$T = RC$

100k ohm → 100 μF

10000 × 0.000

10000 × 0.00

10

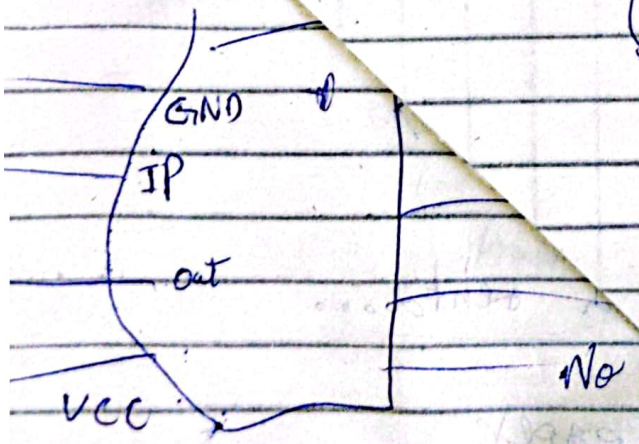
1×10^5

Notes stands for

rem → remember

IMP → important

OPERATOR OVERLOADING



Lecture Contents Checklist

(You should be Able to :)

- ☐ Overloading Binary Operators
- ☐ Overloade I/O "<<", ">>" operators
- ☐ Overload postfix/prefix increment op.
- ☐ Overload Array subscript operator
- ☐ Overloading assignment operator

Similar to function overloading

Operator Overloading

"Using function-overloading to overload operators / redefine the functionality of operators is ____"

* Even when we do:

```
int x=2;
int y=494;
cout << y-x << "\n";
```

The compiler translates it to, operator +(x,y)

the function call.

Bcz,

these are declared and defined as member functions inside the ____ class.

under the, library file.

* For us to add two objects of type Complex w/o using operator overloading, we would need to do that simple task by using a function like this:

```
class complex
{
    double real, imag;
public:
    double getReal() { return real; }
    double getImag() { return imag; }
};
complex add(const& complex c1, const& complex c2)
{
    complex temp;
    temp.real = c1.real + c2.real;
    temp.imag = c1.imag + c2.imag;

    return temp;
}
```

Three

TWO WAYS OF IMPLEMENTING OPERATOR OVERLOADING :

(1) overloaded operator function implemented as member function of class.

(2) overloaded operator function implemented as a non-member function.

↳ Note: In this case, the overloaded operator function must be declared as a friend function if it requires direct access to private or protected data members

(3) * otherwise, getter/setter functions would be used (if not a friend)

** In the case of overloading the binary '+' operator as a member function, when it would be called like in `c3 = c1 + c2;` then compiler would translate that call to

```
c3 = c1.operator +(c2);
```

** Whilst in the case of overloading the binary '+' operator as a non-member friend function, then when it would be called like

c3 = c1 + c2;

↳ would be translated as

↳ `c3 = operator +(c1, c2);`

* In this case, both objects need to be passed to that non member function.

(Defined as a member function)

e.g. code-1

```
class complex
{
    double real, imag;
    public:

    complex& operator + (const complex& passed)
    {
        double r = real + (passed.real);
        double i = imag + (passed.imag);
                    obj
        return (complex(r, i));
    }
    complex(double r, double i): real(r), imag(i) { }
};
int main()
{
    complex c1(2, 4);
    complex c2(99, -4);

    complex c3 = c1 + c2;
}
```

c3 gets:

real	imag
101	0

e.g. code 2 Defined as a non-member function and not a friend

```
class complex
{
    double real, imag;
    public:
        complex(double r, double i) : real(r), imag(i) { }
        double getReal() { return real }
        double getImaginary() { return imag }
};

complex& operator + (const complex& c1, const complex& c2)
{
    double r = (c1.getReal()) + (c2.getReal());
    double i = (c2.getImaginary()) + (c1.getImaginary());
    return ( complex(r, i));   // obj
};

int main()
{
    complex c1(34, -2);
    complex c2(64, 4);

    complex c3 = c1 + c2;

    // c3 becomes
}
```

real	imag
98	2

e.g code 3 — Defined as a friend function

//now class's object must be passed but no need of getter/setter functions due to friend function.

```
class complex
{
    double real, imag;
    public:
        complex(double r, double i): real(r), imag(i){}
    //providing declaration of friend function:
friend complex& (const complex& c1, const complex& c2)
{
    double r = c1.real + c2.real;
    double i = c1.imag + c2.imag;

    return ( complex obj(r, i));
}
```

Note: Being a friend function to a class only means that the function made as friend can directly access its private/protected data member given that object of that class is passed to that function.

i.e. the object of class in main() can not call it by obj.functionName(); just like the object of class could directly call its number function for operator overloading in e.g code-1.

Notes: Implementing operator overloading using non-member function and non-member FRIEND function is very same.

- except that in the case of non-member functions we would've to use

  Getters/setters to access private/protected data

  AND

  incase of non-member friend function, we would directly access the private data by using '.' operator with object name.

- TMP: Given that in both cases, both objects are passed into the functions

* However the previous 3 eg. codes do not cater such scenarios (scenarios A and B)

(A)
```
complex c1;
c1 = c1 + 2.764;
```

(B)
```
complex b9;
b9 = 6.781 + b9;
```

→ For Scenario-A

* we can create a member function - overloaded operator function. like: →

```
complex& operator+(double d)
{
    double r = real + d;
    double i = imag;

    return(complex obj(r,i));
}
```

→ For scenario-B

* AS THE L-OPERAND of the operator is of type double and not the user-defined complex datatype.
* AND we know that L-OPERAND calls the operator (incase if we declare the operator overloaded ~~member~~ function as a member function.)

So As L-OPERAND is not of Complex datatype, we cannot go for having a member function to implement this "b9 = 6.781 + b9;"

for our case
↓
friend

Scenario-B contd.

∴ we must go for a non-member function to implement this feature.

↳ See code e.g. 4 for this :

```
class complex
{
    double real, imag;
    public:
        complex(double r, double i) real(r), imag(i)
        { }
//      friend complex& operator + (double d,
                                    const complex& c2);
};

complex& operator + (double d, const complex& c2)
{
    double r = d + c2.real;
    double i = c2.imag;

    return (complex obj(r, i));
}
```

Summarizing previous stuff

```
class complex
{
    double real, imag;
public:
    complex(double r, double i): real(r), imag(i) { }
    complex& operator + (complex c2);
    complex & operator + (double d);
    friend complex& operator + (double d, const complex &c2);
};
```

~~friend complex& operator + (double~~

$c_1 + c_2$

c_1.operator +(c_2);

c_1 + double

c_1.operator +(d)

double + c1

operator +(d, c1)

Test this On Compiler

Chaining

```
complex c = c1 + c2 + c3;
```

↳ would automatically chain these multiple calls to the binary operator in sequence as:

```
complex c = (c1.operator +(c2)).operator +(c3)
```

even

```
complex c = c1 + 4.96 + c2;
```

would run fine as long as we have implemented

```
c1.operator +(c2)
```

AND

```
c1.operator +(double d)
```

Bcz, this would be converted into: (by compiler)

```
c = (c1.operator +(4.96)).operator +(c2);
```

AND this would return an object of type complex which could further act as L-operand for this

(See prev. pages (return type is complex))

Restrictions on Operator Overloading

* The overloaded operator must be an existing or valid operator.
You cannot create your own operator such as xD or :)

* Certain C++ operators can't be overloaded, such as
  - sizeof
  - dot operator "."
  - scope resolution operator "::"
  - Conditionals like "?", ":" (used in ternary operator)

** The overloaded operator must have atleast one operand of a user-defined data type.
(You can't overload operators working only on fundamental aka built-in data types)

e.g.
```
operator + (int a, int b)
```
won't work.

* Associativity, precedence & no. of arguments of an operator can't be changed.

e.g.
"+" would remain left associative
"=" would remain right associative

Overloading "<<" operator (Stream/line insertion)

Remember aka kem: This is a Binary Operator (takes two operands)

e.g.
```
complex c1;
cout << c1;
```

- cout → L.H.S is a C++ object of the ostream class which is in the iostream library
- << → insertion operator
- c1 → object of type complex

∴ To overload/redefine the "<<" operator, we must pass an object of ostream class (by reference) AND an object of the complex class (by reference) TO the operator overloaded non-member friend function.

Reasons:
L.OPERAND was not of type class complex, infact we want to print details of the object of type complex so it would always be like `cout << obj;`

∴ We couldn't go for member function to implement this.

(Test on compiler, what if cout is not replaced)

Side Note, cout is a c++ object of the ostream class which uses the "<<" operator along with it to print something.

So similarly, we can give our own object name rather than "cout" (as its a reserved keword) to implement overloading the "<<" operator.

e.g code-1

```
class complex
{ private:
        double real, imag;
  public:
        complex(double r, double i) : real(r), imag(i){}

        friend ostream& operator <<(ostream&, complex);
};

ostream& operator <<(ostream& out, complex c1)
{
  out << "\n\n\t Complex number is: "
      << c1.real << " + " << c1.imag << " i\n\n";
  return out;
}
```

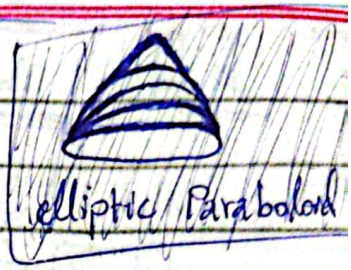
This is done to enable cascading

See this page for revision later V.I.

Shortcut to rem. overloading insertion & extraction operators

* Rem: cout & cin are ~~objects of~~
  - cout → object of the ostream class in the iostream library
  - cin → object of the istream class in the iostream lbr
* ~~AND for overloading~~
* AND that these are Binary operators
* So, we have to pass object of ostream / istream & then the object of class.
* AND for cascading, we return the object
  - therefore, returntype is set as either ostream& or istream&

e.g code 2 for overloading stream extraction operator ">>"




```
class complex
{
        double real, imag;
public:
        complex(double r, double i): real(r), imag(i){}
    friend istream& operator >>(istream& in, complex&c)
};

friend istream& operator >>(istream& in, complex&c)
{
    in >> c.real;
    in >> c.imag;

    return in;
}
int main()
{
    complex c(4.96728, 13);        and imaginary part
    cout<<"\nplease enter the real part: ";
    cin >> c;
}
```

* Now c, would have the user-entered values.

* Notes: Within main() we would use cin & cout, ("in" and "out" were just used to represent objects of istream & ostream classes.

Infact, in the previous code,

* within main()
the statement:

```
cin >> c;
```

basically is converted as: (By compiler)

```
operator >> (cin, c);
```

* we could not write "cin" to represent an object of istream and similarly "cout" to represent an objects of ostream.

could not write

↳ within the function parameters and the function body for the operator overloaded functions of ">>" and "<<"

Because, "cout" & "cin" are reserved keywords

Overloading Unary Operators

- ++
- --
- - (Negation operator)
- ! (Logical NOT operator)

Operating on one operand only

Overloading Logical NOT operator

① As a member function

```
class complex
{
    double real, imag;
public:
    complex(double r, double i) : real(r), imag(i){}

    bool operator !() const
    {
        return (real == 0 && imag == 0);
        // TRUE is returned only if real & imaginary
        // part of complex number are zero.
    }
int main() {
    complex c1(0,0);
    if (!c1) {
        cout << "\n\nIt Empty\n\n";
    }
}
```

In this case yehi AAYEGA

Overloading Logical NOT

② As a non-member friend function

e.g.

```
class complex
{
    double real, imag;
    public:
        complex(double r, double i): real(r), imag(i)
        { }

        friend bool operator !(complex c);
};
friend bool operator !(complex c)
{
    return (c.real == 0 && c.imag == 0);
    //returns TRUE only if condition fulfills.
}
int main()
{
    complex c(0, 0);

    if(!c)
    {
        cout << "Emptyyyyyyy !! ";
    }
}
```

postfix,

Means Dummy argument (int) for postfix

Overloading unary (prefix, postfix) operator

```
class complex
{
    double real, imag;

    complex(double r, double i) : real(r), imag(i) { }

    complex& operator ++(int)   //postfix
    {
        real++;
        imag++;
        return *this; // to enable cascaded
    }

    complex& operator --(int) //postfix
    {
        real--;
        imag--;
        return *this;
    }
    complex& operator ++() { // prefix
    {
        ++real;
        ++imag;
        return *this; }
    complex& operator --() { // prefix
    {   --real; --imag;
        return *this;
    }
};
```

```
int main()
{
    complex c(4,6);

    ++c;    // (5,7) now
    c--;    // (4,7)

    ++((++c)++); // (7,9) now
}
```

Cascading was enabled through

```
return *this;
```

((See professor Hank Stalica's video for reasons))

Overloading Array subscript [ ] operator

* If a class has an array inside of it as a data member

then overloading [ ] operator lets us access (valid) indexes of that array by using subscript operator with the object inside main()!

① for this we have to pass index as paramater to the operator overloaded function for "[ ]" and

② It must return a reference to the memory address occupied by that specific index of the array (which we've passed)

So that what's returned could be used as a L-OPERAND for the assignment operator like to enable

```
complex c;
c[4] = 496;
```

* Note: The overloaded "[ ]" operator should ensure valid index and deal with invalid index passed correctly to prevent undefined behaviour.

** #include <iostream>

e.g. #include <cstdlib>

```
class database
{
    int employeeId[5];
public:
    database()    // CSTR
    {
        for(int i=0; i<5; i++)
        {
            employeeId[i] = 0;
        }
    }

    int& operator [](int index)
    {
        if (!    (index >=0 && index <=4))
        {
            cout << "Index out of bound!";
            exit(EXIT_FAILURE);
        } // this terminates the program.
        else
        {
            return employeeID[index];
        }
    }
};
```

```
int main()
{       database storage1;

        storage1[0] = 1942;
        storage1[1] = 2491;
        storage1[2] = 4219;
        storage1[3] = 2941;
        storage1[4] = 1249;

        cout << "\n\n Displaying ids: \n\n"

        for (int i=0; i < 5; i++)
        {
            cout "\nID-" << i+1 << ": " << storage1[i];
        }

}
```

△ Doing this in next example with using overloaded stream insertion operator

+

✗ Dynamic Array instead of statically allocated array.

```
#include <iostream
#include <cstdlib>  // for use of exit(EXIT_FAILURE)
                    // to terminate program if
class database      // wrong/invalid index is given.
{
    int size;
    int* ptr;
public:                 int getSize(){ return size;}
    database(int s): size(s), ptr(new int[s]){ }

    int& operator [](int index)
    {
        if (!( index>=0 && index<=size ))
        {
            cout<<"Index out of bound";
            exit(EXIT_FAILURE);
        }
        else {
            return ptr[index];
            }
    }
    friend ostream& operator <<(ostream& out,
                                database storage);
};
friend ostream& operator <<(ostream& out, database storage)
{
    out<<"\n\nArray-contents as follows:\n\n";
    for (int i=0; i<(storage.size); i++)
    {
        out<<"  "<<ptr[i];
    }
    out<<"\n\n";
}
```

(object name)

storage

```
int main()
{
    database storage(3);
    //taking user input:

    for (int i=0; i< storage.getsize(); i++)
    {
        cout << "\nPlease enter input for index-"
             << i+1 << ". of array: ";
    //Avoiding lengthy

    storage[0] = 4278;
    storage[2] = 9642;
    storage[1] = 7542;

    cout << storage; // prints the whole
                     Array pointed by
                     ptr.
}
```

Overloading Assignment Operator

* Assigns values to a pre-existing (already constructed object.

* If a user defined Assignment operator is not provided then the default Assignment operator is provided by C++.

```
Complex c;  // default CTOR
Complex c2(4,9); // param. CTOR
Complex c3(c2); // copy CTOR
Complex c4 = c1; // copy CTOR

c4 = c3; // Assignment operator.
```

* Which always does shallow copy

Difference

Copy CTOR	Assignment operator
* making a new object and copying into it (in these cases:)	Copying the contents into pre-existing object (to the left side of the operator.)

(for revision, see slide 50 of 54)

* If class contains pointers, then if still the default assignment operator is used,

then

if one object deletes that ptr / deallocates memory, other object's ptr would be affected too

As shallow copy was done

(SAME PROBLEM AS WAS WITH THE DEFAULT COPY CTOR.)

e.g. code on next eg

```
#include <string>
#include <iostream>
```

length of name including NULL character

```
class name
{       int length; char* ptr;
public:
    name(const char* ptr2="")
    {
        length = strlen(ptr2)
        ptr = new char[length];
        strcpy(ptr, ptr2);
    }
```

//IMP // rem:

// strcpy(destination, source);

// Default Assignment operator looks like thus:

```
/* void operator = (const name &n2)
   {
       length = n2.length;
       ptr = n2.ptr;        //(Shallow-Copy))
   }
*/
```

Deep Copy ON NEXT PAGE.

Assignment operator for Deep

V. IMP

Note: We are going for deep copy in an Assignment because we have any pointer as data member in our class right?

→ So, if a pointer is there let's say of type characters

* And we assume that there are two ~~two classes A, B~~ objects of class Name.

~~then we are doing~~

* And, if in main() we are doing:

```
Name name1 ("Atmar");
Name name2 ("BeHumble");
```

then:

```
name1 = name2;
```

** DID YOU NOTICE?: the length of the two arrays being pointed to by the ptr in the case of name1 & name2 would be different.

→ THIS IS Why: in implementing deep copy in the Assignment operator, we first delete any dynamic memory allocated to the source ptr (to which we have to copy)

the possibility of

* Bcz of varying length of the arrays being pointed to by the ptr of both objects

i.e

object 1 = object 2;

* So first we deallocate any memory allocated to object-1's ptr.

* Then we pass the length of Array being pointed to by the ptr of object 2

and assign it to the variable for storing length in object 1.

* Then we dynamically allocate an array of the new length that has been recently stored in the 'length' variable for object 1 by doing:

```
ptr = new char[length];
```

* Now, that we have same length of Arrays, we can easily use strcpy to copy object 2's array into object 1's array:

Defination of Assignment Operator At Last of page

by doing:

```
strcpy (ptr , n2.ptr);
```

i.e.:

```
strcpy (ptr, object2.ptr);
```

∴

Defination of Assignment operator

** Here we have to pass right-operand i.e. object 2 by reference.

** Recall: overloading Subscript operator "[]" would only be implemented via a member function.

Similarly, overload assignment operator also as a member function bcz L-operand of operator is the object of our user-defined class
→ unlike in the case of stream operators, that's why we went for non-member friend functions for them.

Defination:

```
void operator = (const Name& n2)
{
    delete ptr;
    length = n2.length;
    ptr = new char[length];
    strcpy (ptr, n2.ptr);
}
```

OOP REVISION

(PolyMorphism)

Polymorphism & Virtual functions

Lecture contents:

- Static Binding (Compile-time Polymorphism)
- Dynamic Binding (Run-time Polymorphism)
- Pointers and derived classes
- Derived class members invisible to base class pointers.
- Base class members visible to Base class pointers
- Polymorphism
- Virtual functions
- Virtual destructors

With the help of Apna college

Learned:

Polymorphism

Compile-time Polymorphism A.K.A. Static Binding

- Function Overloading
- Operator Overloading

Run-time polymorphism A.K.A Dynamic Binding

With the help of virtual functions and function overriding within Inheritance

* This would also be Static Binding:

display() of base class would be called

Run: If base class's are not made virtual, then inside main() if its like:

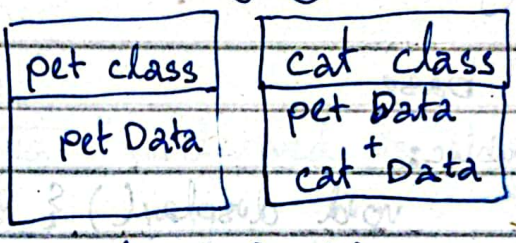
```
{
derived obj;
base* ptr = &obj;
(*ptr).display();
}
```

PAGE-3 (IMP)

Visiting StackOverflow

* Why C++ does not allow pointers of type ~~Base~~ Derived class to point to objects of type Base class without Casting → which is not in our syllabus

Let's picture a scenario:



- There's a base class 'pet'
- There's a derived class from 'pet' as 'cat'.

* Now whatever or however is pet defined as by the programmer

* The cat would at least be a pet, and maybe more.

i.e the cat would have all the characteristics defined of a pet in pet class and the cat could have its own distinct characteristics as well.

Notes from Slides

Static Binding

'Binding' that takes place during Compilation

↑ Connecting a function call to a function body.

e.g #1

```
class base
{   public:
        void display() {
        cout << "\n Base class's display called\n";
                }
};
class derived {}: public base {
 public:
        void display() {
            cout<< "\n Derived class's display called\n";
                }
};
void main() {
        base obj1;
        obj1.display(); // calls base::display()

        derived obj2;
        obj2.display(); // calls derived::display()

        }
```

IMP

Static Binding

Remember: Dynamic Binding / runtime polymorphism would only occur once Base class's member functions (overridden in derived classes) are declared as virtual in the base class.

Eg.2

```
class base {
 public:
        void display() {
          cout<< "\nBase is legend\n";
                }
};
class derived {
public:
        void display() {
          cout << ~~derived~~ "\nderived is legend\n";
                }
};
int main(){

        derived obj;
        base *ptr = &obj;

        (*ptr).display(); //use of static binding
                          // function call would depend
                            on type of pointer
                          // base::display called.

        obj.display(); //use of static binding
                       // derived::display called.
        base legend;
        legend.display(); // use of static binding
                          // base::display called.
}
```

IMP [Use of refference rather than pointer]

E.g #3 of Static Binding On next pg

Rem: If virtual is not used with the member functions of base class,

Then at compile-time,

(1) if a pointer is dereferenced to call a member function like in E.g.#2

OR

(2) if an object is passed by a reference, and then that reference into some other function

is used to call a member function like in E.g #3

Then

In Case-1

- The compiler would Statically bind (at compile time) the function call with the function body

** BASED ON THE TYPE OF POINTER.

In Case-2

- The compiler would Statically bind (at compile time) the function call with the function body

** BASED ON THE TYPE OF REFERENCE.

IMP * E.g #3 (Use of reference & STATIC BINDING)

```
class base { public:
                void display() { cout<< "In Base legend\n"; }
           };
class derived : public base // inheritance
           { public:
                void display() { cout<< "In Derived legend\n"; }
           };

void fun(base& obj)
{
    obj.display();  // calls base::display
                    // rem: function call binded with function body based on TYPE OF REFERENCE incase of static binding just like TYPE OF POINTER incase of POINTERS.
    //no matter of object-type is base or derived, still base::display() would be called bcz
}
int main() {
    derived obj;

    fun(obj);  // calls base::display()
               // due to static Binding rules...
               // case-2 applied here (see prev. pg.)
    base obj2;

    fun(obj2); // base::display() called.

    }
```

DYNAMIC BINDING

* Delaying the binding-process to the runtime is called

Occurs

Once the Base class's member functions are declared virtual.

Side-Notes: 'virtual' keyword can ~~always~~ come before or after the return type of function but in virtual inheritance it's like:

< class derived : public virtual base { }; >

written after public

Polymorphism:

The ability to decide at runtime that which among the several overrided member functions is to be called by checking the actual type (the type of the object passed) and then calling that object's definition for that overrided function.

i.e. ~~not basing~~ this decision is now not based upon the type of pointer or the type of reference.

But rather the type of object.

DYNAMIC BINDING

Re-implementing the previous [e.g #2, e.g. #3] using

```
class base {
public:
        virtual void display() {
            cout << "base is legend\n";
            };                    }
class derived {: public base
{
 public:
          void display() {
            cout << "\n derived is legend\n";
                          }
};
void fun( base& obj)
{
      obj.display();  // function binding would ~~not~~ now
}                     // be dependant upon the type
int main() {          // of object, that is passed.

      base legend1;
      derived legend2;

      base *ptr = & legend2
     (*ptr).display(); // calls derived::display()

     fun(legend2); // calls derived::display()

     base *ptr2 = & legend1;
     (*ptr2).display(); // calls base::display()
     fun(legend1);   // calls base::display()
          }
```

NOTE: DO NOT CONFUSE/MIX HOW PROGRAM BEHAVES IN THE CONCEPT OF ~~SINGLE~~ SIMPLE OVERRIDING & IN THE CASE OF STATIC BINDING.

SEE THIS:

```
#include <iostream>   using namespace std;
class base
{ public:
      void display() {
          cout << "base legendary\n";
                    }
};
class derived
{ public:
      void display() {
          cout << "derived legendary\n";
                    }
};
int main() {
      derived obj;
          obj.display(); // calls derived::display()
                         // use of overriding
      base *ptr = &obj;

      ptr -> display(); // calls base::display()
                        // due to static Binding
      }
```

Rem:

* Derived class pointer can't point to Base class object
[Compiler error is thrown]

[See pg. 3]

* Base class ptr pointing to object of derived class

Base class's member function declared virtual

→ Then
Dynamic Binding
or
Runtime PolyMorph.

* binding function call to function definition would be based on the type of object passed or the type of object being pointed to.

Base ~~Derived~~ class's member function NOT declared virtual

→ Then
Static Binding
or
Compile Time PolyMorph.

Now, binding function call to function definition would be based on the type of reference stated or the type of pointer

See previous code examples

VIRTUAL DESTRUCTORS

* Virtual functions and virtual destructor go hand in hand.
* Making the destructor virtual will ensure that the correct destructor will be called for an object when pointers to the base classes are used.
* If destructor of base class is not declared as virtual, then the destructor is statically bound based on the type of pointer and the destructor of derived class won't be called when the object is deallocated via the base class pointer.

So the object is partially destroyed.

For Visualization see in PC:

OOP/Revising-for-final/revisingOOP/more.cpp

* A simple rule: To have a virtual destructor whenever your class has at least one virtual function.

** Note: ((IMP))

If the base class destructor is virtual, the derived classes destructors will be virtual automatically.

** If Virtual Destructor is used, then:

* Don't forget the concept of CSTRs & DSTRs

* Note: there's inheritance involved in implementing polymorphism.

So doing:

```
baseClass *ptr = new derived;
```

would call

① . baseClass's CSTR

then → ② . derivedClass's CSTR

AND doing:

```
delete ptr;
```

would call

① . derivedClass's DSTR

then → ② . baseClass's DSTR

[See next page]

Similarly,

if it's just:

```
base *ptr = new base;
delete ptr;
```

Then

→ this would call: baseClass's CSTR

AND

This would call: baseClass's DSTR

** For visualization of above thesis xD, see

OOP/Revising_for_final/revisingOOP/nonVirtualDestructor.cpp

Abstract & CONCRETE Classes

Purpose: to define an interface/blueprint

act classes: classes from which
ver intended to instantiate

which are too generic
al objects

orks for the
derived
classes

antiate objects.
for every
ine.

becomes an abstract
oy declaring ~~any of its~~
virtual function.

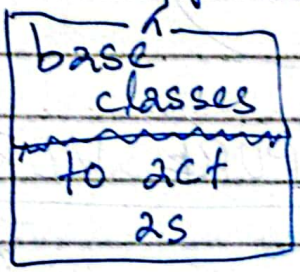
=

ct classes are not required
implement their pure virtual functions.

Derived classes MUST define all
pure virtual functions, failure to
define even 1 let's say out of 10 would make

→ Purpose: to define an interface/blueprint

* **Abstract classes**: classes from which it's never intended to instantiate any objects
  - → i.e classes which are too generic to define real objects
  - Normally used as frameworks for the derived classes



* **Concrete Classes**
  - → Classes which are used to instantiate objects.
  - * Must provide implementation for every member function they define.

IMP: A Base class becomes an abstract base class by declaring ~~any of its~~ a pure virtual function.

* Abstract classes are not required to implement their pure virtual functions.

** Derived classes MUST define all pure virtual functions, failure to define even 1 let's say out of 10 would make

** that derived class an abstract class too.

So that derived class would then become un-instantiable in main().

Disadvantage

↳ Bcz Abstract classes are

MAKING PURE VIRTUAL

```
class A
{ public:
      virtual void fun() const = 0;
};
```

"Pure Specifier"

Tells compiler that there's no implementation of it inside this class.

what makes it PURE

Note: In multi-level inheritance, it can happen that other than the Abstract base class, some derived class become abstract too (for not providing definitions for all pure virtual functions)

***** while some derived classes become concrete classes

1. ** (maybe due to they having defined all the pure virtual functions)

OR

2. ** (maybe due to they defining only some of the total pure virtual functions while definitions for the other pure virtual functions could have been inherited to them by the derived classes above them in the chain of inheritance.

OR

3. they defining none of the pure virtual functions; but due to inheritance and being too down in the chain of inheritance, they get as inheritance all the definitions for all the pure virtual functions.

→ These 3 scenarios are cases for derived classes to become concrete classes rather than abstract classes.

Rem: The rules of polymorphism are used in implementing Abstract Classes.

i.e. Notice: In the later examples that once member functions are made virtual, polymorphism automatically comes into play.

i.e. Dynamic Binding would occur with regards to functions when calling functions through Base class pointer. of derived classes

* Also note that Although abstract classes are un-instantiable, pointers of type Abstract classes could definitely be created inorder to use runtime Polymorphism bcz member functions of base class would be pure virtual.

i.e. that they can't instantiate any objects.

(Defining pure virtual functions for the Abstract Base Class as well)

* Also note that pure virtual functions, although meant to be left undefined in the Abstract base class; can still be defined for ~~that~~ Abstract class to provide a common piece of code for the derived classes inorder to facilitate reuse.

→ But they can not be defined within the Abstract Base class's curly brackets

↳ i.e. they cannot be defined as inline member functions

* However, we can define them outside the class using scope resolution operator.

For this See code e.g. 2 coming later

e.g 1 of Abstract Classes

```
class A {
public:
        virtual void x() = 0;
        virtual void y() const = 0;
          };

class B: public A
{ public:
        virtual void x(); // provides some
                          // definition as well
};

class C: public B
{ public:
        virtual void y(); // provides some
                          // definition as well
};

int main() {

    // A obj; // totally invalid
              // Abstract classes can't instantiate objects

    // B obj; // Abstract class can't instantiate
              // guess why B is abstract

    C obj; // totally legit!!! (Concrete class)

    A *ptr; // legit
    B *ptr; // legit
}
```

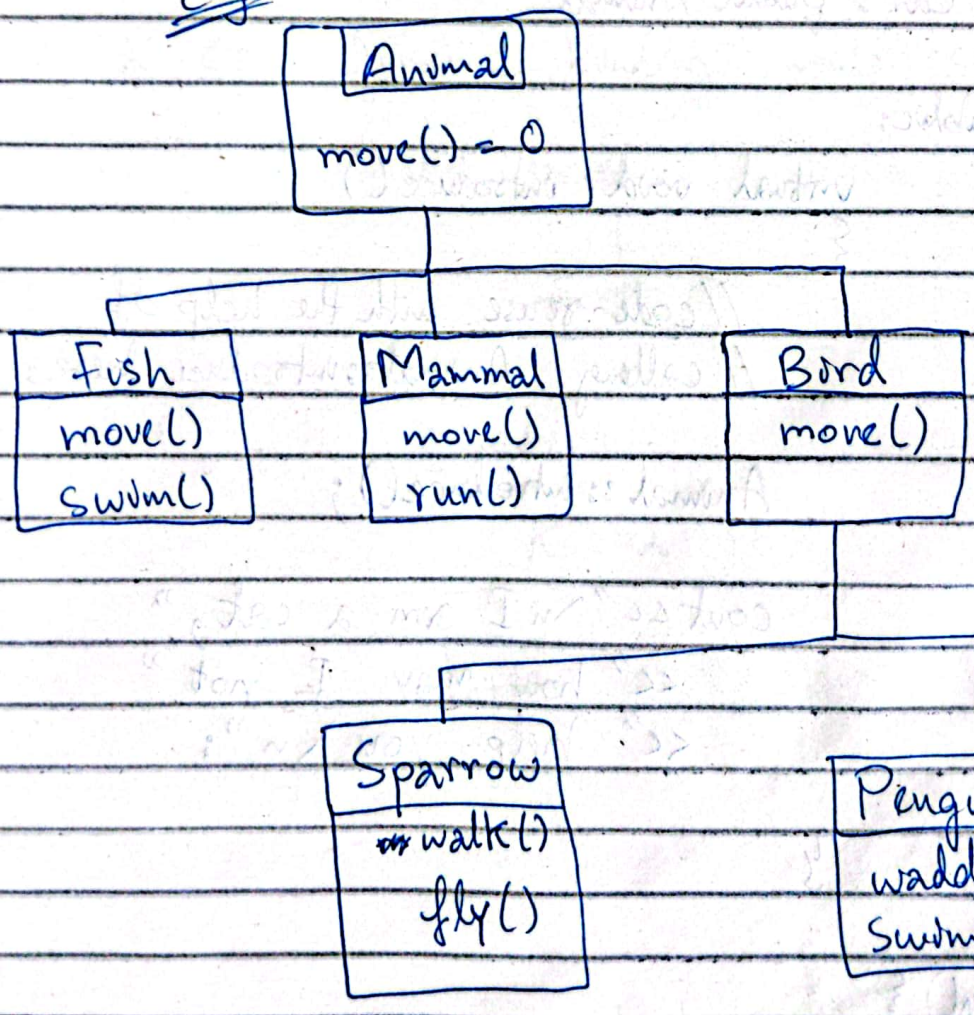
***** pointers of type Abstract class can be made to enable polymorphism

Refinement

* Pure virtual functions let us specify interfaces appropriately & retain implementation decisions till later derived/sub-classes.

* As the hierarchy of classes extends, the pure virtual functions of the Abstract Base class are replaced by virtual functions that fill in and may override the way of implementation.

e.g.



code e.g 2 of Abstract classes

```
class Animal // Abstract base class
{ public:
        virtual void introduce() const = 0;
};

void Animal::introduce()
{
    cout << "\nI am an animal! Yeeheeeee!\n";
}

class cat : public Animal
{
  public:
        virtual void introduce()
        {
            //code-reuse with the help of
            //calling Animal::introduce here:

            Animal::introduce();

            cout << "\n I am a cat, "
                 << " how may I not "
                 << " help you \n";

        }
};
int main() {
        cat persian;
        persian.introduce();
    }
```

* In the prev. eng. code,
runtime polymorphism could be added by doing this inside main().

```
{
    cat persian;
    Animal *ptr = &persian;
    (*ptr).introduce();
}
```

* As member functions of base class are virtual
* So Dynamic binding would occurr and cat::introduce() would be called.

((Payroll system implement karna sekha...
Do it b4 paper
E.A.))