Welcome

Object Oriented Programming (UTA018)

(Session: 2026-2027 ODD)

Instructor: Dr. Suresh Raikwar

Department of Computer Science and Engineering,

Thapar Institute of Engineering and Technology, Patiala, Punjab, India

Important Instruction(s):

- This material is intended for guidance and conceptual understanding. It does not guarantee that examination questions will be drawn directly from it.
- Merely studying this material is not sufficient for achieving good performance.
- Sessional Tests, MST, and EST will assess your understanding of Object-Oriented Programming concepts through **logical reasoning, analytical thinking, and problem-solving**.
- Students are strongly encouraged to attend regular classes and practice programming problems from multiple sources to develop a deeper conceptual understanding and improve problem-solving skills.

UTA 018: OOPs
Poly-morphism



Contents

- Polymorphism
- Function Overloading
- Default Function Arguments
- Ambiguity in Function Overloading

Polymorphism

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

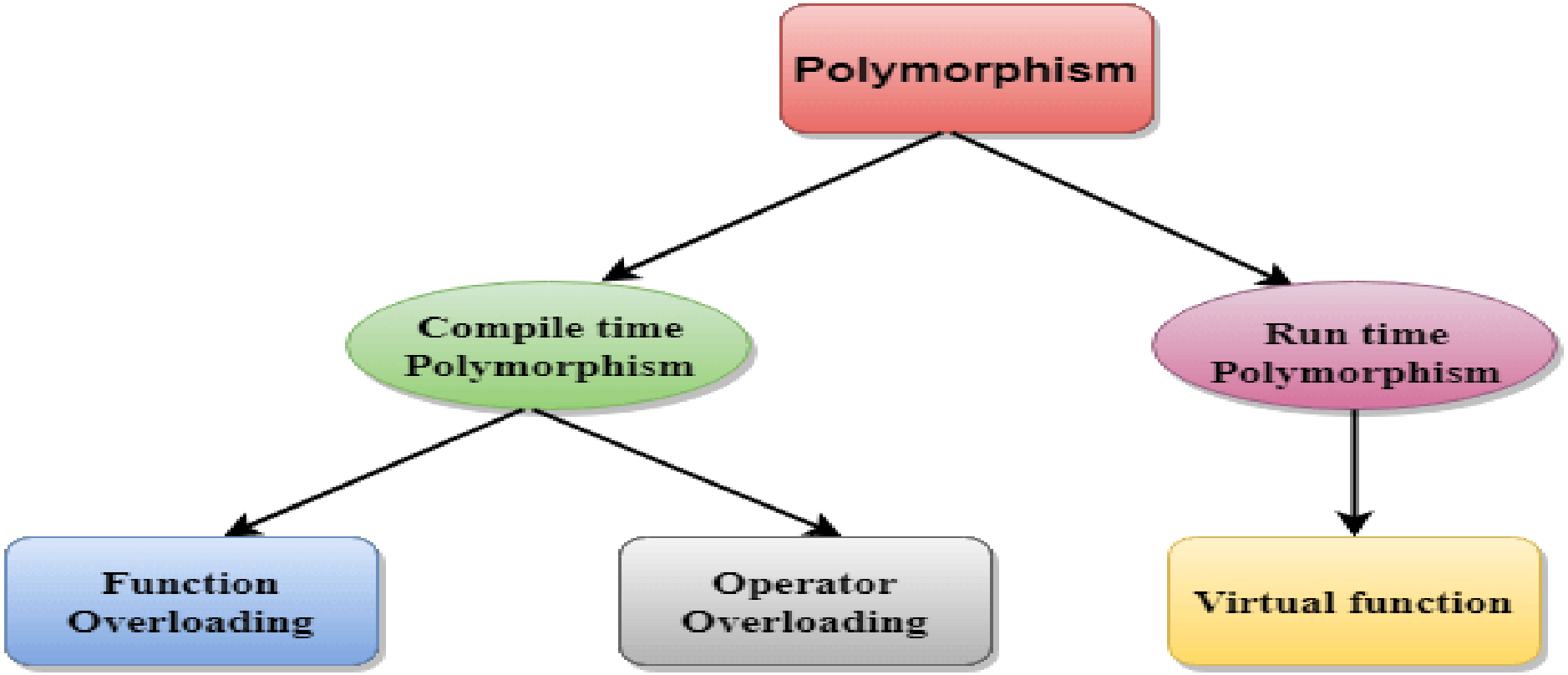
- Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee.

- So a same person posses different behavior in different situations. This is called polymorphism.



Types of Polymorphism

- Compile Time Polymorphism
- Run Time Polymorphism



Compile Time Polymorphism

- **Early binding** refers to events that occur at **compile time**. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation).
- Examples of early binding include **normal function calls** (including standard library functions), **overloaded function** calls, and *overloaded operators*.
- The main advantage to early binding is **efficiency**. Because all necessary information to call a function is determined at compile time, these types of function calls are very fast.

Run Time Polymorphism

- The opposite of early binding is late binding.
- As it relates to C++, late binding refers to function calls that are not **resolved until run time**.
- As function calls are not determined at compile time, ***the object and the function are not linked until run time***.

Run Time Polymorphism

- The main advantage to late binding is flexibility.
- Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code."
- Keep in mind that because a function call is not resolved until run time, late binding can make for **somewhat slower execution times**.
- ***Virtual functions are used to achieve late binding***.

Function Overloading

- Function overloading is the process of using the same name for two or more functions.
- The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters.
- It is only through these differences that the compiler knows which function to call in any given situation.

Example

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
```

```
double myfunc(double i)
{
return i;
}
int myfunc(int i)
{
return i;
}
```

Output:
10 5.4

Example 2

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
```

```
int myfunc(int i)
{
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

Output:
10 20

Key Points

- The key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded.
- For example, this is an invalid attempt to overload **myfunc( ):**

```
int myfunc(int i);
float myfunc(int i);
```

// Error: differing return types are insufficient when overloading.

- Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

- Remember, to the compiler ***p is the same as p[ ]. Therefore, although the two** prototypes appear to differ in the types of their parameter, in actuality they do not.

Example 3

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

void display(int);
void display(float);
void display(int, float);

int main() {

     int a = 5;
     float b = 5.5;

     display(a);
     display(b);
     display(a, b);

     return 0;
}

void display(int var) {
     cout << "Integer number: " << var << endl;
}

void display(float var) {
     cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
     cout << "Integer number: " << var1;
     cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Example 4 (Function overloading in classes)

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

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};
```

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Issue of Name Hiding

In C++, when you declare a function in the **derived class** that has the **same name** as a function in the **base class**, all functions with that name in the base class become **hidden** in the derived class, regardless of their parameter types.

```cpp
#include<iostream>
using namespace std;
class base{
    int a;
    public:
        void showData(int x){
            a=x;
            cout<<"a:"<<a<<endl;
        }
};
class derived:public base{
    int b;
    public:
        void showData(){
            b=7;
            cout<<"b:"<<b<<endl;
        }
};
int main(void){
    base base_ob;
    base_ob.showData(5);

    derived derived_ob;
    derived_ob.showData();
    derived_ob.showData(6);//This will cause error

    return 0;
}
```

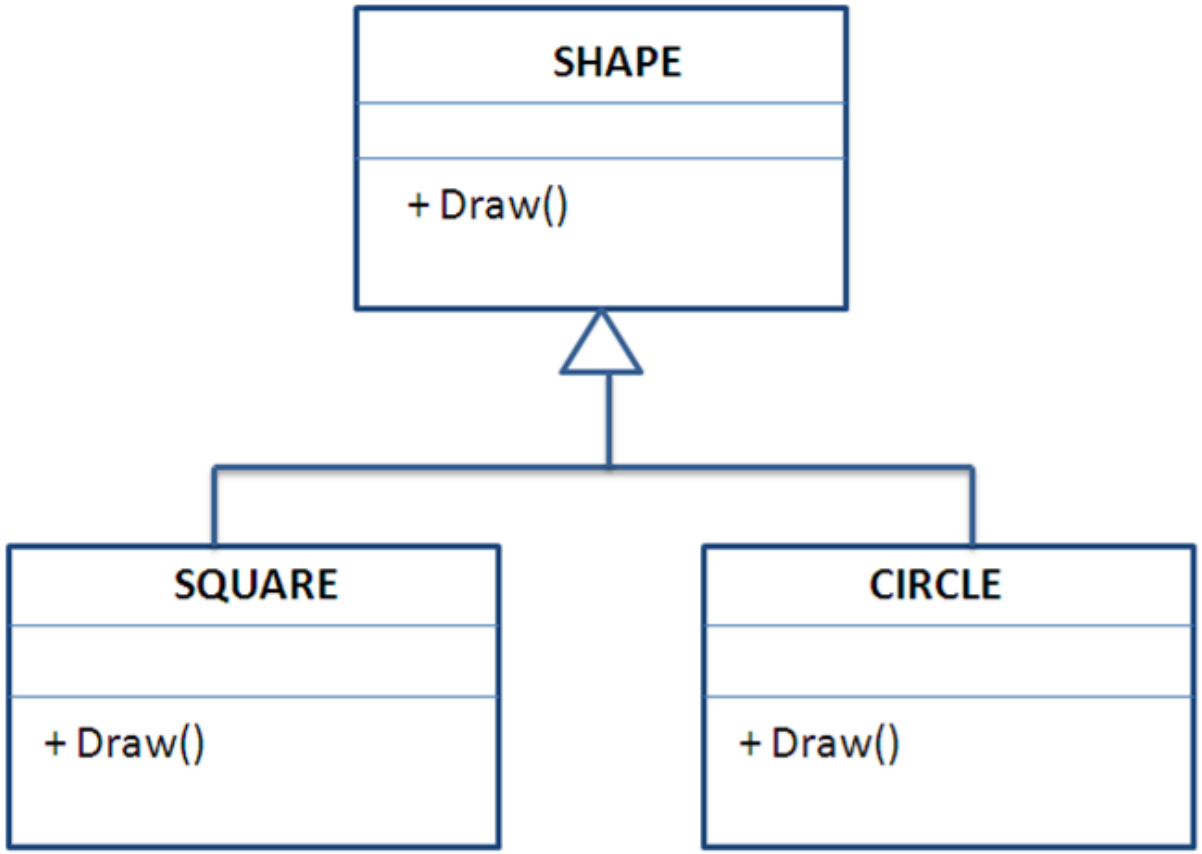
```
using namespace std;
class base{
    int a;
    public:
        void showData(int x){
            a=x;
            cout<<"a:"<<a<<endl;
        }
};
class derived:public base{
    int b;
    public:
        using base::showData;
        void showData(){
            b=7;
            cout<<"b:"<<b<<endl;
        }
};
```

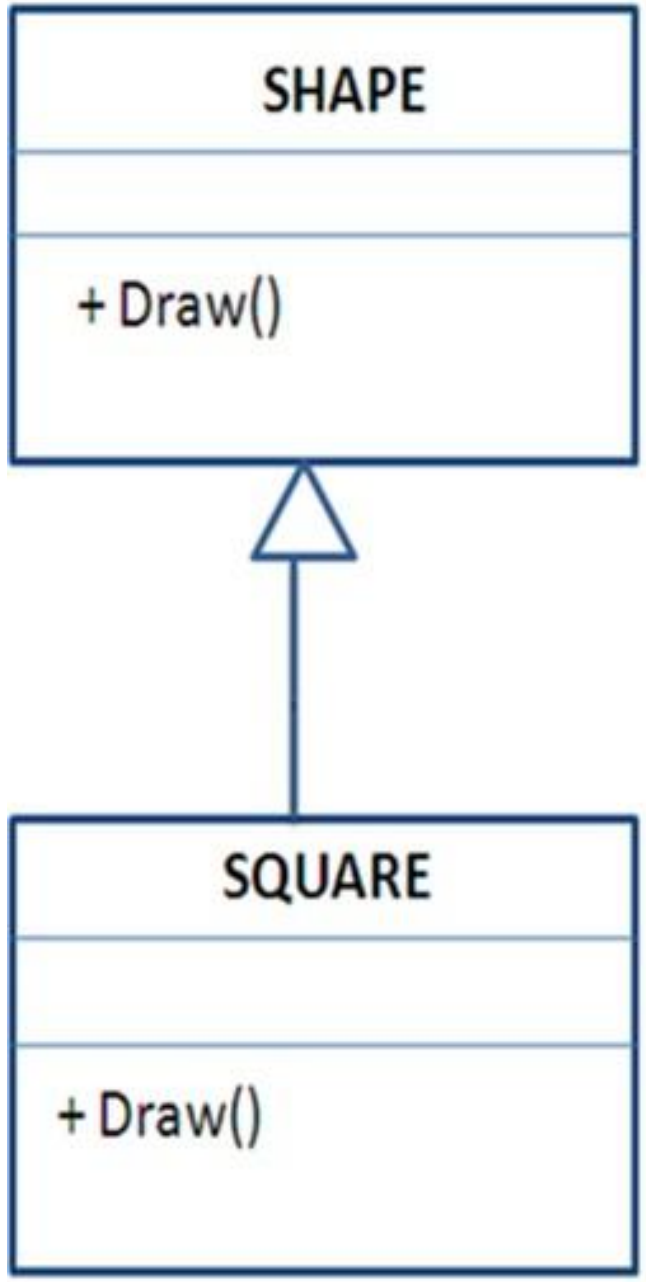
Inheritance based Polymorphism

Function Overriding through class Inheritance:

It is a language feature that allows a subclass overrides (replaces) the implementation in the super super class by providing a method that has same name, same parameters and same return type as the method in parent class.

Example






```
Class SHAPE
{   public:
virtual void Draw()
{
  cout<<" I am shape
  class";
}
};

Class SQUARE : public SHAPE
{ public:
        void Draw()
{
cout<<"I am square";
}
};
```

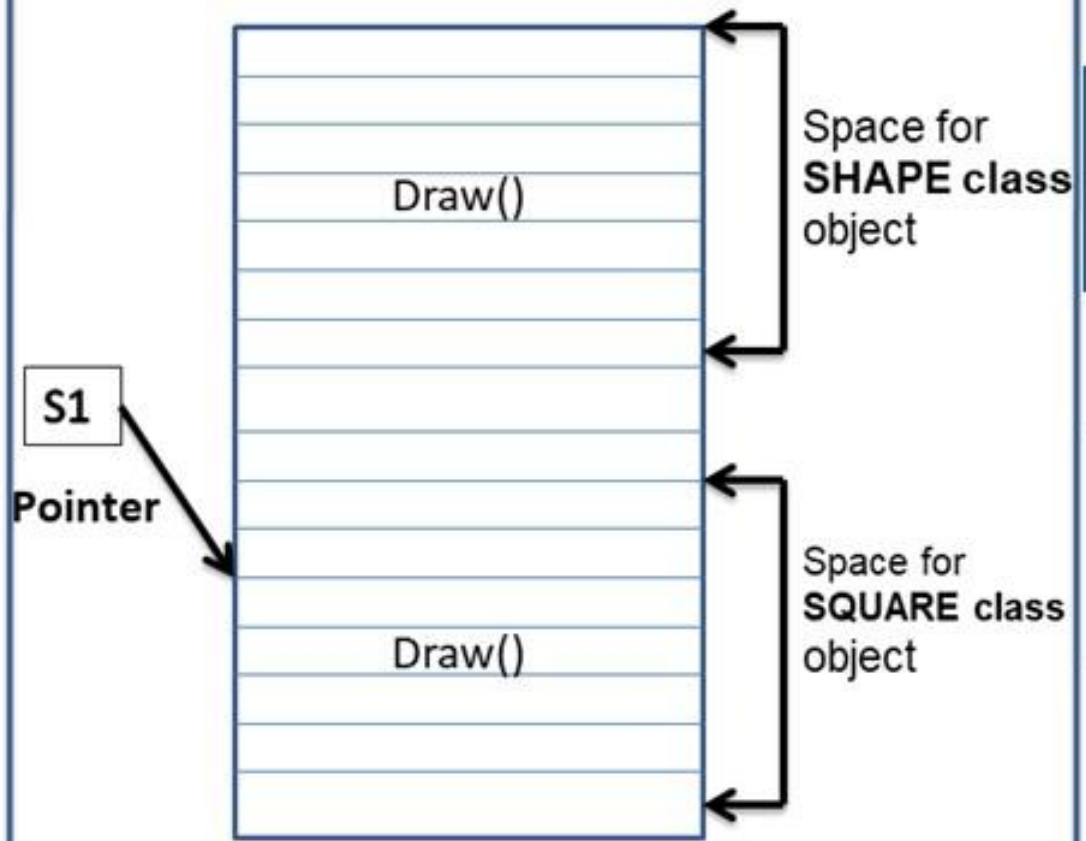
Create Objects

SHAPE *S1 = new SHAPE();

S1→Draw();

S1 = new **SQUARE**();

S1→Draw();

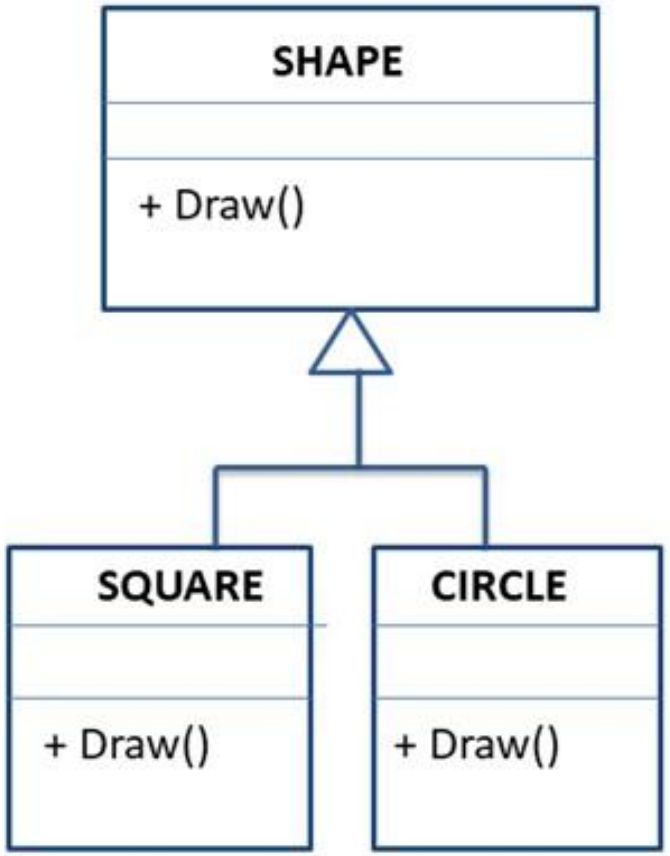


Output

I am shape class

I am Square

Extend the Logic



Create Objects

SHAPE *S1 = new **SHAPE();**

S1→Draw();

S1 = new **SQUARE()**;

S1→Draw();

S1 = new **CIRCLE();**

S1→Draw();

Output

I am in Shape class

I am Square

I am Circle

Virtual Functions

- A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a **virtual** keyword in a base class.
- The resolving of a function call is done at runtime.
- Be aware that the virtual function mechanism works only with pointers to objects and, with references, not with objects themselves.

Rules for virtual function

- Virtual functions cannot be static.
- Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in the base as well as the derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.

Inheritance and Static Functions

- They are **inherited into the derived class**.
- If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden.
- Static Member functions can never be virtual.

Abstract base class - interface

- It describes the capabilities of a C++ class without committing to a particular implementation
- A class is made abstract by declaring at least one of its functions as **pure virtual** function i.e. by placing **"= 0"** in its declaration

```
class Animal{
public:
virtual void sound() = 0;
void sleeping() {cout<<"Sleeping"; }
};
class Dog: public Animal{
public:
void sound() {cout<<"Woof"<<endl;}
};
int main(){
        Dog obj;
        obj.sound();  // Output: Woof
        obj.sleeping(); //Output: Sleeping
}
```



```
class Animal{
public:
virtual void sound() = 0;
void sleeping() {cout<<"Sleeping"; }
};
class Dog: public Animal{
public:
void sound() {cout<<"Woof"<<endl;}
};
int main(){
        Animal *obj = new Dog;
        obj->sound();
}
```



Abstract Class

- An abstract class is designed to act as base class. It is a design concept in program development and provides a base upon which other classes may be built.
- Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

```
//Abstract base class
class Base {
 public:
virtual void show() = 0; };

class Derived:public Base {
public:
void show()
 {
cout << "Implementation of Virtual
   Function in Derived class";
 }
};
```

```
int main()
{
//Base obj; (Compile Time Error)
 Base *b;
 Derived d;
 b = &d;
 b->show();
  return 0;
}
```

Note:

- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

Overloading vs overriding

- Function overloading – same function name but different arguments. Functions are defined in the same class.
- Function overriding – same function name and arguments. Defined in different classes.

Compile time and Run time Polymorphism

- Compile time OR static polymorphism is executed using function overloading.

- Run time polymorphism or dynamic/late binding is done using function overriding and ***virtual functions***.

Compile Time Polymorphism	**Run-Time Polymorphism**
At Compile time, which functions to be called is decided.	At Runtime, which function to be called is decided.
Also known as **early** or **static binding**	Also known as **late** or **dynamic binding**
It executes faster because the function is resolved at compilation time only.	It executes slower because the function is resolved at Run-time.
It is achieved through function and *operator overloading*	It is achieved through function overriding and virtual functions

Ambiguity in Function Overloading

- You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions.
- When this happens, the situation is said to be *ambiguous.*

Ambiguity in Function Overloading

```
int myfunc(double d);
// ...
cout << myfunc('c');
// not an error, conversion applied
```

- Although automatic type conversions are convenient, they are also a prime cause of ambiguity.

Example

```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);

int main()
{
cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
cout << myfunc(10); // ambiguous return 0;
}
```

UTA 018: OOPs
Poly-morphism Part-2
(Operator Overloading)



Introduction

- Operator overloading
  - ➢ Enabling C++'s operators to work with class objects
  - ➢ Using traditional operators with user-defined objects
  - ➢ Examples of already overloaded operators
    - Operator **<<** is both the stream-insertion operator and the bitwise left-shift operator
    - Operator **>>** is both the stream-extraction operator and the bitwise right-shift operator

Introduction (Cont…)

- Overloading an operator
  - Write function definition as normal
  - Functionname is keyword **operator** followed bythe symbol for the operator being overloaded
  - For e.g., **operator+** used to overload the addition operator (**+**)

```
Returntype operator+(arg list){
    //body
}
```

What are Operators in C++?

- **Arithmetic Operators:** +, −, /, ∗, %
- **Assignment Operators:** +=,+=, −=,−=, /=,/=, ∗=,∗=, %=%=
- **Relational Operators:** ==,==, !=,!=, >=,>=, <=,<=,*etc*.
- **Logical Operators:** &&,&&, ||,||, !!
- **Bitwise Operators:** &,&, |,|, >>, <<*etc*.
- **Other Operators:** sizeof, ?:,*etc*.

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - number of operands cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators **&**, *****, **+** and **−** each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

Restrictions on Operator Overloading

- All the C++ operators can be overloaded except the following:
  - Scope Resolution Operator (::)
  - Conditional Operator (? :)
  - Size Operator (sizeof)
  - Class Member Access Operators (. and .*)

Operator Functions as Class Members vs. **as friend Functions**

- Operator functions can be either member or non- member functions of a class.
- Operator functions as member functions
  - Leftmost operand must be an object of the class
- Operator functions as non-member functions
  - Must be **`friend functions`** of the class
  - Enables the operator to be commutative

Note: *If left operand is of a different type than class, operator function must be a non-member function*

Creating a Member Operator Function

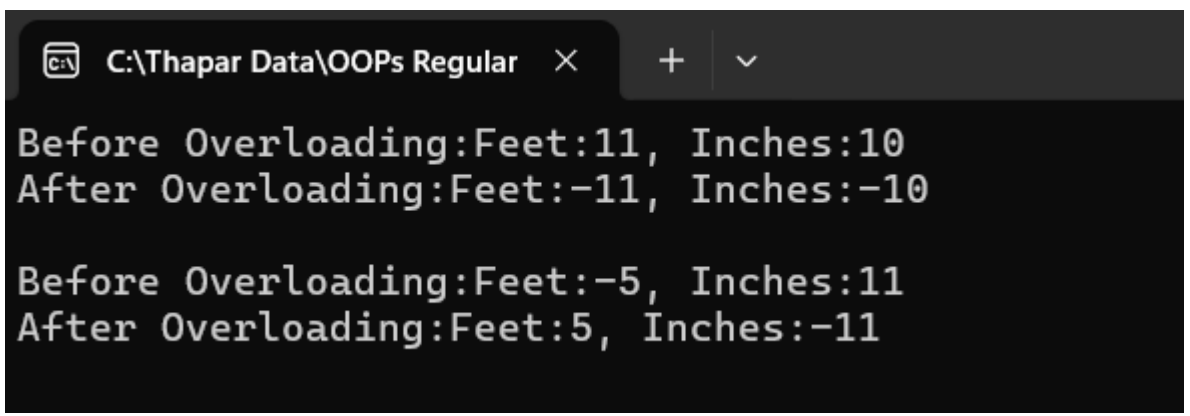
```
returntype classname::operator opname(arg-list)
{
    // operations
}
```



Overloading of Unary Operator - (using **member function)**

```
#include <iostream>
using namespace std;
class Distance{
    private:
        int feet;
        int inches;
    public:
        Distance():feet(0),inches(0){}
        Distance(int f, int i):feet(f),inches(i){}

        void displayDistance(){
            cout<<"Feet:"<<feet<<", Inches:"<<inches<<endl;
        }
        // overloaded unary minus (-) operator
        Distance operator- (){
            feet=-feet;
            inches=-inches;
            return Distance(feet,inches);
        }
};
```

```
int main(){
    Distance D1(11, 10), D2(-5, 11);
    cout<<"Before Overloading:"; D1.displayDistance();
    -D1; // activates operator-() function
    cout<<"After Overloading:";D1.displayDistance(); // D1display
    cout<<"\nBefore Overloading:"; D2.displayDistance();
    -D2; // activates operator-() function
    cout<<"After Overloading:";D2.displayDistance(); // display D2
    return 0;
}
```



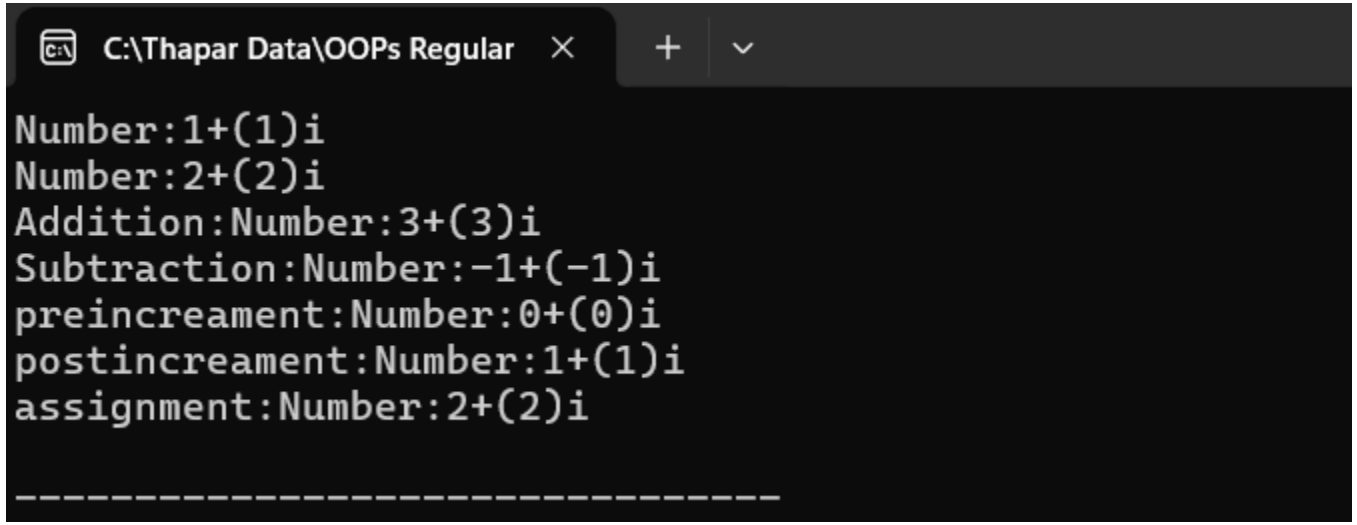
Complex Class Example

```
#include<iostream>
using namespace std;
class Complex{
    private:
        int real, img;
    public:
//      Complex():real(0),img(0){}
        Complex(int r=0,int i=0):real(r),img(i){}

        void showData(){
            cout<<"Number:"<<real<<"+("<<img<<")i"<<endl;
        }

        Complex operator+(Complex c){
            Complex temp;
            temp.real=real+c.real;
            temp.img=img+c.img;
            return temp;
```

```
//  real=real+c.real;
//  img=img+c.img;
//  return *this;
}
Complex operator-(Complex c){
        real=real-c.real;
        img=img-c.img;
        return *this;
}
Complex operator++(){
    real++;
    img++;
    return *this;
}
Complex operator++(int){
    ++real;
    ++img;
    return *this;
```

```
        Complex operator=(Complex c){
            real=c.real;
            img=c.img;
            return *this;
        }
};
int main(void){
    Complex c1(1,1),c2(2,2);
    c1.showData();
    c2.showData();
    Complex c3;
    c3=c1+c2; //equivalent to c3=c1.operator+(c2);
    cout<<"Addition:";c3.showData();
    c3=c1-c2;
    cout<<"Subtraction:";c3.showData();
    c3=++c1;
    cout<<"preincreament:";c3.showData();
    c3=c1++;
    cout<<"postincreament:";c3.showData();
    c3=c1=c2;
    cout<<"assignment:";c3.showData();
    return 0;
}
```

Continued…


C:\Thapar Data\OOPs Regular
Number:1+(1)i
Number:2+(2)i
Addition:Number:3+(3)i
Subtraction:Number:-1+(-1)i
preincreament:Number:0+(0)i
postincreament:Number:1+(1)i
assignment:Number:2+(2)i

Overloading using Friend Functions

- There are certain situations where we would like to use a friend function rather than a member function.
- For example, if we want to use two different types of a binary operator, say, one an object and another a built in type as shown below:

  A=B+2, where A and B are objects of same class. This will work for a member function, but the statement **A=2+B**; will not work.
- This is because the left-handed operand which is responsible for invoking the membership function should be an object of the same class.
- Friend function allows both approaches as it is invoked without the use of objects.

Continued…

- Unary operators, overloaded by means of a member function, take no explicit arguments, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).

- Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

- When using binary operators overloading through member function, the left hand operand must be an object of the relevant class.

```cpp
#include<iostream>
using namespace std;
class Complex{
    private:
        int real, img;
    public:
//      Complex():real(0),img(0){}
        Complex(int r=0,int i=0):real(r),img(i){}

        void showData(){
            cout<<"Number:"<<real<<"+("<<img<<")i"<<endl;
        }

        friend Complex operator+(Complex,Complex);
};
Complex operator+(Complex c1,Complex c2){
            Complex temp;
            temp.real=c1.real+c2.real;
            temp.img=c1.img+c2.img;
            return temp;
        }
int main(void){
    Complex c1(1,1),c2(2,2);
    c1.showData();
    c2.showData();
    Complex c3;
    c3=c1+c2;
    cout<<"Addition:";c3.showData();
    return 0;
}
```

```
Number:1+(1)i
Number:2+(2)i
Addition:Number:3+(3)i

--------------------------------
```

Restrictions on application of friend **function**

- Operators that can not be overloaded using a friend function are:

=	Assignment operator
( )	Function call operator
[ ]	Subscript operator
->	Class Member Access Operator

Overloading new and delete

```
// Allocate an object
void *operator new(size_t size)
{
/* Perform allocation.  Throw
bad_alloc on failure.
Constructor called
   automatically. */
return pointer_to_memory; }
}
```

```
// Delete an object
void operator delete(void
   *p)
{
/* Free memory pointed to by p.
Destructor called
   automatically. */
```

Example

```
#include <iostream>
using namespace std; class loc {
int longitude, latitude; public:
loc() {}
loc(int lg, int lt) longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " "; cout << latitude <<
"\n";
}
void *operator new(size_t size);
void operator delete(void *p);
};
```

```
// new overloaded relative to loc
void *loc::operator new(size_t size)
{
void *p;
cout << "In overloaded new.\n";
p = malloc(size);
return p;
}
// delete overloaded relative to loc void
loc::operator delete(void *p)
{
cout << "In overloaded delete.\n"; free(p);
}
```

Continued…

```
int main()
{
loc *p1, *p2;

p1 = new loc (10, 20);
p2 = new loc (-10, -20);
}
p1->show();
p2->show();
delete p1;  delete p2;
return 0;
}
```

```
In overloaded new.
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

Overloading [ ] operator

- In C++, the [ ] is considered a binary operator when you are overloading it.

- Therefore, the general form of a member operator[ ]( ) function is as shown here:

  returntype classname::operator[](int i)

  {

  // . . .

  }

- Technically, the parameter does not have to be of type int, but an operator[ ]( ) function is typically used to provide array subscripting, and as such, an integer value is generally used.

- Given an object called obj, the expression obj[3] translates into this call to the operator[ ]( ) function: obj.operator[](3)

Example 1

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int operator[](int i)
 { return a[i];
}
};
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
return 0;
}
```

Output:

2

Example 2

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:
atype(int i, int j, int k) {
a[0] = i;
a[1] = j;
a[2] = k;
}
int &operator[](int i) {
return a[i];
 }
};
```

```
int main()
{
atype ob(1, 2, 3);
cout << ob[1]; // displays 2
cout << " ";
ob[1] = 25; // [] on left of =
cout << ob[1]; // now displays 25
return 0;
}
```

Output:

2 25

Overloading ()

- When we overload the ( ) function call operator, we are not creating a new way to call a function.

- Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Example:

double operator()(int a, float f, char *s);

and an object O of its class, then the statement

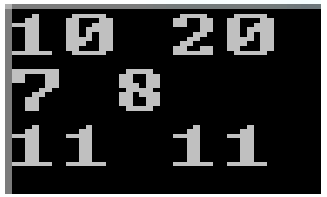
O(10, 23.34, "hi");

translates into this call to the operator( ) function.

O.operator()(10, 23.34, "hi");

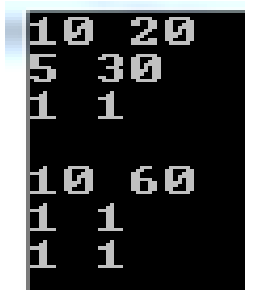
Example

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt)
{
longitude = lg; latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);  loc operator()(int i, int j);
};
// Overload ( ) for loc
loc loc::operator()(int i, int j)
{
longitude = i; latitude = j;
return *this;
}
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}

int main()
{
loc ob1(10, 20), ob2(1, 1);
ob1.show();
ob1(7, 8); // can be executed by itself
ob1.show();
ob1 = ob2 + ob1(10, 10); // can be used in expressions
ob1.show();
return 0;
}
```



Overloading the Comma Operator

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) { longitude = lg;  latitude = lt;
}
void show() {
cout << longitude << " "; cout << latitude << "\n";
}
loc operator+(loc op2); loc operator,(loc op2);
};
// overload comma for loc
loc loc::operator,(loc op2)
{
loc temp;
temp.longitude = op2.longitude; temp.latitude =
op2.latitude;
cout << op2.longitude << " " << op2.latitude << "\n";
return temp;
}
// Overload + for loc
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;  return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);
ob1.show();
ob2.show();
ob3.show();  cout << "\n";
ob1 = (ob1, ob2+ob2, ob3);
ob1.show(); // displays 1 1, the value of ob3
return 0;
}
```



Overloading ->

- The –> pointer operator, also called the *class member access operator, is considered* a unary operator when overloading.

- Its general usage is shown here:

  object->element;

- Here, *object is the object that activates the call. The operator–>( ) function must return* a pointer to an object of the class that operator–>( ) operates upon.

- The *element must be* some member accessible within the object.

- The example illustrates overloading the –> by showing the equivalence between ob.i and ob–>i when operator–>( ) returns this pointer:

Example

```
#include <iostream> using namespace std; class myclass {
public:
int i;
myclass *operator->()
{
return this;
}
};
int main()
{
myclass ob;
ob->i = 10;
// same as ob.i
cout << ob.i << " " << ob->i;
return 0;
}
```

Output:

10 10

UTA 018: OOPs Poly-morphism Part-3 (Type Conversion)



What is Type Conversions?

- When constants and variables of different types are mixed in an expression, C++ applies automatic type conversion to the operand as per certain rules. Similarly, an assignment operator also causes the automatic type conversion.
- The type conversions are automatic as long as the data types involved are built-in types.

- ❑ What happens when they are user defined data types?
- ❑ What if one of the operands is an object and the other is built-in type variable?
- ❑ What if they belong to two different classes?

What is Type Conversions?

- Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types.

- Types of conversions: Three type of situations might arise in the data conversion between incompatible types:
  - **Basic to User-defined**
  - **User-defined to Basic**
  - **One User-defined to Another User-defined**

Basic to User-defined (Class Type)

- In this type of conversion, the source type is basic type and the destination type is class type, i.e. basic data type is converted into the class type.

- The conversion from basic type to the class type can be performed in two ways:

  - Using constructor
  - Using Operator Overloading

Example (Using Constructor)

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

class Distance {
private:
    int feet;
    float inches;
public:
    // Constructor for basic to user-defined conversion

    Distance(int meters) {
        feet = meters * 3.281; // Convert meters to feet
        inches = (meters * 3.281 - feet) * 12;
    }

    void display() {
        cout << feet << " feet " << inches << " inches" << endl;
    }
};
```

```
int main() {
int distance;
cout<<"Enter distance in meters";
cin>>distance;
Distance d = 5; // Implicit conversion from int to Distance

d.display();          // Output: 16 feet 4.12 inches
    return 0;
}
```

During type conversion using the constructor we can pass only one argument and we can do type conversion at the type of initialization only.

Using Operator Overloading

- Type conversion from basic to class type can also be done by operator overloading.
- Assignment operator can be overloaded for this purpose.
- Previous example of *Disptance* class can be rewritten for type conversion using operator overloading concept to overload the assignment operator (=).

Example (Using (=) Operator Overloading)

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

class Distance {
private:
    int feet;
    float inches;
public:
    Distance(int f = 0, float i = 0) : feet(f), inches(i) {}

  Distance& operator=(int meters) {
      // Convert meters to feet and inches
      feet = meters * 3.281;  // 1 meter = 3.281 feet
      inches = (meters * 3.281 - feet) * 12;
      return *this;
  }

void display() const {
      cout << feet << " feet " << inches << " inches" << endl;
    }
};
```

```
int main() {
    Distance d; // Default Distance object
    int meters = 5; // Basic type (meters)

    // Assign the basic type (int) to a user-defined object
using overloaded assignment operator

  d = meters;

    // Display the converted distance
    d.display(); // Output: 16 feet 3.36 inches

    return 0;
}
```

User-defined to Basic Type

- In this type of conversion the source type is class type and the destination type is basic type, i.e. class data type is converted into the basic type.
- The constructor functions do not support this operation.
- It requires special casting operator function.
- The syntax for an overloaded casting operator function, usually referred to as a conversion function, is:

```
operator typename( )
{
   //Function statements
}
```

Continued…

- The conversion function should satisfy the following condition:

- It must be a class member.
- It must not specify the return value.
- It must not have any argument.

Example

```
#include<iostream>
using namespace std; class Time
{
 int hrs, min;
public:
 Time (int ,int); // constructor
  operator int();      // casting operator function
  ~Time()    // destructor
   {
    cout<<"Destructor called..."<<endl;
   }
};
Time::Time (int a,int b)
{
cout<<"Constructor called with two parameters..."<<endl; hrs=a;
min=b;
 }
```



Continued...

```
Time :: operator int()
 {
cout<<"Class Type to Basic Type
Conversion..."<<endl;
return(hrs*60+min);
 }
int main()
{
int h, m, duration;
cout<<"Enter Hours ";
cin>>h;
cout<<"Enter Minutes ";
cin>>m;
```



```
Time t(h,m);    // construct object
duration = t;   // casting conversion
                OR duration = (int)t
```

```
cout<<"Total Minutes are "<<duration;
cout<<"2nd method operator overloading "<<endl;
duration = t.operator int();
 cout<<"Total Minutes are "<<duration;
return 0;
}
```

Example-2

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

class Distance {
private:
    int feet;
    float inches;
public:
    Distance(int f, float i) : feet(f), inches(i) {}
operator int() {       // convert Distance to int
      return feet + static_cast<int>(inches / 12);
}
operator float() {
       float totalFeet = feet + (inches / 12);
       return totalFeet / 3.281; // Convert feet to meters
    }

void display() const {
       cout << feet << " feet " << inches << " inches" << endl;
    }
};
```

```
int main() {
    Distance d(10, 6); // 10 feet 6 inches

    // Implicit conversion from Distance to int
    int totalFeet = d;
    cout << "Total feet (as int): " << totalFeet << endl;

    // Implicit conversion from Distance to float (in meters)
    float totalMeters = d;
    cout << "Total meters (as float): " << totalMeters << "
meters" << endl;

    return 0;
}
```

static_cast<int> is a type of casting operator in C++ that is used to perform explicit type conversions between compatible types. It is a safer and more precise alternative to C-style casting **(type)value** and is part of C++'s **explicit type conversion** mechanisms.

Syntax: **static_cast<new_type>(expression);**

User-defined to User-defined Type

- In this type of conversion both the type that is source type and the destination type are of class type.

- Conversion from one class to another class can be performed either by using the ***constructor*** or ***type conversion function***.

Example

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

class Distance {
private:
    int feet;
    float inches;
public:
    Distance(int f, float i) : feet(f), inches(i) {}
    int getFeet() { return feet; }
    float getInches() { return inches; }
};


 int main() {
     Distance d(10, 6); // 10 feet 6 inches
     MetricDistance md = d;
 // Conversion to MetricDistance
     md.display(); // Output: 3.2 meters
     return 0;
 }
```

```
class MetricDistance {
private:
    float meters;
public:
    // Conversion constructor from Distance to MetricDistance

    MetricDistance(Distance d) {
        meters = (d.getFeet() + d.getInches() / 12) / 3.281;
    }

    void display() {
        cout << meters << " meters" << endl;
    }
};
```

Example

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

class Time{
private:
  int hrs, min;
public:
   Time(int h=0,int m=0): hrs(h), min(m){}
int getMinutes(){
int tot_min = ( hrs * 60 ) + min ;
return tot_min;
 }
 void display() {
 cout<<"Hours: "<<hrs<<endl;
 cout<<" Minutes : "<<min <<endl ;
 }
};
```

```
class Minute{
private:
  int min;
public:
  Minute():min(0){}

  void operator=(Time T) {
    min=T.getMinutes();
  }
    void display()
    {
    cout<<"\n Total Minutes : " <<min<<endl;
    }
};
```

```
int main()
{
Time t1(2,30);
t1.display();
Minute m1;
m1.display();

m1 = t1;

t1.display();
 m1.display();
return 0;
}
```

// conversion from Time to Minute

Type Conversions Summary

Conversion required	Conversion takes place in	
	Source Class	**Destination Class**
Basic to Class	Not Applicable	Constructor
Class to Basic	Casting Operator	Not Applicable
Class to Class	Casting Operator	Constructor

Case Study on Developing an Eco-Friendly Smart Transportation System

The city’s transportation challenge

A growing city needs cleaner, more coordinated journeys.

Congestion

Too many private vehicles increase delays and make journeys unpredictable.

Air pollution

Fuel use and traffic contribute to local pollution and greenhouse-gas emissions.

Fragmented services

Passengers must compare modes, availability and costs across separate services.

Poor visibility

Operators need a shared view of vehicles, bookings and environmental impact.

Case prompt

How can one software system coordinate buses, electric taxis and bicycles while helping passengers make greener choices?

System goals and scope

01 Coordinate transport

Register and manage different transport modes.

02 Support journeys

Estimate fare and emissions for a requested distance.

03 Encourage green choices

Compare options using cost and environmental impact.

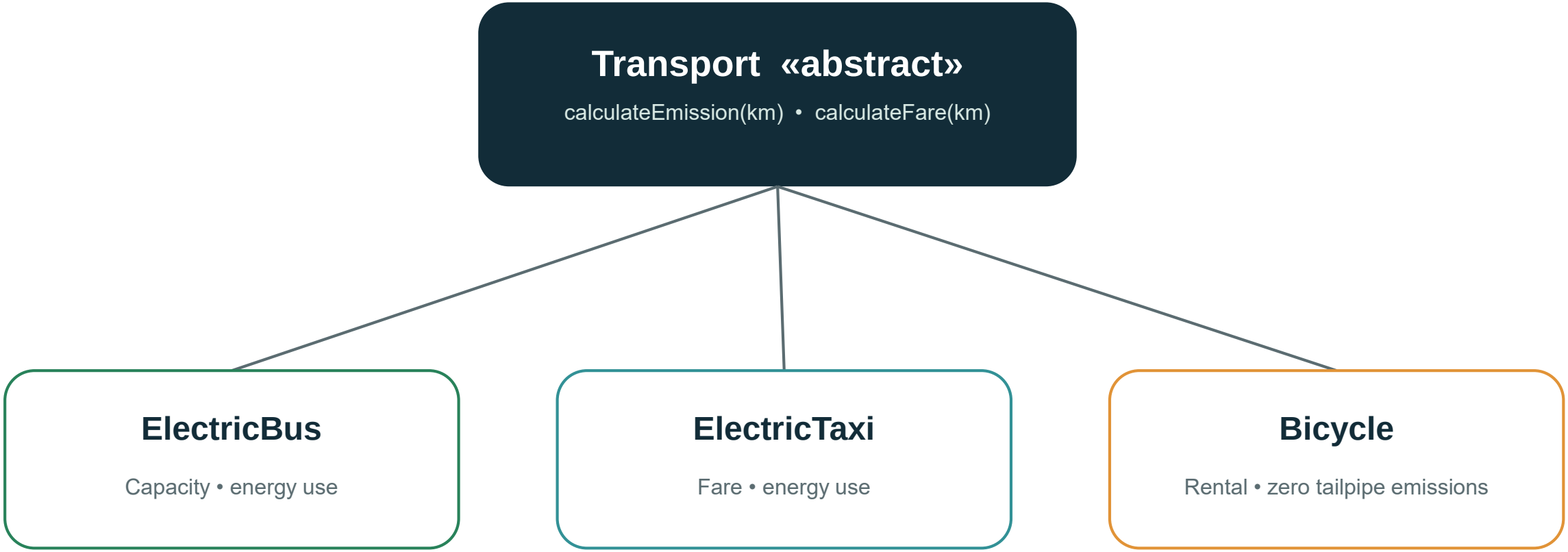
04 Enable extension

Allow new transport types to be added with minimal changes.

Functional requirements

Vehicle management	Register vehicles and display supported modes.
Journey planning	Choose a mode for a trip distance and passenger needs.
Booking	Create a booking and associate it with a selected transport mode.
Fare estimation	Calculate an illustrative fare for a journey.
Emissions estimation	Estimate kg CO_2e using documented coefficients.
Reporting	Compare costs and estimated emissions across transport modes.

Object-oriented model



Additional classes: Passenger • Route • Booking • TransportManager

Where the OOP principles appear

Encapsulation	Keep vehicle state private; expose methods that validate and update it.
Abstraction	Give every transport mode a common interface for fare and emissions.
Inheritance	Specialized transport classes reuse the Transport contract.
Polymorphism	Call the same virtual method on different transport objects.
Composition	A Booking brings together journey details and a selected transport.

C++ design: common transport interface

Pure virtual methods define what every transport mode must provide.

```
class Transport {
public:
    virtual double calculateEmission(double km) const = 0;
    virtual double calculateFare(double km) const = 0;
    virtual string name() const = 0;
    virtual ~Transport() = default;
};
```

Why this works

- Common interface
- Derived classes implement details
- New modes can be added without rewriting client code

Include <string> and <memory> (and use std::string) in the complete program.

C++ design: specialized transport modes


```
class ElectricBus : public Transport {
public:
    double calculateEmission(double km) const override {
        return km * 0.04;  // illustrative kg CO2e/km
    }
    double calculateFare(double km) const override {
        return 10.0 + km * 1.5;
    }
    string name() const override { return "Electric Bus"; }
};

class Bicycle : public Transport {
public:
    double calculateEmission(double) const override {
        return 0.0;  // tailpipe emissions only
    }
    double calculateFare(double km) const override {
        return km * 2.0;
    }
    string name() const override { return "Bicycle"; }
};
```

Override

Each subclass provides its own calculation.

Assumptions

Replace example coefficients with measured, documented values.

Runtime polymorphism in action


```
vector<unique_ptr<Transport>> options;
options.push_back(make_unique<ElectricBus>());
options.push_back(make_unique<ElectricTaxi>());
options.push_back(make_unique<Bicycle>());

double distance = 5.0;
for (const auto& vehicle : options) {
    cout << vehicle->name()
         << " | Fare: " << vehicle->calculateFare(distance)
         << " | Emissions: "
         << vehicle->calculateEmission(distance) << " kg CO2e\n";
}
```

What to notice

- All modes are stored through Transport pointers.
- The correct overridden method runs for each object.
- A new mode can be inserted without changing the loop.

Requires #include <memory>, <vector>, <iostream>, and <string>.

Illustrative 5 km journey comparison

Transport mode	Fare (currency units)	Estimated emissions
Electric Bus	17.50	0.20 kg CO_2e
Electric Taxi	70.00	0.30 kg CO_2e
Bicycle	10.00	0.00 kg tailpipe CO_2e

Interpret carefully

Values follow the illustrative coefficients and fare formulas in the code.

Real-world extension

Account for grid electricity, occupancy, lifecycle impacts and actual fares.

From prototype to smart mobility platform

Booking	Passenger, origin, destination, selected mode
Route	Distance, estimated duration, traffic conditions
TransportManager	Availability, recommendations, allocation
SustainabilityReport	Cost, energy, estimated emissions, comparisons

Potential integrations: GPS • traffic APIs • charging-station data • public transit schedules

Activities and assessment

01	**Extend the hierarchy**	Add ConventionalBus and document its emission assumptions.
02	**Implement booking**	Create a Booking class with passenger, distance and selected mode.
03	**Recommend greener travel**	Select the lowest-emission available mode under a fare limit.
04	**Explain design choices**	Show how polymorphism simplifies adding a new transport class.
05	**Evaluate sustainability**	Explain why zero tailpipe emissions do not mean zero lifecycle impact.

KEY TAKEAWAYS

Model the domain

Turn real entities and responsibilities into cohesive classes.

Program to an interface

Use a shared Transport contract and runtime polymorphism.

Measure responsibly

Treat emissions as estimates and document assumptions.

Design sustainably. Code modularly. Evaluate transparently.

Thanks