Call by value
Call by reference

* Assignment-

1) int n1, n2, sum, sub, mul;
void calculate (n1, n2, ref of sum, sub & mul)
using references make changes into sum, sub & mul variables.
Print results into main fun^n.

2) int calculate (int, int, int choice)
choice = 1 return sum
choice = 2 return sub
Choice = 3 return mul.

3) use fun^n pointer to put appropriate address of fun^n, based on choice, call fun^n using fun^n pointer.

int cal (int, int, int &, int &, int &)

int cal (a, b, int sum, int sub, int mul)
{
sum = a+b;
sub = a-b;
sub = a*b;

int cal (int, int, int)
main
{
a
b
cal (a, b, c)

OOPs

* Class → A blueprint from which object is created.
↳ User defined datatype.
↳ collection of data members & fun^n.
↳ collection of different types of elements
↳ classification of an entity.

* Object → Physical existance of an entity.
↳ variable of type class.
↳ size of object = All data members present in class
↳ which has state, behaviour, unique identity.
↳ Gets a birth ⇒ Lives life ⇒ Dies
↳ state will change but behaviour will remain same.

* Abstraction → When we declare a class take only a relevant data & Ignore which is irrelevant data.
↳ wherever we call a fun^n by object the implements details are hidden & this is fun^n level abstraction.

* Encapsulation → When an object's state changed inside a fun^n then this is encapsulation.
object's state changed because a method is called on that object.
↳ Encapsulation is achived by keeping data mem private & member fun^n public.

* Polymorphism → Pointer of general class pointing towards
Mechanism to call ~~same variable name~~ fun^n using same variable name.

* Inheritance → Creating a new class by using existing class

e.g class

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

/* class student {
     int rno;
     string name;
     float m1, m2, m3;
     char grd;
   }; */

class date {          //developer user
   int dd, mm, yy;
   Private: // accessed from within the class
   int dd, mm, yy;
   public: // accessed from outside class
     void display_date()
     {
       cout << " in" << dd << "/" << mm << " / " << yy;
     }

};

Void read_date()
{
  cout << "Enter date(dd mm yy)" ;
  cin >> dd;
  cin >> mm;
  cin >> yy;
}

int main()          // user of class (programmer user)
{
   date obj;
   obj.read_date();
   obj.display_date();
   return 0;
}
```

* Class must have
  - ↳ setter & getter
  - ↳ constructor
  - ↳ destructor
  - ↳ Fun" like add_day

(accessors) getter → get the values of data members

(modifiers) setter → set the values of data members → obj.setDate()

Setter

```
Void setDate (int d)
{
   dd = d;     // call by value
}
void setMonth (int d)
{
   mm = d;
}
Void setYear (int d)
{
   yy = d;
}
```

if (d <= 31) dd = d;

if (d <= 12) mm = d;

```
obj2.setDate (5);
obj2.setMonth (10);
obj2.setYear (2023);
obj2.display_date();
```

getter

```
int getDate()
{
    return dd;
}
```

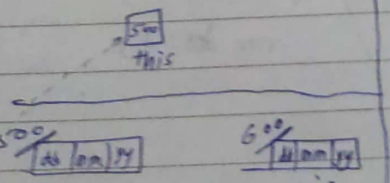
```
int getMonth()
{
    return mm;
}
```

```
int getYear()
{
    return yy;
}
```

```
if (obj.getDate() > 10)
    cout << "\n ok";
cout << "Day = " << obj.getDate();
```

* this pointer - [Encapsulation is possible because of 'this pointer']

↳ Is implicitly passed in every non-static fun^n of class.
↳ It contains address of calling object.



↳ Every member Fun^n of class has hidden parameter: this pointer.
↳ holds the address of the object invoking the member.
↳ available only in member fun^n of the class
↳ 'this pointer' is constant pointer.

obj.read_date(); => read_date(&obj)

compiler did this

```
Void read_date(date * this)
{
    ~~cin >> dd;~~ this
```

e.g. - Void setDate(int dd)

:: Scope resolution operator
↳ to access global variable

Scope of variable - visibility of variable

1) Local - declared in a fun^n, with in fun^n
2) global - entire prog.
3) class - with in class
4) Static - all static & non static fun^n
5) block - with block

Q. Scope of variable

↳

```
#include <iostream>
using namespace std;
class mycomplex {
    int ~~imaginary~~; real;   }//class scope
    int ~~complex~~; imag;
public:
    void setReal(int r) //local to ~~fun~~ fun^n
    {
        real = r;
    }
    void display()
    {
        cout << real << "+" << imag << "i\n";
    }
};
int main()
{
    mycomplex obj;
    obj.setReal(5);
    obj.setImag(7);
    obj.display();
}
```

Assignment -1

```
int n1,n2, sum, sub, mul;
Void calculate (n1,n2, sum, sub, mul)
Using references make changes into sum, sub, mul variables.
print results into main fun^n.
sum, sub, mul -> call by reference
n1,n2 -> call by value.
#include <iostream>
using namespace std;
int calculate (int, int, &int &, int &, int&);
int main ()
{
 int a,b,sum, sub, mul;
 cout << " Enter number ";
 cin >> a;
 cout << " Enter number";
 cin >>b;
 Calculate (a,b, sum, sub, mul);
 cout << " Result :" << sum << "\n" << sub << "\n" << mul;
 return 0;
}
int calculate ( int a1, int b1, int &sum, int &sub, int &mul)
{
  sum = a1+b1;
  sub = a1 - b1;
  mul = a1 * b1;
}
```

Assignment - 2

```
int calculate (int, int, int choice)
      choice = 1  return sum
      —"— 2  —"— sub
      —"— 3  —"— mul
#include <iostream>
using namespace std;
int calculate ( int, int, int);
int main()
{
 int a,b,c;
 cout << " Enter number";
 cin >> a;
 cout << " Enter number";
 cin >>b;
 cout << " Enter choice:" ;
 cout << " 1) sum 2) sub 3) mul ";
 cin >> c;
 ~~Calculate (a,b,c);~~ int x = Calculate (a,b,c);
 ~~return 0;~~ ~~int~~ x cout << " Result = " << x;
  return 0;
}
int Calculate ( int a, int b, int c)
 {
   switch (c)
     { case 1 : return a+b;
       break;
     case 2: return a-b;
       break;
     case 3: return a*b;
       break;
     default : cout << " Invalid choic";
```

Assignment 3

Use fun^n pointer to put appropriate address of fun^n based on choice, call fun^n using fun^n pointer.

```
#include <iostream>
using namespace std;
int calculation (int ,int ,int);
int (* fun) (int ,int);
int sum (int ,int);
int sub (int ,int);
int mul (int ,int);
int main()
{
 int a,b,c;
 cout << "Enter number";
 cin >> a;
 cout << "Enter number";
 cin >> b;
 cout << \n 1) add 2) sub 3) mul Enter choice: ";
 cin >> c;
 int x = calculation (a,b,c);
 cout << " \n Result : " << x;
 return 0;
}
int calculation (int a, int b, int c)
{
   switch (c)
    {
     case 1 : fun = sum;
      break;
     case 2: fun = sub;
      break;
     case 3: fun = mul;
      default : cout << " Invalid choice";
    }
    return fun (a,b);
}
int sum (int a, int b)
{
  return a+b;
}

int sub (int a, int b)
{
  return a-b;
}

int mul (int a, int b)
{
  return a*b;
}
```

Name mangling means ... given different names at the time of compile time binding of fun" is done.

Date 5/10/23

* Function overloading - compile time binding

we can write one fun" manytimes but the fun" signature for each fun" should be different.

we can use the same fun" name to define more than one fun" with different signatures.

Signature of a fun" means type & number of parameters.

e.g:
```
int sum (int, int);
float sum (float, float);
Float sum (float, int); ( int, float);
int sum (int, int, int);
Float sum (int, float, float);
float sum (float, int);
float sum (int, int); // Not allowed because (fun 1) matches this.
```

[Return data type is not consider in signature of fun".]

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

int main()
{
cout << "\n #s" << int sum(10,12);
         float sum(10.0f, 12.0f);
         float sum( 10, 12.0f);
```

① fun" overloading > Compile time polymorphism
↳ name decoration } Done by compiler at the time of compilation.
↳ name mangling

classmate Date Page

```
int sum (int a, int b)
{
  return a+b;
}
float sum (float a, float b)
{
  return a+b;
}
Float sum (int a, float b)
{
  return a+b; / return (float) a + b; / return float(a+b);
}
int sum ( int a, int b, int c)
{
return a+b+c ;
}
float sum ( int a, float b, float c)
{
 return a+b+c; / return a+(float)b +(float)c ; / return float(a+b+c);
}
float sum ( float a, int b)
{
 return a+b; / return float(a+b);
}
```

> objectdump -d filename.exe

It shows assembly language of a programm

mycomplex
{
}

You can read the value but can't change it.

* operator overloading -

Giving meaning to operators in user define class, so that the objects of that class should support expressions with operators.

e.g:-
```
#include <iostream>
using namespace std;
class complex {
    int real;
    int imag;
public:
    void setReal (int r)
    {
      real = r;
    }
    void setImag (int r)
    {
      imag = r;
    }
    void display()
    {
      cout << real << "+" << imag << "i\n";
    }
```

↳ operator overloading is a feature of C++ because of which additional meanings can be given to existing operators for user-defined datatypes.

~~↳ operator is |~~

- Following operators can NOT be overloaded ➔ :-
  : / . / :: / ?: / sizeof
- New & delete can be overloaded but they will be global

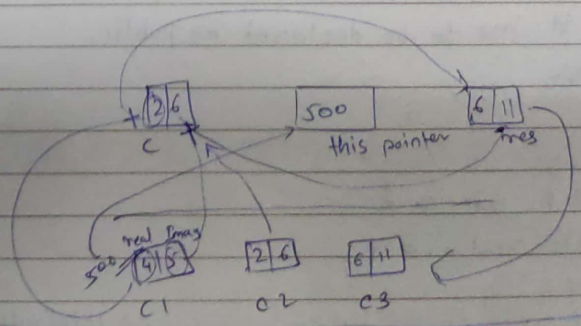
```
complex add ( complex c )   // complex datatype (user defined)
{
    complex res;
    res.real = real + c.real;
    res.imag = imag + c.imag;
    return res;
}
```

operator +

Call by value (c2 in main)

this → real, imag

```
int main()
{
    complex c1, c2, c3;
    c1.setReal (4);
    c1.setImag (5);
    c2.setReal (2);
    c2.setImag (6);
    c1.display();
    c2.display();

    c3 = c1.add (c2);   // c3 = c1 + c2;   // c1.operator+(c2
    return 0;
}
```



Fun^n overloading → In C++, two or more fun^n can share the same nam as long as their parameter declarations are d

- The fun^n that share the same name are sai to be overloaded, & this features is referred t as fun^n overloading.

constructor & destructor-

Constructor
- ↳ Member fun^n of class
- ↳ constructor fun^n name is same as class name
- ↳ called implicitly whenever the object is created. (need not to be called)
- ↳ We can overload constructor (constructor is nothing but the fun^n)
- ↳ We write instructions for memory allocation & initialization of member variables
- ↳↳ If constructor is not given in the class, compiler provides default constructor for each class.
- ~~eg:~~ ↳ If any ~~point~~ type of constructor is given in class, compiler will not provide default constructor
- ↳ constructor fun^n does not have any return data type, ~~eg:~~ not even void.
- ↳ It has to be declared as public.

~~class~~

e.g:-

```
class complex {
int real, imag;
public:
complex () { real = 0; imag = 0; } //default because no parameter
{
  cout << "\n Default Construct is called...";
  real = 0;
  imag = 0;
}
```

Compiler will give you copy constructor.

```
complex ( int r)
{
  cout << " in single parameter constructor called";
  real = r;
  imag = r;
}

complex ( int r, int i)
{
  cout << "\n 2 parameter called ----";
  real = r;
  Imag = i;
}

complex () { } // Default constructor created by Compiler.

int main()
{
  Complex C1(4), C2(4,4), C3 (5);
  C1.display();
  C2.display();
  C3 = C1.add(C2); / C3 = C1 + C2;

  return 0;
}
```

destructor

↳ Member fun° of class
↳ public
↳ name of fun° is ~nameofclass()
↳ takes no arguments
↳ No return datatype & not even void.
↳ one class only one destructor. (We can not overload destructor)
↳ will be called whenever the object goes out of scope. (will get called when object dies)
↳ will release resources aquired by object

e.g -

```
class{
  ~complex()
   {
    cout << "in Destructor is called...";
   }
};

int main()
{
  c1.display
  c2.display
  c3.display
  cout << "\n\n\n\n\n the end ";
  return 0;
}
```

Q. Add constructor, Destructor, operator overload > parameterized constructor in Date class. Write operat with int value

Q. ~~e.g.~~

```
class point{
      int x,y;
public:
   constructor ---

   destructor ---

   setter & getter for all member

   display()  (x, y)
};
int main(){

   create object of point & use functionalities

}
```

Q. class ~~stud~~ Student {

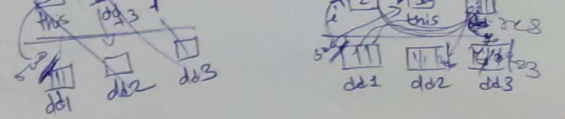
```
      int rollno;
//{   char name[20];
    char branch[30];
    int marks;
    char grade;

public:
    constructor -> default & parameterized (
    Destructor
    getter
    setter for all members
    display
};
```

page no 133

Assignment 1 Date #

```
#include <iostream>
using namespace std;
class mydate {
    int dd, mm, yy;
Public:
  mydate() {
     dd = 1;
     mm = 1;
     yy = 2000;
  }
mydate (int d) {
     dd = d;
     mm = 1;
     yy = 2000;
  }
mydate ( int d, int m , int y)
  {
    dd = d;
    mm = m;
    yy = y;
  }
void display ()
  {
    cout << dd <<"-/ "<< mm <<"/ "<<yy;
  }
mydate operator + ( int i)
  {
     mydate res;
     res.dd = dd + r;
     res.mm = mm;
     res.yy = yy;
```

dd2 dd3 dd3




```
int main()
{
  mydate dd1(3,10,2023); dd2, dd3(3);

// dd1 + 3;     // dd1.operator+(3)

dd1 = dd1 + 3;  // dd1.operator+(3)

//dd2 = dd1 + 4; // dd2.operator+(4)

   dd1.display();   -> 4/10/2023
   dd2.display();   -> 1/1/2000
   dd3.display();   -> 3/1/2000
return 0;
}
```

```
class {
mydate ~~void~~ operator++()
  {
     dd = dd+1;
     return *this;
  }

int main()
  {
~~dd2 = dd1++;~~      dd2 = ++dd1;
  }
```

(int d) -> dummy parameter ++dd1 or

used ~~only~~ only when "dd++"

Not for "++dd"

Q. Point class

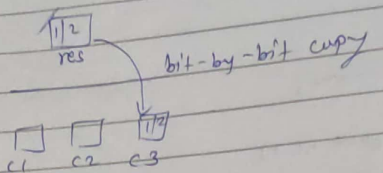
↳

```
#include <iostream>
using namespace std;
class point {
  int x;
  int y;
public:
  point()          // Default constructor
  {
    x = 1;
    y = 2;
  }
  point (int a)    // constructor with 1 para
  {
    x = a;
    y = 4;
  }
  point (int a, int b) // constructor with 2 para
  {
    x = a;
    y = b;
  }
  void setX(int a)
  {
    x = a;
  }
  int getX()
  {
    return x;
  }
  void setY(int b)
  {
    y = b;
  }
  int getY()
  {
    return y;
  }
  void display()
  {
    cout << "\n" << x: " << x << " y: " << y;
  }
  ~point()
  {
    cout << "\n Destructor";
  }
};
int main()
{
  point P, P1(3), P2(5,6);
  P.display();     // x: 1  y: 2
  P1.display();    // x: 3  y: 4
  P2.display();    // x: 5  y: 6
  P.setX(7);
  P.getX();
  P.getY(8);
  P.getY();
  P.display();     // x: 7  y: 8
  return 0;
}
```

Q. Student class

```
#include <iostream>
using namespace std;
class student {
    int rno;
    char name[30];
    char branch[30];
    int marks;
    char grade;
public:
    student()
    {
     rno = 0;
     strcpy(name, "NULL");
     strcpy(branch, "NULL");
     marks = 0;
     grade = 'F';
    }
    student(int r, char n[])
    {
     rno = r;
     strcpy(name, n);
    }
    void setrno(int r)
    {
     rno = r;
    }
    int getrno()
    {
     return rno;
    }
    void setname(char n[])
    {
     strcpy(name, n);
    }
    void getname()
    {
     cout << "\n" << name;
    }
    void setbranch(char b[])
    {
     strcpy(branch, b);
    }
    void getbranch()
    {
     cout << "\n" << branch;
    }
    void setmarks(int m)
    {
     marks = m;
    }
    int getmarks()
    {
     return marks;
    }
    void setgrade(char g)
    {
     grade = g;
    }
    void getgrade()
    {
     cout << "\n" << grade;
    }
    void display()
    {
     cout << "\n rno = " << rno;
     name << name;
     branch << branch;
     marks << marks;
     grade << grade;
    }
    ~student()
    {
     cout << "\n Destructor";
    }
};
int main()
{
 student s;
 s.display();
 char sname = "Rushi";
 char sbranch = "cs";
 s.setrno(2);
 s.setname(sname);
 s.setbranch(sbranch);
 s.setmarks(80);
 s.setgrade('A');
 s.display();
 return 0;
}
```

* shallow copy → bit-by-bit copy



Copying bit by bit from one object to another object.

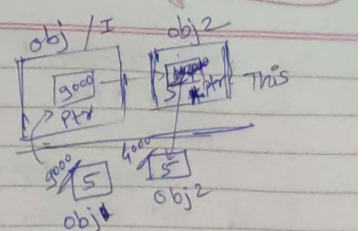
- Assignment operator also copies bit by bit.

Q. copy constructor

```
class Integer {
    int *ptr;

Public:
  Integer() { ptr = new int; *ptr = 0; }

  Integer(int n) { ptr = new int; *ptr = n; }

  ~Integer() { ptr = delete ptr; ptr = NULL; }
  Integer(Integer const &I)
  {  cout << " in copy constructor is called ";
     ptr = new int;
int main() *ptr = *I.ptr;
{     }
  Integer Obj1(5);
     void display()
     {
       cout << " \n value " << *ptr;
     }
```

6/10/23

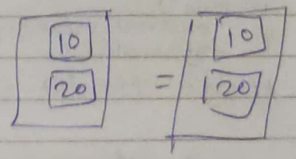
Assignment operator




```
int main()
{
  Integer obj(5);
  Integer obj2(obj);
  obj.display();
  obj2.display();
  return 0;
}
```

Assignment (=) operator - performs bit by bit (bitwise) copy of d from one obj to another.

```
void main()
{
  Complex c1, c2 (10, 20)
  c1 = c2;
  c1.Display();
}
```

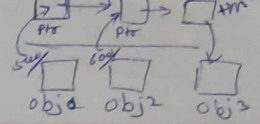


↳ Default definition is provided by compiler.
↳ OK for simple classes that do not have resource pointers.
↳ Fails for classes that have resources pointers.

overloading = operator

↳ Release the existing memory
↳ Get new memory block of appropriate siz
↳ Copy the value to new memory.
↳ must return *this. or * the defined variable.

```
class Integer {
    int *ptr;
public:
    Integer() { ptr = new int; *ptr = 0; }
    void set_value()
    {

    }
    Void get_value()
    {

    }
};
Const Integer & operator = (Integer const &I)
{
    delete ptr;
    ptr = new int;
    *ptr = *I.ptr;
    return *this;
}
int main()
{
    obj1.set_value(1,2);
    obj2 = obj1;
    // obj2.operator = (obj1);

    obj2.set_value(20);
}
Integer obj3;

obj3 = obj2 = obj;
obj.display();
obj2.display();
obj3.display();
```




```
class {

Integer operator + (Integer const &I)
{
    Integer res(*this->ptr *ptr);
    *res.ptr * * += *I.ptr;
    return *res;
}
```

return Integer(*ptr + *I.ptr);

This is called on fly object returned

```
int main()
{

    obj3 = obj2 + obj1;

    obj3.display();
}
```

→ Q. operator overloading on Integer class
+, -, /, *, %

int main()
obj2 += obj (-=/*=//=/%=)
⇒ obj2.operator += (obj1)

int main
{ obj3 = (obj2 += obj1);
}

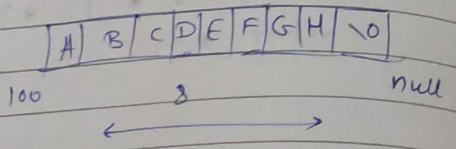
Q. >, <

```
class {
bool Integer obj2.operator >(obj1)
{ if(*ptr > *I.ptr) return true else return fal
```

Q. Class cstring (~~Array~~ String in class) Array

String class -

```
class cstring
{
  Char *p;
  int len;
Public:
  cstring();
  void display();
};
```



e.g:-

```
#include <string.h>
class cstring {
  char *str;
  int len;
public:
  cstring()
  {
    ~~str~~ len = 2;
    str = new char[2];
    //(*str = '\0';)
    strcpy(str, "\0");
  }
  cstring(int n)
  {
    len = n;
    str = new char[n];
  }
  cstring(char *P)
  {
    len = strlen(P);
    str = new char[len];
    strcpy(str, P);
  }
  ~cstring()
  {
    delete [] str;
    str = NULL;
  }
  cstring(cstring const & O)
  {
    len = O.len;
    str = new char[len];
    strcpy(str, O.str);
  }
  void display()
  { cout << "String is:"
    << str;
  }
  cstring &operator=(cons
    cstring &O)
  { delete[]str;
    len = ~~strlen~~O.len;
    str = new char[len
    strcpy(str, O.str);
    return *this;
  }
};
int main()
{
  char name[30];
  cout<<"\n Enter String";
  cin >>name;
  cstring obj1;
  cstring obj2(15);
  cstring obj3("Infoway");
  cstring obj4(obj3);
  obj2 = obj4;
  return 0;
}
```

What are the scenarios of memory leak?

MATRIX

Q.
```
class matrix {
    int row, col;
    int **mat;
public:
    matrix()
    {
      row = 2;
      col = 2;
      int i;
      mat = new int *[row]; // array of pointers
      for(i=0; i<row; i++)
        {
          mat[i] = new int [col];
        }
      for( i=0; i<row; i++)
        {
          for (int j =0; j<col; j++)
            {
              ma[i][j] = 0;
            }
        }
    }

matrix(int r, int c)
  {
    row = r;
    col = c;
```

```
~matrix()
{ int i;
  for ( i=0 ; i<row ; i++)
      delete [] mat[i];
  delete mat;
  mat = NULL;
}
void input_data()
{
  int i,j;
  for( i=0; i<row ; i++)
    {
      for (j=0 ; j<col ; j++)
        {
          cout << "\n Enter number";
          cin >> mat[i][j];
        }
    }
}
Void display()
{
  cout int i,j;
  for( i=0 ; i<row ; i++)
    {
      for (j=0 ; j<col ; j++)
        {
          cout << mat[i][j];
        }
      cout << "\n";
    }
}
```

copy constructor

```
matrix (matrix const &m)
{
    row = m.row;
    col = m.col;
    int i;
    mat = new int *[row];
    for ( i=0 ; i<row ; i++)
```

```
for (i=0; i<row; i++)
{
  for (j=0; j<col; j++)
  {
    mat[i][j] = m.mat[i][j];
  }
}
}
```

operator =

```
const matrix &operator=(const matrix &m)
{
  int i;
  for(i=0; i<row; i++)
  {
    delete[] mat[i];
  }
  delete mat;

  {code same as copy constructor}

  return *this;
}
```

```
int main()
{
  matrix obj;
  obj.input_data();
  obj.display;

  matrix obj2(obj);
  obj2.display();
  return 0;
}
```

Q. class Integer - constructor, destructor, copy cons, assign op, +, +=, ==

Q. class cstring - ~~~~~ occ of a char

Q. class matrix - ~~~~~ input_data

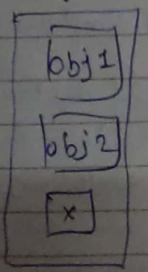
Q. class student -
- rno, char *name, int age

Inheritance -

containment

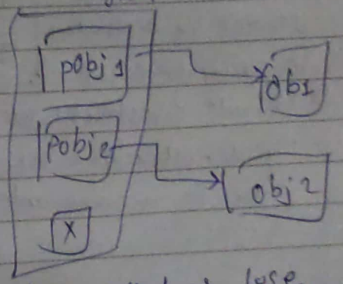
- ↳ represents "has a" relationship.
- ↳ Containment relationship means the use of an object of a class as a member of another class
- ↳ ex - Birth_Date or joining date as a part of Employ class.
- ↳ It brings reusability of code.
- ↳ Ex. Already written Date class can be used in class Employee.

Physical containment



Also called as tight coupling.

Logical containment



Also called as loose coupling.

e.g - Car-Documents

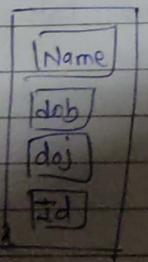
[oop says as far as passible objects loosly.]

'Has-A' Type of Relationship -

→ Facilitates code reuse
→ Already written classes can be used as members of another class.
→ Example :

```
class Employee
{
    CString Name;
    Date dob;
    Date doj;
    int Id;
public:
    Employee();
    Employee(char*, int, int, int, int, int, int, int);

    ~Employee();
    void Display();
};
```

↳ Contained objects created first
↳ Order of creation decided by declarations in container class
↳ Default constructor gets called
↳ Object destruction takes place in reverse order.



Sequence of constructor
1) CString() :: CString()
2) Date :: Date() for dob
3) Date :: Date() for doj
4) Employee :: Employee()

Sequence of destructor
1) Employee :: ~Employee()
2) Date :: ~Date() doj
3) Date :: ~Date() dob

Syntax : Member Initialization list

```
Employee :: Employee(char *p, int a, int b, int c,
                     int d, int e, int f, int g)
: Name(p), DOB(a,b,c), DOJ(d,e,f), Id(g)
{

}

Date :: Date(int a, int b, int c)
  : d(a), m(b), y(c).
{
}

CString :: CString(char *q)
{
    len = strlen(q);
    p = new char[len+1];
    strcpy(p,q);
}
```

→ Container object gets all the data from user & distribut it to appropriate contained object and to itself as wel
→ If not specified, all contained objects get created using default constructor. User input is reassigned later using member fun?.
→ Call appropriate constructor right at the time of creation. This improves performance.
→ Sequence of member initialization list can not override that of class declaration.
→ Even built in data members can be initialized in thi

Delegation-

Container delegates a responsibility to the contained objects by calling appropriate member fun°.

```
Void Employee :: Display()
{
Name.Display();
DoB.Display();
DoJ.Display();
}
```

* Inheritance

→ Extending the feature of a existing class into a new class.
→ Inheritance is where one class (child) inherits the members of another class (parent).
→ Subclass can have additional specific attributes & methods.
→ Establishes 'is-a' kind of relationship.
→ Moves down from generalization to specialization.
→ Most general at top to most specific at the bottom of inheritance chain.
→ The benefit of inheritance is that the child class doesn't have to redeclare & redefine all the members which it inherits from the parent class. It is therefore a way to re-use code.

~~Employee~~

Syntax:

```
Class DerivedClassName : access-level BaseClassName
```

where,
access level specifies the type of derivation
↳ Private by default.
↳ public
↳ Protected.

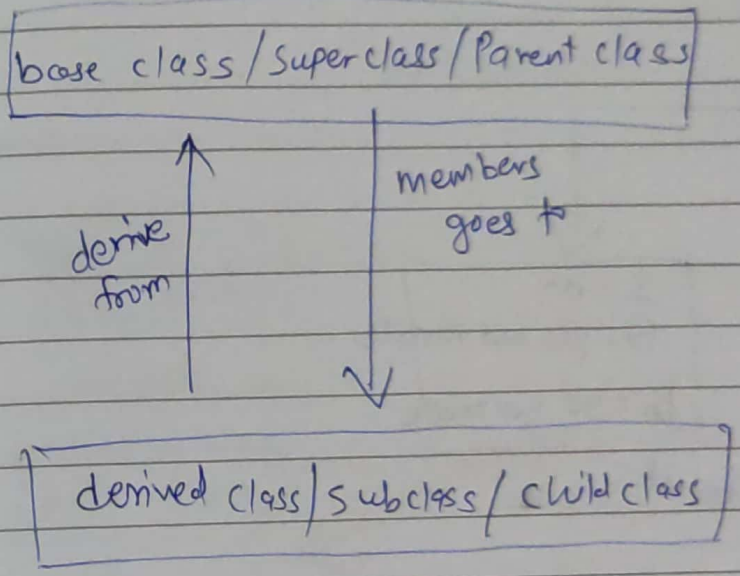
e.g-

```
class Base {
  int i;
  public :
  Base() { cout << "\n base default constructor"; i=0; }
  Base(int a) { cout << "\n base param constructor"; i=a;
  void display() { cout << "\n i=" << i; }
  int get_i() { return i; }

  ~Base() { cout << "\n base destructor"; }
};
class derived :
public Base {
    int j;
    public :
      derived() { cout << "\n derived default constructor
                  j=0; }
      derived(int b) { cout << "\n derived param cons
                  ctor"; j=b; }
      derived(int a, int b) : Base(a)
{
```

7/10/23

Code Reuse:-

→ Data as well as fun^n of base.

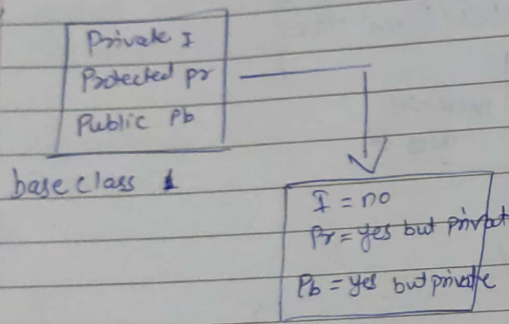


Access specifiers

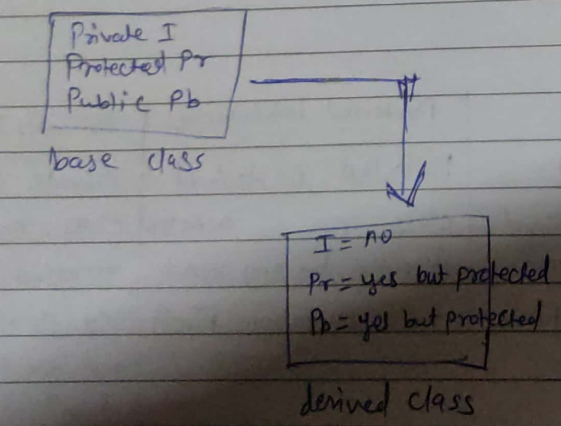
Base class member Access specifier	public inheritance	Protected inheritance	Private Inheritance
public	Public in derived class can be accessed directly by any non-static member fun^n, friend fun^n, & non-member fun^n.	Protected in derived class. Can be accessed directly by any non-static member fun^n, friend fun^n.	Private in derived class. Can be accessed directly by any non-static member fun^n, friend fun^n.
Protected	Protected in derived class can be accessed directly by any non-static	Protected in derived class. can be accessed directly by any non-static member fun^n, friend fun^n.	Private in derived class. Can be directly by any non-static mem fun^n, friend fu

Private	Hidden in derived class. Can be accessed directly by non-static member fun^n, friend fun^n through public or protected member	Hidden in derived class can be accessed directly by non-static member fun^n, friend fun^n through public or protected member	Hidden in derived class can be accessed direct by non-static member fun^n, friend fun^n through public or protected member

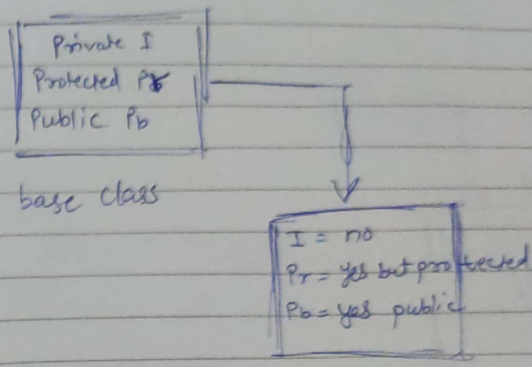
private



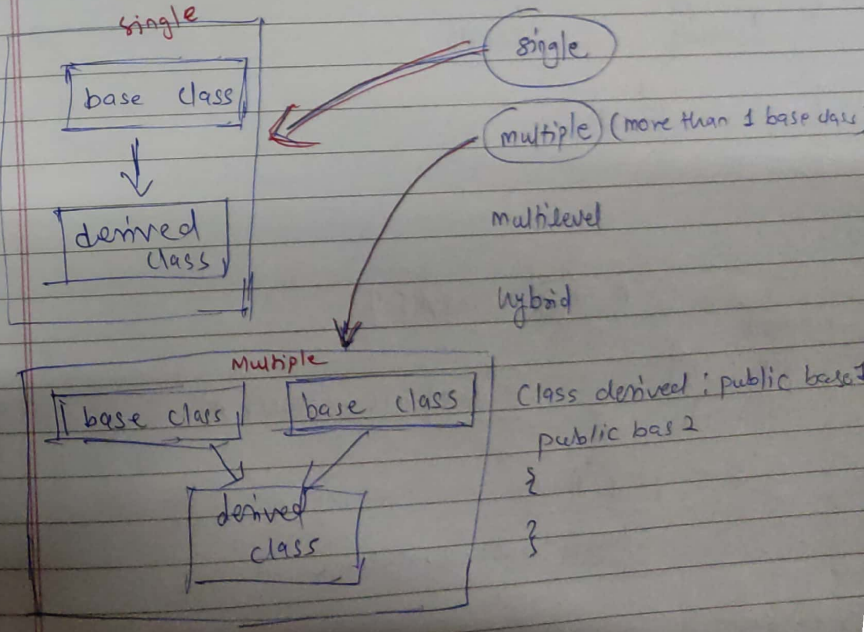
protected



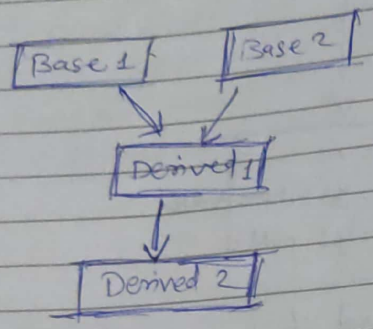
public



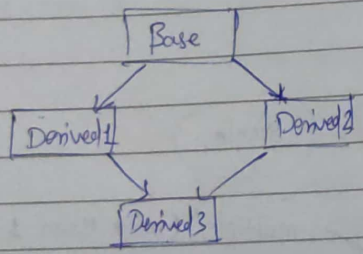
Types of Inheritance

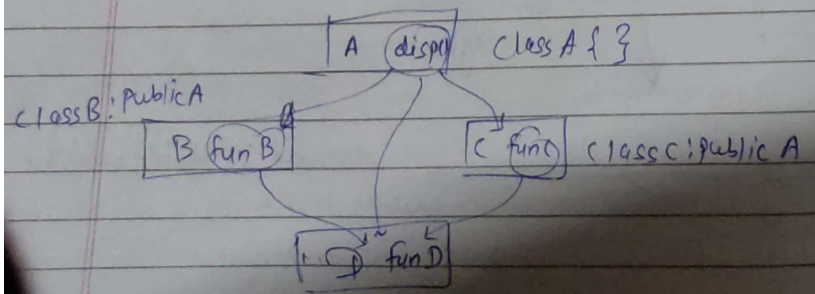


Multilevel



Hybrid





* Diamond problem.cpp

```
#include <iostream>
using namespace std;
class base {
public:
    void display()
    {
      cout << "\n" display";
    }
};
class A : public base {
public:
      void funA()
      {
        cout << "\n funA";
      }
};
class B : public base {
public:
      void funB()
      {
      cout << "\n funB";
      }
};
class C : public A , public B {
public:
      void funC()
      {
        cout << "\n funC";
      }
};
```

class C : public A, public B {

};

['Virtual' Keyword used in Dimand Problem.]

Try it with constructor & destructor too

~~Polymorphism~~ IS-A relationship

Polymorphism Q.

```
#include <iostream>
using namespace std;
class shape {
    int color;
  Public:
  virtual float Area()          // virtual float area()=0;
      {
        cout << " In Area of shape ";
        return 0.0;
      }
};
class circle : public shape {
  int rad;
  ~~circle~~ Public:
          circle ()
          {
            rad = 5;
          }
        circle (int i) { rad = i; }
};
```

This is pure virtual fun.

P1) Polymorphism is possible because base class pointer can hold Address of derived class object.

```
cout << "\n" << sh1 -> area();

sh1 = &s1;
cout << "\n" << sh1 -> area();
```

Polymorphism → one fun^n is taking multiple forms

→ run time binding

Java says each fun^n is virtual fun^n. so that each fun^n can give you run time binding. In C++ we need to write "virtual" keyword to use run time binding.

P4) If pure virtual fun^n is present in your class then it will become abstract class.

P5) [If class is abstract we cannot create object with that class.]

P2) [Make a fun^n virtual for run time binding]

P3) [If fun^n has no definition then make it pure virtual]

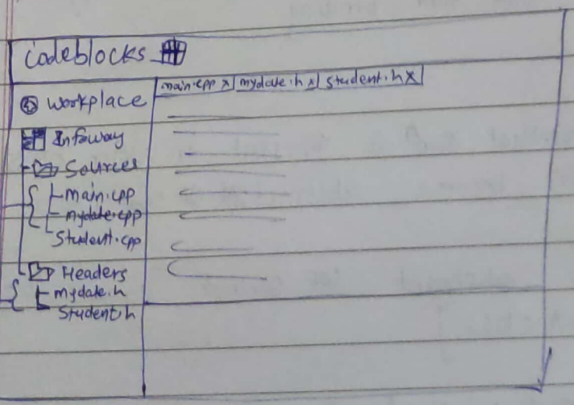
Create empty project

D:\

Title - Infoway

mydate.h (class)

```
#include <iostream>
using namespace std;
class mydate{
    int dd, mm, yy;
public:
```



→ write classes here with '.h' extension

→ write main code here with '.cpp' extension

To call headers in your main cpp file or another

Q. Assignment -

Define following class & add functionality

```
class A
 {
   int data;
 };
```

the above given is just a suggestion for class defini
Complete the class & fun^ definitions so that you ca
do following calculations on objects of that class
(write constructors & destructors)

Assume:

```
A a1, a2, a3 ;  //objects of classA
a1 = a2;
a1 = a2 * a3;
a1 = 10;
```

↳

```
#include <iostream>
using namespace std;
class A{
   int a;
public:
   A()          // destructor with default constructor
   {
    a = 5;
   }
   A(int x)     // parameterized constructor
   {
    *a = x;
   }
~A()
   {
```

```
A(A const &x)   // Copy constructor
{
    a = x.a;
}
const A
A operator = (A const A &x)   // assignment operator
{
    a = x.a;
    return *this;
}

A add operator * (A X)
{
    A res;
    res.a = a * ax.a;
    return res;
}

void seta(int x)
{
    a = x;
}
int geta()
{
    return a;
}
void display()
{
    cout << "Result " << a;
}
```

```
int main()
{
    A a1, a2, a3;
    a1.display();
    a1 = a2;   / [a1(a2); a2(a1);]
    a1.display();
    a1.seta(10);
    a1.geta();
    a1.display();
    a1 = a1 * a3;
    a1.display();
    return 0;
}
```

H.W Q. student s1(10,1,1,1998,"Pooja", 3,9,2023, 80.75f)
write ↑ constructor

main.cpp

```
#ifndef STUDENT_H
#define STUDENT_H
```

```
#endif
```