatypes in C

- Primary data types
  - integer → long, short
  - Floating → signed, unsigned
  - Character
- Void
- Enumerated types
  - enum
- Derived data types
  - Array
  - pointer
  - Structure
  - Union

Enumerated datatype

enumerations ⇒ 'Specifically listed'

⇒ enum variable { —, —, —, —, — };

assign values to the array from 0 to n

⇒ User defined datatype madeup of integer constants, each with its own identifier.

⇒ defined by "enum".

4/8/23

#include <bits/stdc++.h> = # include <iostream> — very very important

bool → in C use header file <stdbool.h>; in C++ no need

↳ logical operator which returns True | False.

Ex in C

```c
# include <stdio.h>
# include <stdbool.h>
int main( )
{
    bool x = false;
    if (x == true)
    {
        printf ("The value of x is true");
    }
    else
        printf ("The value of x is false");
    return 0;
}
```

Q) Using bool datatype find odd or even.

```
#include <stdio.h>
#include <stdbool.h>
int main ()
{
    int x;
    printf("enter the x: ");
    scanf("%d", &x);
        if( x%2==0){
    bool y = true
    } else { bool y = false; }
    if ( y = true)
    {printf(" y is even ");
    }
    else
     printf (" y is odd ");
    return 0;
    }
```

Algorithm OE

1) If x%2 == 0
2) bool y = true
3) else
4) bool y = false
5) If y is true
6) print x is even
7) Else
8) print x is odd
9) return 0

Reference data type

It is a type of data type that stores a reference or memory address rather than the actual value. This reference points to the location in memory where the data is stored.

Ex:

```
#include <stdio.h>
void incrementByReference(int *numPtr)
{
    (*numPtr)++;
}
int main()
{ int value = 5;
  printf("Original value: %d\n", value);
  incrementByReference(&value);
  printf("Incremented value: %d\n", value);
  return 0;
}
```

Scope → Global Scope
→ Local Scope

Type casting / narrowing conversion.

→ converting one datatype into other.

Syntax

```
int x;
float y;
y = (float)x;
```

i) Implicit → without change the data type just assign the value of one data type to other.

ii) Explicit → Change the datatype of the variable and assign the variable to the similar datatype

7/8/23

Operators & statement

1) Arithmetic
2) Logical <, >, <=, >=
3) Relational
4) Ternary
5) Bitwise <<, >>, ~
6) Assigment (=), (+=), (-=), (*=), (/=)
7) Increment / Decrement
8) Comparison ==, !=
9) conditional

Syntax

```
(condition) ? exp : exp
```

Write a c program by using ternary find the given number is odd or even.

```
#include<stdio.h>
int main()
{
  int n;
  printf("Enter n: ");
  scanf("%d", &n);
  n%2==0 ? true : false;
  If(true)
  {
  printf("Even number");
  return 0;
  }
  Else
  printf("odd number");
  return 0;
}
```

Write a c program by using conditional operator find the biggest number among the 3 numbers.

```
#include <stdio.h>
int main()
{   int a, b, c;
    printf("Enter 3 numbers : ");
    scanf("%d, %d, %d", &a, &b, &c);
    int max = a>b ? (a>c ? a : c) : (b>c ? b : c);
    printf("max = %d", max);
    return 0;
```

Types of selection statement -

1) selection control statement
2) Insertion control statement
3) case control statement
4) Jump control statement.

Write a program to find the year is leap year or not.

```
#include <stdio.h>
int main()
{   int year;
    printf("Enter the year : ");
    scanf("%d", &year);
    If (year % 4 == 0 && year % 100 != 0)
    {printf("leap year");
     return 0;
    }
    else (year % 400 == 0 && year % 100 != 0)
    {  printf("leap year");
    }return 0;
```

int isleapyear = (year%4==0 && year%100!=0) || (year%400==0)

Write a C program to find Uppercase, Lowercase & digits in an string.

```
#include<stdio.h>
#include<string.h>
int main()
{
    char ch[100];
    printf("Enter the string: ");
    scanf("%c", &ch);
    ch = (ch>='A' && ch<='Z') || (ch>='a' && ch<='z')
        || (ch>='1' && ch<='9');
    switch (ch)
    {
    case 1 : printf("Upper case");
             return 0;
    case 2 : printf("Lower case");
             return 0;
    case 3 : printf("Digits");
             return 0;
    default : puts("Invalid");
             break;
    }
    return 0;
}
```

10/8/23 Write a C program to search an element from an array and sort the array elements.

```
#include <iostream>
using namespace std;
const int maxsize = 100;
int linearsearch (int A[maxsize], int x)
{
    for (int i=0; i<=n; i++){
        If (A[i] == x)
        { return i;
        }
        Else { return 0; }
}
int Binarysearch (int A[maxsize], int x, int n);
{
    int low = 0, high = n;
while (low <= high)
    { int mid = low+ high/2;
while(low <=high)
        If (mid == x)
          return 1;
        Else If (mid < x)
          low = mid+1;
        Else If (mid > x)
          high = mid-1;
        Else
          return 0;
    }
}
int arraysort (int A[maxsize], int n)
{ for(int i=1; i< n; i++) {
    int key = arr[i];
    int j = i-1;
    while (j >=0 && arr[j] > key) {
      arr[j+1] = arr[j];
      j--;
    } arr[j+1] = key;
  }
}
```

```
int main()
{
  int n, A[maxsize], op, x;
  ~~char op~~
  cout << " Enter the size of array: ";
  cin >> n;
  for (int i=0; i<n; i++)
  {
    cout << " Enter the array element " << i << ":";
    cin >> A[i];
  }
  cout << " original array \n";
  for (int i=0; i<n; i++)
  { cout << A[i] << ", "
  cout << " \n Sorted array \n ";
  for (int i=0; i<A; i++)
  { cout << A[i] << ", ";
  }
  cout << " Enter the search you want \n
          1. Linear search \t 2. Binary search \n";
  cin >> op;
  switch (op)
  {
    case 1: ~~int~~ n.
            cout << "Enter the element
                 you want to search:";
            cin >> x;
            If (linearsearch(A, x))
            { cout << " Element found
              in the array ";
            }
            else
              cout << " Element found
                 in the array ";
            break;
```

```
case 2 : cout << "Enter the element you want
          to search : " ;
         cin >> x ;
         If ( Binarysearch ( A, x)
         { cout << "Element found in the array" ;
         }
         Else
         cout << "Element Not found in the
              array" ;
          break ;
Default : cout<< Error! Invalid Input " ;
          break ;
  }
  return 0 ;
}
```

```
Void swap( int * r, int *s)
{ int temp = *r ;
    *r = *s ;
    *s = temp ;
     return ;
}
int main()
{ int P = 7, q = 9 ;
   swap ( &P, &q ) ;
   cout << P << q ;
   return 0 ;
}
```

Break

come out from the whole loop

Continue

Just skip the current iteration

11/8/23

Dangling loop → Without closed parenthesis (})

Diff b/w malloc and calloc

↓

It allocates the memory according to the largest element.

How to remove the allocated memory?

— using **free of ( ) funition.**

Questions

Inline funtions →
- pointer to constant
- constant pointer
- constant pointer to constant

pointer to constant

```
#include <iostream>
using namespace std;

int main() {
    int high = 100;
    int low = 66;
    const int * score = &high;
    cout << *score << "\n" ; // print 100
    score = &low ;
    cout << *score << "\n"; // o/p 66
    return 0;
}
```

can't able to change the value of score manually.

Constant pointer

```
#include<iostream>
using namespace std;
int main()
{
    int a[90];   a[90];
    int b[50];   b[50];
    int * const ptr[&a];
    cout << *ptr << "\n";
    cout << ptr << "\n";
    *ptr=56;
    cout << *ptr << "\n";
    cout << ptr << "\n";
    return 0;
}
```

Constant pointers to constant

```
#include<iostream>
using namespace std;
int main()
{
    const int a[50];
    const int b[90];
    const int * const ptr[&a]; // *ptr=90
    cout <<ptr << "\n";
    cout << *ptr << "\n";
    return 0;
}
```

Function overloading

It is a feature in C++ that allows you to define multiple functions with the same name but different parameters lists.

Function overloading is a form of compile-time polymorphism.

```
#include<iostream>
using namespace std;
void print(int num){
cout<<"Integer: "<<num<<endl;
}
void print(double num){
 cout<<"Double: "<<num<<endl;
}void print(const char *str){
  cout<<"String: "<<str<<endl;
 }
int main(){
 print(5);
 print(3.14);
 print("Helloworld");
  return 0;
 }
```

float & double both are always same.

14/8 Write a c++ program to find area of diff. geometric shapes using function overloading

```
#include<iostream>
using namespace std;
const int pi = 3.14;
void Area(int r){
   Area = pi * r * r;
   cout<<"Area of circle = "<<Area;
   return 0; }
void Area(int l, int b){
     Area = l * b;
    cout<<"Area of rectangle = "<<Area;
    return 0; }
```

```
Void Area (int a){
    Area = a*a;
    cout << "Area of square = " << Area;
}
Void Area (int b, int b)
    Area = 1/2 * b*b;
    cout << "Area of triangle = " << Area;
}
void Area (int a){
    Area = a*a*a;
    cout << "Area of cube = " << Area;
}
int main (){
int r, l, b, a, h, base, aa;
    int op;
    cout << "Available shapes: \n1.circle \n2. rectangle
        \n3. square \n4. triangle \n5. cube";
    cout << "\n enter your choise";
    cin >> op;
    Switch (op){
    case 1: cout " Enter the radius
            of circle: "
            cin >> r;
            Area (r);
            break;
    case 2: cout " Enter the length &
            bredth of rectangle: "
            cin >> l >> b;
            Area (l, b);
            break;
    case 3: cout " Enter the length of
            the square: "
            cin >> a;
            Area (a);
```

```
case 4: cout "Enter the height & base of
                the ole:";
        cin >> h >> base;
        Area (h, base);
        break;
case 5: cout "Enter the length of the
                cube: ";
        cin >> aa;
        Area (aa);
        break;
default : cout "Error! Invalid ~~op~~ choice";
        break;
}
return 0;
}
```

collection of names & definition which helps to separate identifier from similar names.

definition of

To use

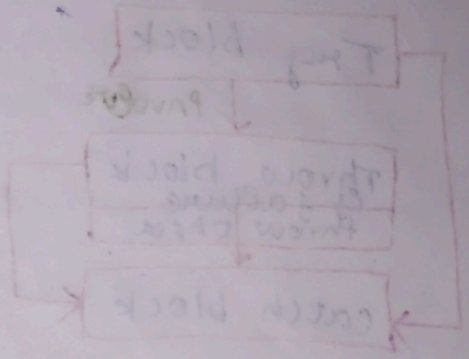
User defined namespaces

```
using namespace std;
namespace n1 {
    int x = 2;
    void fun()
    { cout << "This is function of n1 ";
    }}
namespace n2 {
    int x=5;
    void fun() {
    cout << "This is function of n2" ;}
    }
```

```
int main
cout <<
n1 : :
cout <<
n2 : :
retur
}
```

```
int main() {
cout << n1 :: x << endl;
 n1 :: fun();
cout << n2 :: x << endl;
 n2 :: fun();
 return 0;
}
```

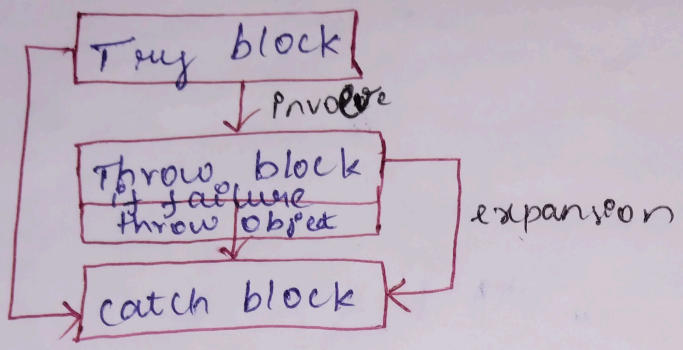
Scope resolution operator.



17/8/23

Error / Expansion Handling

→ It is managed by Try, Throw, catch




```
void main()
{ int a, b;
cout << "Enter two numbers : ";
cin >> a >> b;
try { if (b==0)
        throw b;
      else
        cout << a/b;
    }
catch (int x)
  { cout << "2nd operand can't be 0";
  }
}
```

Digraph and Trigraph sequences → Supported by ANSI C

↳ It is supported by ISOC 94 mode

Digraph

<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

Ex:- Tri

??

int

??

pr

??

Trigraph sequence

??=

??C

??/

??)

??

??<

??!

??>

??-

PRE -

P

d

int

{

pr

}

Ex:- Trigraph

```
??=include <stdio.h>
int main()
??<
printf("Hello world");
??>
```

Trigraph sequence	Equivalent character
??=	#
??(	[
??/	\
??)	]
??	^
??<	{
??!	|
??>	}
??-	~

PRE-PROCESSOR Directive

```
#include <stdio.h>
#define AREA (x,y) (x*y)
int main()
{ int a=4, b=2;
printf("%d", AREA(a+2, b+3));
}
```

#undef
To remove all defn created by macros

Ex:-
```
#define pi 3.1415
printf ("Area of circle", pi*2*2);
#undef pi
#define pi 3.14
printf ("circumference", 2*pi*1.5);
```

```
#define num 7
int s1 = num * num;
#define num 8
int s2 = num * num;
int main ()
{
  printf (s1);
  printf (s2);
}
```

pre defines value at any where

```
#define stdid 101
#undef stdid
int main()
{
#ifdef stdid
  printf ("[redacted] stdid exist");
#else
  printf ("Not exist");
#endif
}
```

o/p
Not exist

```
#define Fac 5
int main()
{
#if Fac < 5
    pf("Less than 5");
#elif Fac = 5
    pf("Equal to 5");
#else
    pf("Greater than 5");
#endif
}
```

```
#define gfg 7
#if gfg > 200
  #undef gfg
  #define gfg 200
#elif gfg < 50
  #undef gfg
  #define gfg 50
#else
  #undef gfg
  #define gfg 100
#endif
```

O/P

50

18/8/23

Pragma Warn Directives

(warning)

↳ Compiler specific

→ Used to hide warning menages during compilation.

```
# pragma warn -rvl /* return value */
# pragma warn -par // parameter never used
# pragma warn -rch // unreachable code
```

```
int show (int x)
{
   printf ("Hi");
}
int main()
{
   show (10);
   return 0;
}
```

pragma startup → These directive specify the compiler which function need to run before the main

pragma exit → Exit

```
void func1();
void func2();
```

```
# pragma startup func1
# pragma exit  func2
void func1(){ printf ("Inside func1");}
void func2 (){ printf("Inside func2");}

main ()
{ printf ("Inside main");

}
```

CLA

Syntax

Ex:-

CLASS & OBJECT in C++

Syntax

```
class classname{
    data members;
    member func;
};
```

Ex:-

```
class Room{
public:
    double length;
    double breadth;
    double height;
    double calculate_Area()
    { return length*breadth;
    }
    double calculate_volume()
    { return length*breadth*height;
    }
};
int main()
{   Room r1;
    r1.length=42.5;
    r1.breadth=30.8;
    r1.height=19.2;
    cout<<"Area of ROOM = "<<r1.calculate_Area();
    cout<<"Volume of ROOM = "<<r1.calculate_volume();
    return 0;
}
```

data members

member func

21/8/23

#line directive

↳ tell the preprocessor to set the compiler's reported values for the line number & filename to a given line number

Syntax

line digitsequence "filename"
↳ optional

you can alter
↳ line number &
↳ filename

"hi.c"

```
#include <stdio.h>
int main(){
printf("This code is online %d, in file %s\n",
                    __LINE__, __FILE__);
#line 10
printf("This code is online %d, in file %s\n",
                    __LINE__, __FILE__);
#line 20 "hello.c"
printf("This code is online %d, in file %s\n",
                    __LINE__, __FILE__");
```

used to print the current line

used to print the current line

Unit-02

Constructor in C++

↳ special method involved automatically at the time of object creation.

→ constructor has name similar to class.

→ NO return value.

Syntax :-

```
class name( list of parameters)
{
}
```

Outside class

```
classname :: class name ()
{
}
```

using constructor inside the class.

```
#include<iostream>
using namespace std;
class student{
public:
    int rno;
    char name[50];
    double fee;
    student(){
        cout<<"Enter Roll no:";
        cin>>rno;
        cout<<"Enter Name :";
        cin>>name;
        cout<<"Enter fee :";
        cin>>fee;
    }
    void display(){
        cout<<rno<<name<<fee;
    }
};
int main()
{
    student s;
    s.display();
    return 0;
}
```

object → constructor

function

creation of object

calling the function.

using constructor outside the class

```
class student{
public:
    int rno;
    char name[50];
    double fee;
    student();
    void display();};
student::student()
{ cout<<"Enter Rollno";
  cin>>rno;
  cout<<"Enter name";
  cin>>name;
  cout<<"Enter fee";
  cin>>fee;
}
void student::display()
{ cout<<rno<<name<<fee;
}
~~void student::~~ int main(){
  student S;
  S.display();
  return 0;
}
```

```
class Bank{
public:
    int pin, ~~a~~ cash, op
    long int acc-no;[10]
    ~~char name[50];~~
    Bank();
    void display();};
```

```
Bank::Bank(){
  cout << "Enter your account Number : ";
  cin >> acc_no;
  cout << "Enter PIN : ";
  cin >> pin;
  cout << "Enter your choice \n1- cash deposit\n
          2- Cash withdrawal ";

  cin >> op;
  switch(op){
  case 1 : cout << "Enter the amount of deposit"
           cin >> cash;
           break;
  case 2:  cout << "Enter the amount of withdrawal"
           cin >> cash;
           break;
  }
}
void display
void Bank::display()
{
  cout << acc_no << cash;
}
int main()
{
  Bank B
  B.display();
  return 0;
}
```

parameterised constructor

```
class rectangle {
  public :
    double length();
    double breadth;
rectangle (double len, double h)
{
    length = ln;
    breadth = h;
}
double area () {
   return length * breath;
}
};
int main () {
rectangle r1 (8.4, 8.6);
rectangle r2 (60.0, 46.0);
cout << " Area 1 " << r1. area ();
cout << " Area 2 " << r2. area ();
}
```

24/8/23 Copy constructor

↳ It is used to create an object by initialising of its object of same class shall be created previously.

```
class Distance {
  int feet;
  float inches;
  public:
    Distance() { // constructor
       feet = 0;
       inches = 0.0;
    }
    Distance (int f, float in) { // parameterised constructor
       feet = f;
       inches = in;
    }
```

// constructor overloading

```
void disp(){
    cout << " feet:" << feet << "inches << inches "<<"\n";
}
~Distance(){ //Destructor
    }
};
int main(){
    Distance d2(5,4.5);
    d2.disp();
    distance d3(d2); // one argument constructor
    d3.disp();                    (or)
}                                 Default copy constructor
```

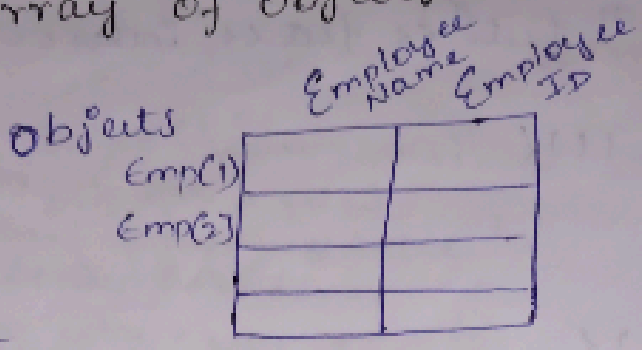
↳ by a default compiler calls this destructor function, eventhough you didn't call this in main func. It free up the memory

---

OOPs concepts

- → class
- → object
- → Data abstraction
- → Inheritance
- → polymorphism
- → Dynamic binding
- → Menage passing

25/8 Array of objects



Syntax :-

Classname objectname [no. of objects];

Employee Emp[30];

```
class Emp {
    int id;
    char name[30];
    public:
        void getdata();
        void putdata();
};
void emp::getdata(){
    cout << "Enter ID: ";
    cin >> id;
    cout << "Enter name: ";
    cin >> name;
}
void emp::putdata(){
    cout << id << " ";
    cout << name << " ";
    cout << endl;
}
```

```
int main () {
  emp e[30];
  int n;
  cout << "Enter No. of Employees : ";
  cin >> n;
  for ( i=0; i<n; i++) {
      e[i].getdata();
  }
  cout<<"Employees Data : ";
  for ( i=0; i<n; i++) {
      e[i].putdata ();
  }
return 0;
}
```

28/8/23 Dynamic Memory Allocation in C++

New operator

↳ Used to allocate memory for a single / array of objects on heap.

Syntax:- type * pointer = new type;

for array :- type * pointer = new type[size];

Ex:- int * value = new int;
int * array = new int[10];

Delete operator

↳ used to free memory allocated using new

↳ used to prevent memory leak.

Syntax:- delete value;
delete [] array;

```
int main(){
  int *variable = new int;
  int size=100;
    *variable = 42;
   int *array =new int[size];
for (int i=0; i< size; i++)
{  array[i] = i*10;
}cout << "array values";
 for (int i=0; i<n; i++)
 {  cout << array[i];
 }
 delete [] array; delete variable;
  return 0;
}
```

Dynamic Datastructure: {std:: Vector}

↳ Here dynamic array created with pointers & array is resized using vector

```
#include <vector>
{
 std::vector <int> dynamicarray;
 dynamicarray.push_back(10);
 dynamic array.push_back(20);
for (int num : dynamic array)
 {  std:: cout << num << " ";
 }
 return 0;
}
```

Element declaration → container expression

Dynamic array functions

- push_back → Add element at last
- pop_back → remove element at last
- insert → insert element at specified position
- erase → Remove element at specified position

EXAMPLES

```
dynamic.pop_back();
dynamic.push_back(10);
dynamic.insert(dynamic.begin()+1,15);
dynamic.erase(dynamic.begin()+2);
```

11/9/23

IO Manipulation

i) setprecision()

```
#include<iomanip>
int main()
{
    double num = 3.14234167;
    cout<<"Before setting precision";
    cout<<num;
/* using set precision */
    cout<<"set precision to 5:"<<
                    setprecision(5);
    cout<<num;
    return 0;
}
```

ii) setw()

```
int main()
{ cout<<setw(10); is to set the
    cout<<77;      width of the
                   output screen
}
```

---77

iii) Set base()

```
cout << setbase(16);
cout << 110;
```

16 → Hexadecimal
8 → octal
10 → decimal
2 → Binary

iv) Set fill()

```
int main()
{
cout << setfill('*');
cout << setw(10);
cout << 77;
return 0;
}
```

→ use anything you want
→ fill the empty space/ white space in the output

v) get c vi) put c

vii) scanf("%.[^\n]", str)

→ it accepts the string including spaces until you press 'Enter'.

```
int main(){
char str[50];
printf("Enter a word:");
scanf("%s", str);
printf("Entered word is %s", str);
while(getchar()!='\n'){
printf("Enter the sentence:");
scanf("%[^\n]", str);
printf("Entered sentence is %s", str);
return 0;
}
```

→ it only accepts the string
Ex: hello world
↓
only Hello

11/9/23 AFTER CIA I

Inheritance

↳ We can able to reduce redundancy

↳ One class can able to use the properties of already existing class; derived class, base class.

Syntax

Single level : class A
{
class B : public A
{
};
};

Example

```
class user {
public :
    string name;
    int    pswd;
    void get () {
    cout << "Enter username & password:";
    cin >> name >> pswd;
    }
    void verify () {
if ( (name == "SASTRA") && (pswd = 12345)
    cout << "Login successful";
    else
      cout << " Invalid User";
    }
};
```

fout << setfill('0') << 77

```
class student : public User {//derived class.
public:
  int sem;
  string branch, program;
  void getst(){
    cout<< "Enter branch & prgm:";
    cin >> branch>>prog;
    cout << "current sem";
    cin>> sem;
  }
  void display(){
    cout << "Reg.no" << pswd <<
    "Degree" << branch << "Pgm" << sem;
  }
};
int main(){
student s;
  s.get();
  s.verify();
  s.getst();
  s.display();
  return 0;
}
```

21/9/23 Operator Overloading

+, -, *, /, %, ^, &, |, =

new, delete, [ ], <<, >>

cannot be overloaded :-

- Scope resolution operator (::)
- Ternary operator / conditional operator (?:)
- Member access / dot (.)
- pointer to num (*)
- size of

```
int main()
{ int a, b c;
  a = 5;
  b = 10;
  c = a+b;
  cout << a;
  string a = 'xyz';
  string b = 'Abc';
  cout << a+b;
}
```

operator ++ Pre Increment

```
class Time
{
  private:
    int hrs;
    int mins;
  public:
    Time(int h, int m)
    { hrs = h;
      mins = m;
    }
  void operator ++ () {
    mins ++;
    if(min > 60) {
      hrs ++;
      mins -= 60;
    }
  }
  void display() {
  cout << "hrs" << chrs << ":"
  << mins;
};
int main(){
  Time t1 (6,30);
    ++t1;
  t1.display();
}
```

22/9/23

BINARY OPERATOR OVERLOADING

```
class Sum {
    int a;
public:
    void input() {
        cout << "Enter a" : ;
        cin >> a;
    }
Sum operator+(Sum c) {
    Sum t;
    t.a = a + c.a;
    return t;
}
void display() {
    cout << "The result is " << a;
}
};
void main() {
    Sum S1, S2, S3;
    S1.input();
    S2.input();
    S3 = S1 + S2; // If run this code without operator + then it shows Error of operator cannot be Overloaded.
    S3.display();
}
```

Assignment operator Overloading

```
class complex
{ int real, img;

  public:
    void input()
    cout << "enter real : ";
    cin >> real;
    cout << "enter img : ";
    cin >> img;
  }
void operator = (complex c)
{
    real = c.real;
    img  = c.img;
}
void display()
{
  cout << "Result is : " << real << "+i" << img;
}};
void main()
{
  complex  s1, s2;
  s1.input();
  s2 = s1;
  s1.display();
  s2.display();
}
```

23/9/23

Multiple Inheritence

Syntax:-

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

```
class A{};
class B : public A{};
main()
{}
```

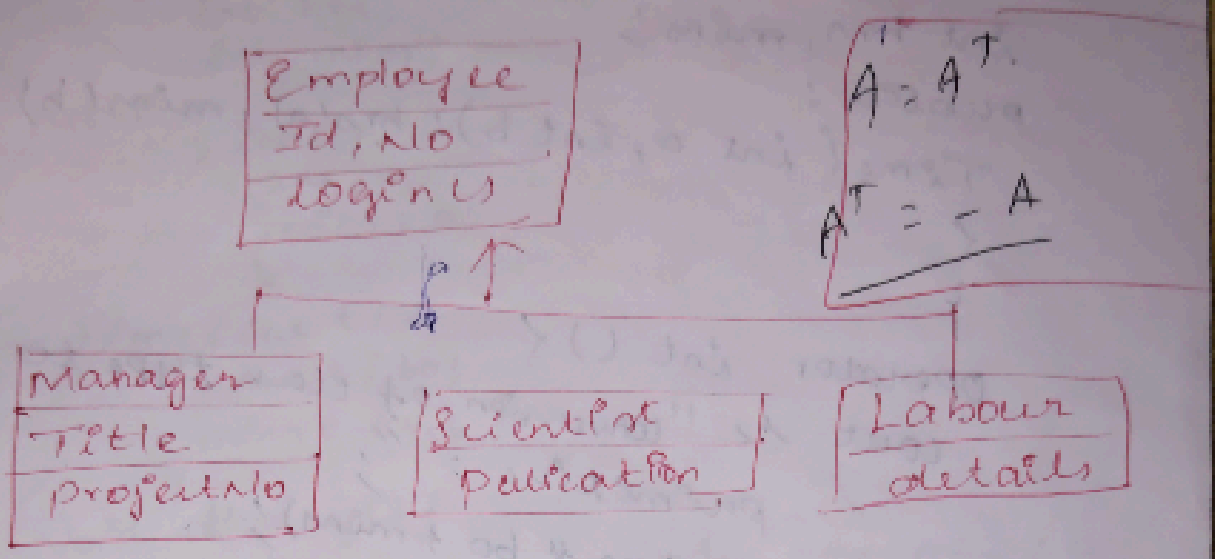
Multilevel Inheritence

Syntax:-

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

25/9/23

Class Hierarchies




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

Type conversion

i) conversion of user defined (or class object) to Basic primitive data type.

ii) Primitive to user defined

iii) class to class type conversion.

1) obj : int a
2) int a = obj
3) obj = obj

```
class Time{
    int hrs, mins;
public:
    Time(int a, int b): hrs(a), mins(b)
    {
    }
    operator int (){
        cout << "conversion of class type to
                primitive: ";
        return (hrs * 60 + mins);
    }
};
int main (){
    int deviation;
    Time t(2, 10);
    duration = t;
    cout << "Total minutes are" << duration;
    cout << "2nd method operator overloading";
    duration = t.operator int(); // Here class
                                 type is
                                 converted into
                                 int type.
    Cout << "Total minutes are" << duration;
    return 0;
}
```

19) Primitive to Userdefined

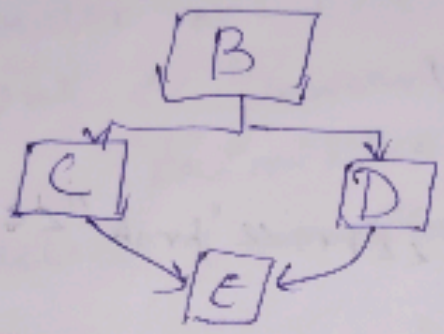
```
class Time {
        int hrs;
        int mins;
public:
        Time()
        { hrs = 0;
          mins = 0;
        }
    Time(int t) {
          hrs = t / 60;
          mins = t % 60;
    }
    void display() {
    cout << "Time = " << hrs << "hrs:" << mins <<
        "min\n";
    }
};
int main() {
    Time T1;
    int dur = 95;
    T1 = dur; // conversion of int type to
                class type
    T1.display();
    return 0;
}
```

26/09/2023

Diamond Problem

↳ It arise the problem of ambiguity

↳ Solution for this funtion is virtual funtion.




```
class parent {
~~product~~ protected:
        int basedata;
};
class child1 : virtual public parent {};   // shares copy of parent
class child2 : virtual public parent {};
class Grandchild: public child1, public child2
{
  public:
    int getdat() {
    return basedata; // only copy of present
    }
};
```

private ⇒ only within same class

protected ⇒ can able to accessed by derieved classes.

public is private Inheritance

```
Class A {
    private:
        int x;
    protected:
        int y;
    public:
        int z;
};
Class B: public A {
    public:
        void fn() {
            int a;
            a = x; // Error
            a = y;  } ok
            b = z;  }
};
Class C: private A // private derived
{ public:
        void fn() {
            int a;
            a = x; // Error
            a = y;  } ok
            a = z;  }
};
int main() {
    int a;
    B obj B;
    a = obj B.x;  } Error not accessible
    a = obj B.y;  }
    a = obj B.z; // ok
    C = obj C;
```

// a = obj.x;
a = obj.y; } Error
a = obj.z;
}

Friend function :

↳ Fn. can access private / protected & public members of a class.

Syntax :-

class class name {
friend returntype function name (arguments);
};

class beta;

class alpha {
private:
int data;
public:
alpha()
{
data = 3;
}
friend int frifunc (alpha, beta);
};

class beta {
private:
int data;
public:
beta() { data = 7; }

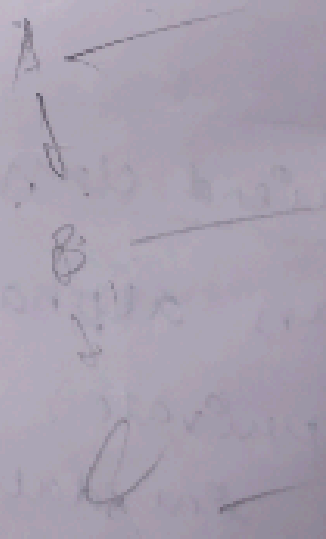
```
friend int frifun(alpha, beta);
};
int frifun(alpha a, beta b)
{ return( a.data + b.data);
}
int main() {
  alpha aa;
  beta bb;
cout << fn func (aa, bb);
  return 0;
}
```

Friend class

```
class alpha {
  private:
    int data1;
  public:
    alpha: data1(99);
  {};
  friend class beta;
};
```

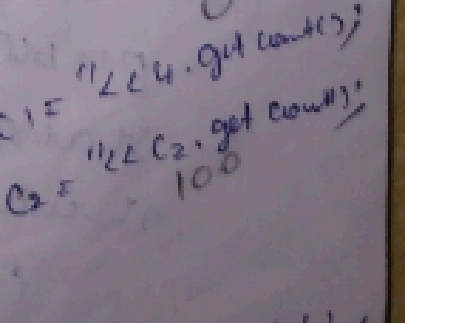
```
class B {
int b = 10;
friend A;
};
B b;
b.a;
```

```
class beta {
public: // allen private alpha data
void func1(alpha a)
{ cout << "Data1 = " << a.data;
}
void func2(alpha a)
   {
   cout << "data = " << a.data;
    }
};
int main() {
   alpha a;
   beta b;
   b.func1(a);
   b.func2(a);
  cout << end;
   return 0;
}
```



29/9/23 Derived class constructor

```
class counter {
protected:
    int count;
public:
    counter()
    {
        count = 0;
    }
    counter(int c)
    {
        count = c;
    }
    int getcount()
    {
        return count;
    }
    counter operator++()
    {
        return (++count);
    }
};
class countdown: public counter
{
public:
    countdown(): counter()
    {
    }
    countdown(int c): counter(c)
    {
    }
    countdown operator--()
    {
        return (--count);
    }
};
int main()
{
    countdown c1;
    countdown c2(100);
    ++c1;
    cout << "After increment" << c1.getcount();
    --c2;
```



```
cout << " After decrement " << c2.getcount() <<
    getcount();
}
```

//Function Overridding//

```
class parent {
  public:
virtual void print()
  { cout << "Base fn";
  }
};
class child: public parent
{ public: void print()
  { cout << " Derived fn";
  }
};
int main() {
  child c;
  c.print(); // Derived function.
  }
```

Static Member in a class

```
class student {
  public:
  static int total;
  student() {
      total += 1;
  }
};
```

int student :: total=0;

```
int main()
{
  student s1;
  cout << "Number of stds " << s1.total;
  student s2;
  cout << "Number of std " << s2 total;
  return 0;
}
```

---

static functions

```
class gamma {
  private:
    static int total;
public:
    gamma()
    { total++;
      id: total;
    }
static void showtotal()
{ cout<<"Total is" << total;
}
void showid()
{ cout << "ID no\n " << id;
}
};
int gamma :: total : 0;
```

```
int main()
{
    gamma g1;
    gamma::showtotal();
    gamma g2,g2;
    gamma::showtotal();
    g1.showid();
    g2.showid();
    g3.showid();
}
```

Vector

- It is a dynamic array size
→ allows us to resize the ~~change~~ of the array during run time
→ Worked basis of template class
↳ It can store elements of any data type

```
vector<int> a;
vector<string> a;
```

→ you can able to access any elements by their index
↳ like arrays

Hint

You must use <VECTOR> Header file

```
#include <iostream>
# Include <vector>
Using namespace std;
int main() {
   vector <int> a;
   a.push_back(10);
   a.push_back(20);
   a.push_back(30);
```

Insertion @ back

position & Index both different

@ Specific position from Front.

```
a.Insert(a.begin()+1, (value) 40);
```

position

// normal array begin(a) end(a)

// Vector a.begin() a.end()

Don't confuse.

Merge two vectors

```
vector <int> b = { 50, 60, 70 }
```

```
a.insert(a.begin()+1, b.begin(), b.end());
```

pos of 1st vector

@ Specific position from Back

```
a.insert(a.end()-1, 100);
```

position before the last element from last

// Deleting elements from the Vector.

a.popback(); delete element from the last.

@ Specific position. (pop_back())

a.erase(a.begin() + 2);
a.erase(a.end() - 2);

Displaying the vector

```
for(int i : a){
    cout << i << end;
}
```

This method is used for array, string, Vector, everything (those thing you don't know the size of it)

a.size(); // size of the vector

// string a.length();
a.size();

// array
int size = size of(a) / size of(a[0]);

auto (X) It is the keyword which automatically assigns the datatype of the variable according the value you panit.

To find the particula datatype

#include <typeinfo>

~~cout << type id(a)~~

```
typeid(a).name();
```

return 0;}

---

for Matrix:

```
Vector<Vector<int>> a = {{1,2,3},{4,5,6},
                         {7,8,9}};

for(auto &i : a){
  for(auto &j : i){
    cout << j << " ";
  }
  cout << endl;
}
row = a.size();
column = a[0].size();
```

Normal matrix

```
row = sizeof(a) / sizeof(a[0]);
column = sizeof(a[0]) / sizeof(a[0][0]);
```

5/9/23

Const data Member

```
class Number {
private:
  const int x;
public:
  Number() : x(36) { }
void display() {
  cout << "x : " << x << endl;
}; }
int main() {
  Number Num;
  Num.display(); // op: x=36
}
```

Const member function

used for display function

```
class Demo {
  int x;
public:
void set_data(int a) {
  x = a;
}
int get_data() const {
  ++x;
  return x;
}
int main() {
  Demo d;
d.set_data(10);
cout << d.get_data();
return 0;
}
```

used to avoid accidental changes in data

++x; → Throw error

const object

```
class Dog {
public:
    string name1;
    string breed1;
Dog (string name, string breed ) {
    name1 = name;
    breed1 = breed;
}
};
int main () {
const Dog dog ("puppy", "Breed_1");
    cout << dog.name1 << " is of " << dog.breed1;
    dog.breed = "breed 2";
    cout << dog.name << " is of " << dog breed1
}
```

Pointer

Void pointer :-

↳ pointer that has no associated datatype with it.

A void pointer can hold add of any type of & can be typecasted to any type.

```
int main() {
    int a = 10;
    char b = 'x';
    void *p;
    p = &a;
    p = &b;
}
```

---

1) void pointer * (cannot) be dereferenced

```
cout << *p; // Error
```

for dereferencing :

```
cout << *(int *)p;
```

2) pointer Arithmetic using void pointer is allowed.

```
#include <iostream>
#include vector
using namespace std;
int main () {
vector <int> a = { 1, 2, 3, 4, 5 };
vector <int> * p = &a;
for ( auto & i : *p) {
for ( auto & p : *p) {
cout << i << " ";
}
return 0;
```

printing the array elements.

---

pointer to object

```
class Distance {
  private:
    int feet;
    float inches;
  public:
    void getdist () {
      cout << "Enter feet : ";
      cin >> feet;
    cout << "Enter inches : ";
    cin >> inches;
} };
```

```
int main() {
    Distance dist;
    dist.getdist();
    dist.showdist();
    Distance *distptr;
    distptr = new Distance;
    distptr->get_dist();
    distptr->show_dist();
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int n = 5;
    for (int i = 0; i < n; i++) {
        cout << &a[i] << endl;
    }
    int n = sizeof(a) / sizeof(a[0]);
}
```

a.length()

a =

6/9/23

'This'

↳ It is a keyword which holds the current instance of the class.

```
class A {
  public:
  int x;
  A (int x)
  { this -> x = x; }
  void print () {
    cout << "The value of x is " << this -> x;
  }
};
int main () {
  A obj1 (4);
  obj1. print ();
  A obj2 (100);
  obj2. print ();
  return 0;
}
```

parameter & data member had same name to avoid confusion we use (this ->)

"This" ↳ It is a keyword

↳ It represents a pointer to the object on which the memory for it being called.

→ used to distinguish class member & fun parameter:

hiding

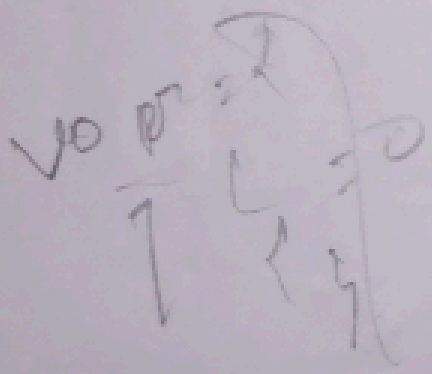
```
class Base {
public:
virtual void show()
  { cout << "Base class";
};
class Derived : public Base {
public:
  void show() {
  cout << "Derived class";
  }
};
int main() {
  Derived obj1;
  obj1.show();  → calls derived class show()

  Base *ptr = &obj1;
  ptr -> show();
                  calls Base class show()
                  hides the derived class show()
}
```

override

Diff b/w hiding & overriding

Operator overloading (+, -, *, /)

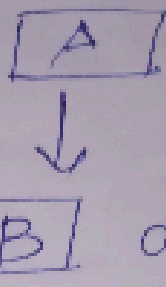
- can able to customise these operators for functioning various operation b/w the objects.



Inheritence (4)

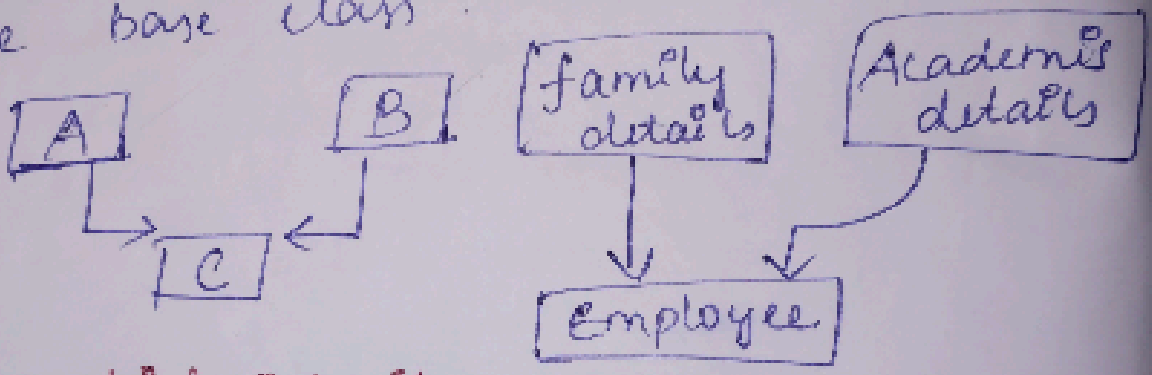
- Multiple
- Hierarchical
- Multilevel
- Hybrid

Simple Inheritence

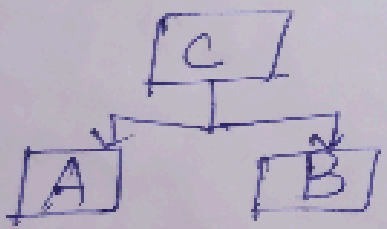


Multiple

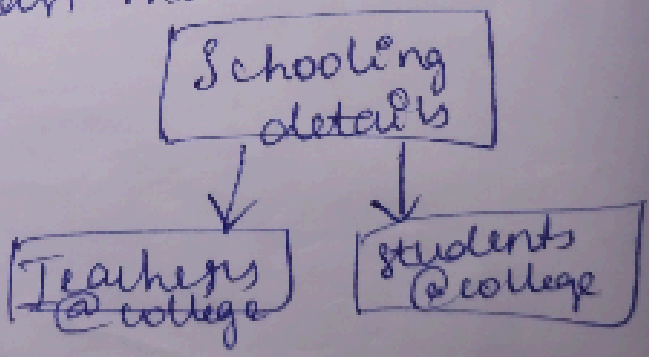
↳ derived class had more than one base class.



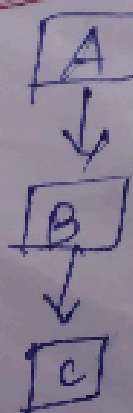
Hierarchial Inheritence



⇒ Single Base class more derived class.



Multilevel



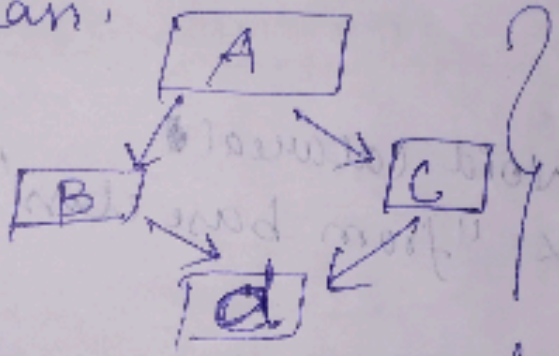
derived class from already derived class

Hybrid Inheritence / Diamond problem.

=) Hybrid inheritence is a combination of multiple inheritence & multilevel inheritence.

=) A class is derived from two classes as in multiple inheritence.

=) However one of the parent classes is not a base class.



for that we use virtual function.

```
class A {
protect: int x = 10;
};
class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {
public:
void d() {
cout << x;
}
};
int main() {
D d;
}
```

Virtual function

⇒ It is declared in the base class with keyword VIRTUAL & it is intended to be overriden by derived classes.

⇒ primary use of virtual func is to achieve polymorphism
↳ fundamental concepts of OOP's.

```
class A{
  public:
  virtual void calarea(){
    cout<<"from base class";
};};
class B: public A{
  public:
  void calarea() override{
    cout<<"from class B";
};};
class C: public A{
  public:
  void calarea() override{
    cout<<"from class c";
};};
class D: public B, public C{
  public:
  void calarea() override{
    cout<<"from class d";
};};
```

```
int main(){
  B b;
  b.calarea();

  D dd;
  dd.calarea();

  A a;
  a.calarea();
}
```

9/10/23

Template function.

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

Both codes are same to avoid the repition of codes we use template function.

```
double add(double a, double b)
{
    return a+b;
}
```

A template fn in C++ is a fn that is definite with one or more template parameters allowing it to work multiple datatypes.

→ code reusability
→ Avoid code duplication.

Syntax

```
template < typename (or) class  Type1 >

return_type fnname (Type1 x)
                    └parameter
{
  //
}
```

key words

User defined

```
template <typename T>
T add (T num1 , T num2)
 { return num1+num2;
}
int main() {
  int a=6, b=3;
  double x = 2.5 , y=4.7;
  int num1 = add (a,b);
  double num2 = add (x,y);
  cout <<" num of integers : "<< num1 ;
  cout <<" num of double : "<< num2 ;
  return 0;
}
```

parameters of different data types

```
template <class T1, class T2 >
T1 add (T1 a, T2 b)
 { return a+b;
}
int main() {
  int a=5;
  double b=7.5;
  cout << add (5, 7.5);
  return 0;
}
```

type int

for return double

```
double aa = add (5,7.5);
cout << aa;
```

Class Template

↳ Create a family of classes that share a common structure but can work in different data type.

```
template <class T>          // we can use typename also.
class Numbers {
    T first;
    T second;
public:
    Numbers (T a, T b){
        first = a;
        second = b;
    }
    T large ()
    { if ( first > second)
        { return first;
        }
      else
        return second;
    }
};
int main () {
    Number <Int> num(5,6);      // parameter   // Not a return type.
    cout << num.large();
    return 0;
}
```

Int a = add(5.5, 6.6)

Template Specialisations

```
template <class T>
class value {
    public:
        value (T a) {
        cout << a << "a number";
    };
};
template <>
class value <char> {
    public:
        value (char a)
        {
        cout << a << "is a char";
        }
};
int main () {
    value <int> v1(5);
    value <double> v2(5.5);
    value <char> v3('A');
    return 0;
}
```

18/10/23 C++ Virtual fns.

```
class Base {                    // Abstract class
    public:
        virtual void print () {
        cout << "base fn";
    }
};
```

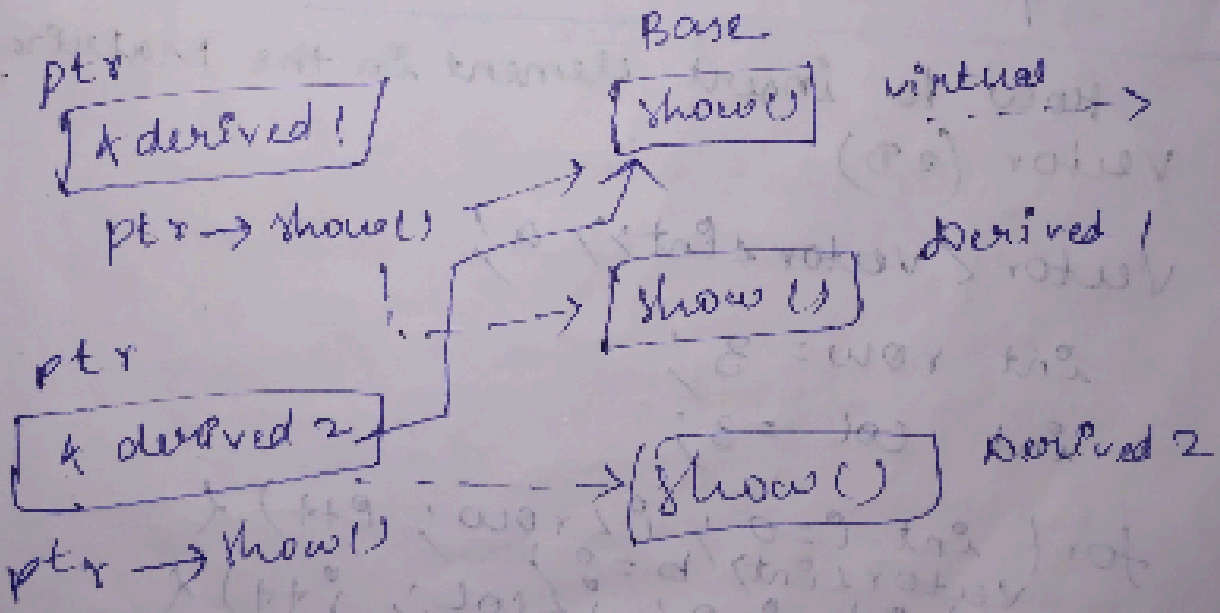
If virtual is not used here the O/P will be base fn.

```
class Derived : public Base {
public:
void print() {
    cout << "Derived fn";
};
int main()
{ Derived d1;
  Base * base1 = &d1;
  base1 -> print();
    return 0;
}
```

Base * base = new Derived()

o/p Derived fn

Virtual & non-virtual pointer Assign.



Vector Continuation

How to insert element in the vector. 1D

```
Vector<Int> a;
int n= 10
for (int i=0; i<10; i++){
  int x=0;
  cout<<"Enter the element : "<<i+1;
  cin>>x;
  a.push_back(x);
}
for(auto &i : a){
   cout<<i<<endl;
}
```

How to insert element in the matrix/ Vector (2D)

```
Vector<Vector<int>> a;
  int row= 3;
  int col =3;
for( int i=0; i<row; i++){
    Vector<int> b=0;
  for(int j=0; j<col; j++){
    int x=0;
    cout<<"Enter the element
          a["<<i+1<<"]["<<j+1<<
                "]: ";
    cin>> x;
    b.push_back(x);
}
```

```
    a.push_back(b);
}
for(auto &p : a){
    for(auto &p : p){
        cout<<p<<" ";
    }cout<<endl;
}
```

After CIA-II

fstream Library

3 classes

ofstream → Creating & writing information into files

ifstream → Reading information from files

fstream → used for Creating, Reading & writing files.

Create & Write to files

```
#include <fstream>
int main() {
ofstream MyFile("filename.txt");
MyFile << "Helloworld";
MyFile close();
}
string mytext;
ifstream MyReadFile("filename.tat");
while( getline (MyReadFile, myText) {
cout << myText;
}
MyReadFile close();
```

Exception Handling

```
#include<fstream>
#include<stdexcept>
int main()
{
    string filename;
    ifstream inputfile;
    cout << "Enter the name of the file";
    cin >> filename;

try {
    inputfile.open(filename);
    if(!inputfile)
        throw("Err");
    }
    string line;
    while(getline(inputfile,line))
        { cout << line << endl;
        }
inputfile.close();
    }
    catch(...)
    { cout << "An Unknown Error";
      return 1;
    }
    return 0;
}
```

30/10/23

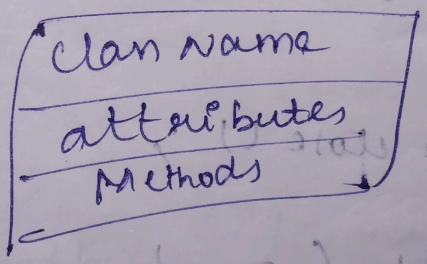
UML

↳ Unified Modeling Language

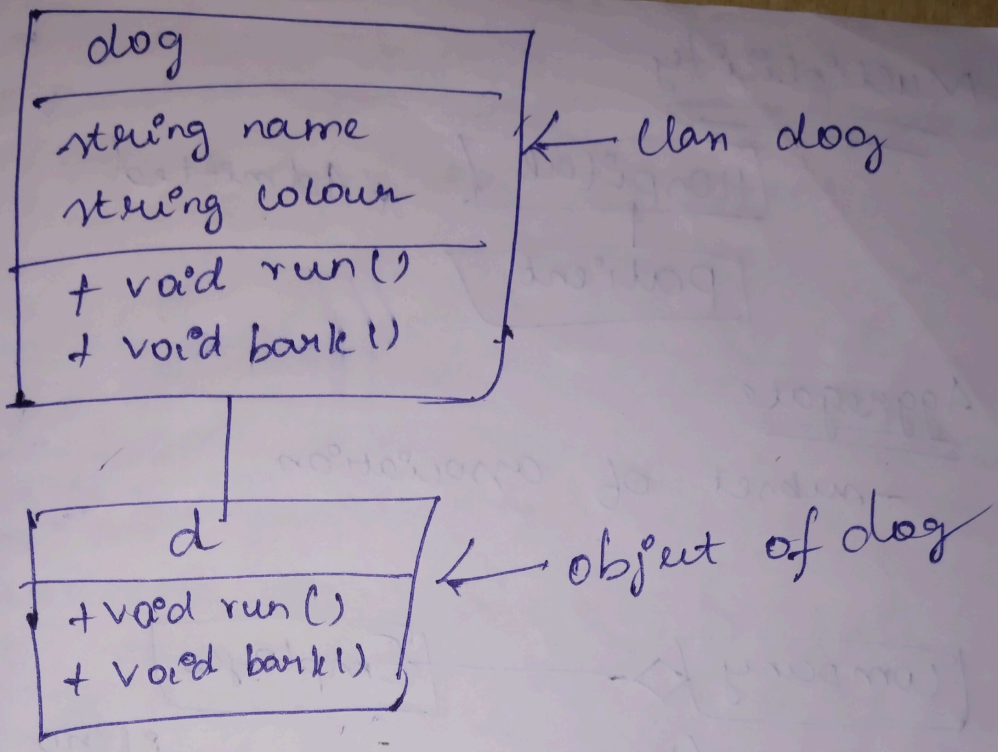
- used to visualise the system.
- It is a graphical Langua

UML Diagram

- Structural
  - i) class
  - ii) package
  - iii) object
  - iv) component
  - v) composite
  - vi) Deployment
- Behavioral
  - Activity Sequence
  - Use case
  - State
  - Interactive
  - Timing

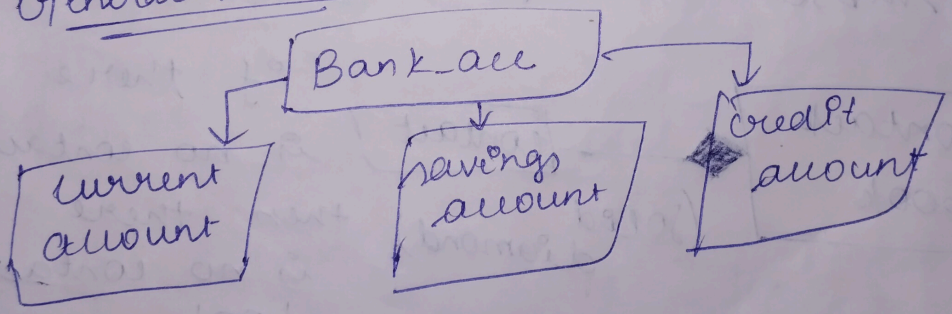


+ → Used for Public Visibility
# → Protected
- → Used for Private.



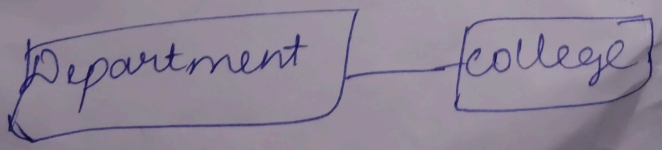
Relationship depending

Generalization



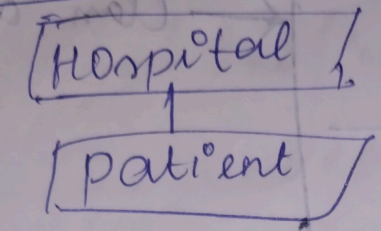
Association

- physical connection b/w objects



No arrows allowed (→)

2. Multiplicity



Aggregate

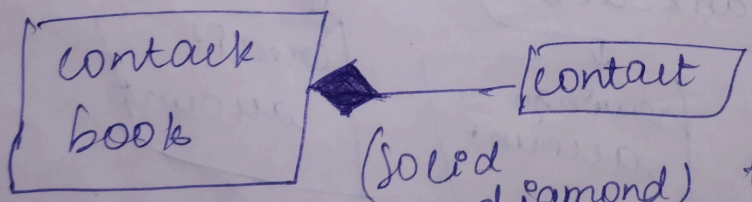
- subset of association

(hollow diamond)

If no employees present then also company run

Composition

- subset of aggregate

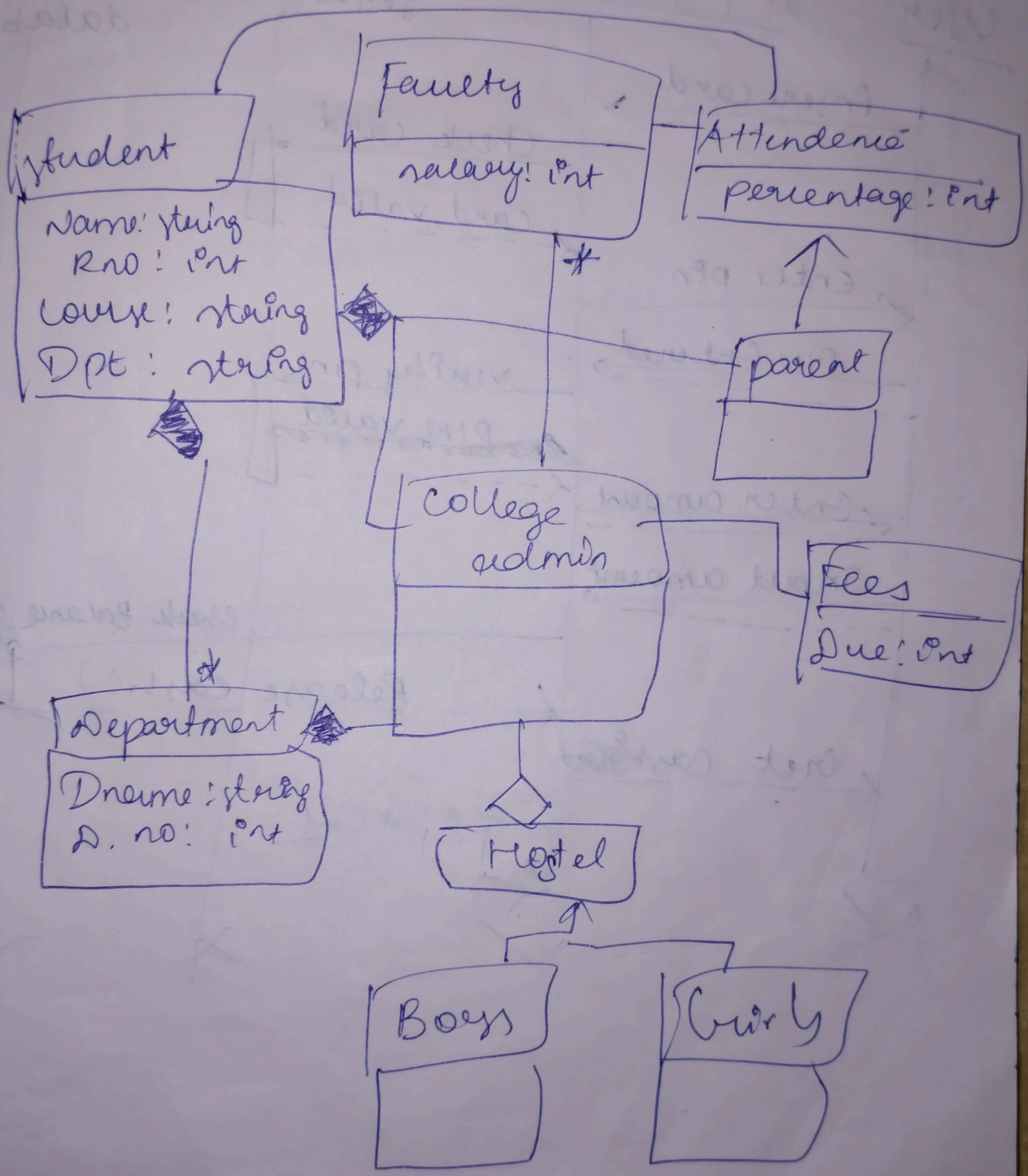


(solid diamond)

if there is no contact then there is no contact book.

College website class diagram:

External → in future it can be extended.
Include → includes in present



Sequence diagram