Practical Lecture : Polymorphism

Quick Recap

Let’s take a quick recap of previous lecture –

- Dynamic memory allocation using new and delete operators
- Memory leak and allocation failures
- Dangling, void, null , Wild pointer

Today's Agenda

Today we are going to cover –

- Compile and run time polymorphism
- Virtual functions, Pure virtual functions
- virtual destructor
- Abstract classes and concrete class
- Self-Referential class
- Early binding and late binding, Dynamic constructors.

Let’s Get Started-

Polymorphism

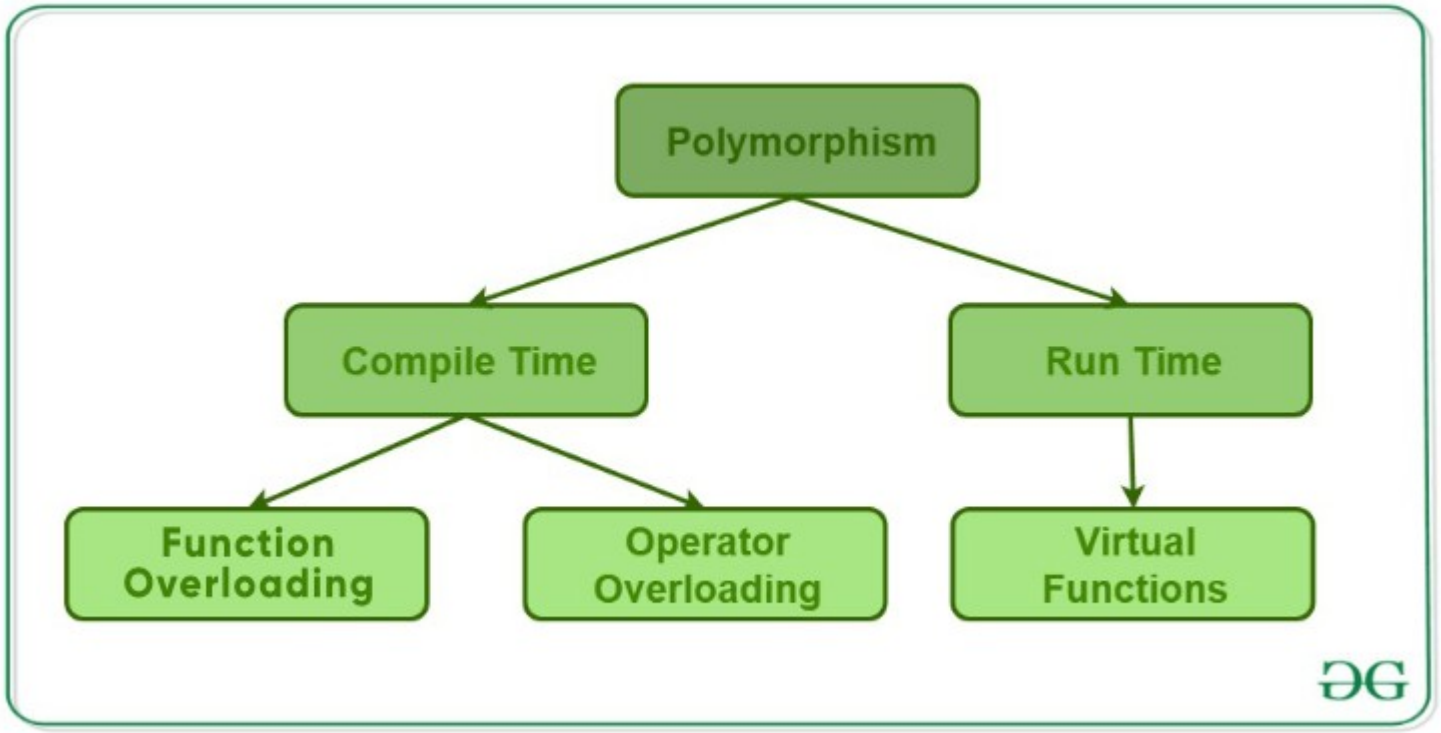
- Crucial feature of OOP

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

- A real-life example of polymorphism, a person at the same time can have different characteristics. Like a person (or student) at the same time is a son/daughter, a student, a friend, a brother/sister, an employee etc. So the same person posses different behavior in different situations. This is called polymorphism.

- One name, many forms.

Types of polymorphism



Compile time polymorphism

Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

```
Class A{
void func(int); //assume implementation of these overloaded functions
void func(double);
void func(int, float);
};
int main() {
        A obja;
        obja.func(7);//These functions behave differently in different situations
        obja.func(8.345);
        obja.func(9, 5.76);
}
```

Compile time polymorphism

Operator Overloading:

C++ also provide option to overload operators.

For example, we can make the operator (‘+’) for string class to concatenate two strings.

We know that this is the addition operator whose task is to add two operands.

So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    void print() { cout << real << " + i" << imag << endl; }
```

Compile time polymorphism

```
  // This is automatically called when '+' is used with between two Complex objects
   Complex operator + (Complex const &obj) {
       Complex res;
       res.real = real + obj.real;
       res.imag = imag + obj.imag;
       return res;
   }
int main() {
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
}
```

The operator ‘+’ is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

Runtime polymorphism

This type of polymorphism is achieved by Function Overriding.

Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Typically, Run time polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

Run time polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function

Always implemented using pointers and virtual function .

Practice question

```
using namespace std;
 class base
 {
 public:
    void print ()
    {
       cout<< "print base class" <<endl;
   }
     void show ()
    {
       cout<< "show base class" <<endl;
    }
};
```

Practice question

```
class derived:public base
{
public:
    void print ()     {
       cout<< "print derived class" <<endl;
    }

    void show ()
    {
       cout<< "show derived class" <<endl;
    }
};
```

Practice question

```
//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    bptr->print();

    bptr->show();

    return 0;
}
```

Practice question

Output:
print base class
show base class

The reason for the this output is that the call of the functions print() and show() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the print() and show() functions is set during the compilation of the program.

Now let us change the program by making use of virtual function

Practice question

```
using namespace std;
 class base
 {
 public:
    virtual void print ()
    {
       cout<< "print base class" <<endl;
   }
     void show ()
    {
       cout<< "show base class" <<endl;
    }
};
```

Practice question

```
class derived:public base
{
public:
    void print ()    //print() is already a virtual function in base class
 {
       cout<< "print derived class" <<endl;
    }

    void show ()
    {
       cout<< "show derived class" <<endl;
    }
};
```

Practice question

```
//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
  //virtual function, binded at runtime (Runtime polymorphism)
   bptr->print();
// Non-virtual function, binded at compile time
  bptr->show();
    return 0;
}
```

Practice question

Output:
print derived class
show base class

This time, the compiler looks at the contents of the pointer instead of it's type. In earlier case, compiler was looking at only the type of pointer, which was base class pointer. So though it was storing object of derived class, it was calling base class member function.

Virtual function / late binding

A virtual function is a member function which is declared in the base class using the keyword virtual and is re-defined (Overriden) by the derived class.

Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

This sort of operation is referred to as dynamic linkage, or late binding.

The main thing to note about the program is that the derived class's function is called using a base class pointer.

The idea is that virtual functions are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference.

In other words, virtual functions are resolved late, at runtime.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation, we only declare it.

A pure virtual function is declared by assigning 0 in the declaration.

```
virtual int area() = 0;
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Pure Virtual function

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};

//class Shape is abstract class here
```

Abstract class

Classes that contain at least one pure virtual function are known as abstract base classes.

Abstract base classes cannot be used to instantiate objects

If derived class do not redefine virtual function of base class, then derived class also becomes abstract just like the base class.

It is the responsibility of the all further derived classes to provide the definition to the pure virtual member function.

But an abstract base class is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities as shown in example in slide 15.

And can actually be dereferenced when pointing to objects of derived (non-abstract) classes.

Concrete class

- An abstract class is a class for which one or more methods are declared but not defined, meaning that the compiler knows these methods are part of the class, but not what code to execute for that method. These are called abstract methods. Here is an example of an abstract class.

```
class shape {
public:
virtual void draw() = 0;
};
```

- To be able to actually use the draw method you would need to derive classes from this abstract class, which do implement the draw method, making the classes concrete.
- A class that has any abstract methods is abstract, any class that doesn't is concrete.
- It's just a way to differentiate the two types of classes.
- Every class is either abstract or concrete. A base class can be either abstract or concrete and a derived class can be either abstract or concrete:

Abstract versus concrete class

Abstract class can not be used to create an object. Whereas, concrete class can be used to create an object.
In other words, an abstract class can't be instantiated. Whereas, a concrete one can.

Concrete means “existing in reality or in real experience; perceptible by the senses; real''. Whereas, abstract means 'not applied or practical; theoretical'.

An abstract class is one that has one or more pure virtual function. Whereas a concrete class has no pure virtual functions.

An abstract class serves as "blueprint" for derived classes, ones that can be instantiated.

E.g. Car class (abstract) whilst Audi S4 class (deriving from Car) class is a concrete implementation.

Virtual destructor

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior.

To correct this situation, the base class should be defined with a virtual destructor.

```
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};
```

Virtual destructor

```
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
   return 0;
}
```

Virtual destructor

Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following:

Constructing base
Constructing derived
Destructing base

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example:

```
#include<iostream>
using namespace std;
 class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
```

Virtual destructor

```
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

Virtual destructor

Output:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Self referential classes

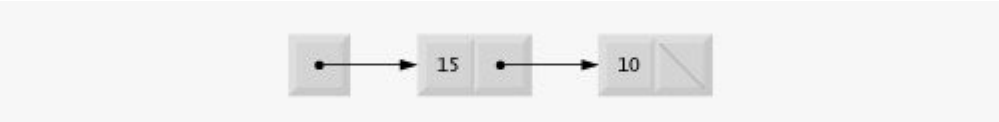
It is a special type of class.
It is basically created for linked list and tree based implementation in C++.
If a class contains the data member as pointer to object of similar class, then it is called a self-referential class. Eg.

```
class node
{
        private:
                int data;
                node * next;  //pointer to object of same type

        public:
                //Member functions.
};
```

node *next; represents the self-referential class declaration, node is the name of same class and next is the pointer to class (object of class).

Self referential classes



Self-referential class objects linked together.

Self-referential objects can be linked together to form useful data structures, such as linked lists, queues, stacks and trees.

Dynamic constructors

When allocation of memory is done dynamically using dynamic memory allocator “new” in a constructor, it is known as dynamic constructor.
By using this, we can dynamically initialize the objects.

```
#include <iostream>
using namespace std;
 class A{
    int* p;
 public:
    // default constructor
    A()
    {
         // allocating memory at run time
        p = new int;
        *p = 0;
    }
```

Dynamic constructors

```
// parameterized constructor
    A(int x)
    {
        p = new int;
        *p = x;
    }
    void display()
    {
        cout << *p << endl;
    }
};
```

Dynamic constructors

```
int main()
{

    // default constructor would be called
    A obj1 = A();
    obj1.display();

    // parameterized constructor would be called
    A obj2 = A(7);
    obj2.display();
}
```

Dynamic constructors

In this integer type pointer variable is declared in class which is assigned memory dynamically when the constructor is called.
When we create object obj1, the default constructor is called and memory is assigned dynamically to pointer type variable and initialized with value 0.
And similarly when obj2 is created parameterized constructor is called and memory is assigned dynamically.
Creating and maintaining dynamic data structures requires dynamic memory allocation
The new operator is essential to dynamic memory allocation.
Operator new takes as an operand the type of the object being dynamically allocated and returns a reference to an object of that type.
For example, the statement
Node nodeToAdd = new Node(10);
allocates the appropriate amount of memory to store a Node and stores a reference to this object in nodeToAdd.

MCQ

Which of the following in not true about virtual function and pure virtual function?

1. Both are members of base class and redefined by derived class
2. Base class having virtual function can’t be instantiated whereas the one with pure virtual function can be.
3. Virtual void show()=0; is a definition of pure virtual function
4. Classes with pure virtual function are known as abstract class

MCQ

Which of the following in not true about virtual function and pure virtual function?

1. Both are members of base class and redefined by derived class
2. Base class having virtual function can’t be instantiated whereas the one with pure virtual function can be.
3. Virtual void show()=0; is a definition of pure virtual function
4. Classes with pure virtual function are known as abstract class

MCQ

Which of the following in not correct about abstract and concrete classes?

1. Abstract class is the class with pure virtual function
2. Concrete class is the derived class with implementation of pure virtual method
3. Concrete class cannot be instantiated
4. Abstract class cannot be instantiated

MCQ

Which of the following in not correct about abstract and concrete classes?

1. Abstract class is the class with pure virtual function
2. Concrete class is the derived class with implementation of pure virtual method
3. Concrete class cannot be instantiated
4. Abstract class cannot be instantiated

MCQ

Choose an incorrect option.

Run time polymorphism is achieved using

1. pointers
2. Virtual function
3. Function overriding
4. Operator overloading

MCQ

Choose an incorrect option.

Run time polymorphism is achieved using

1. pointers
2. Virtual function
3. Function overriding
4. Operator overloading

MCQ

Choose an incorrect option.

1. compile time polymorphism is also called static binding
2. Run time polymorphism is also known as late binding
3. Function overriding is an example of run time polymorphism
4. Run time Polymorphism is always implemented using inheritance

MCQ

Choose an incorrect option.

1. compile time polymorphism is also called static binding
2. Run time polymorphism is also known as late binding
3. Function overriding is an example of run time polymorphism
4. Run time Polymorphism is always implemented using inheritance

MCQ

Which of the following is incorrect?

A. Making base class destructor virtual guarantees that the object of derived class is destructed properly
B. An abstract class and concrete class has one pure virtual function
C. Dynamic constructor allocates memory dynamically using “new” in a constructor.
D. Self referential classes are used to create dynamic data structures likes stacks and queues.

MCQ

Which of the following is incorrect?

A. Making base class destructor virtual guarantees that the object of derived class is destructed properly
B. An abstract class and concrete class has one pure virtual function
C. Dynamic constructor allocates memory dynamically using “new” in a constructor.
D. Self referential classes are used to create dynamic data structures likes stacks and queues.

Assignment

Write a C++ program to create a class Shape with length and width as data members. Have pure virtual method print_area() in base class. Derive a class called rectangle which will implement the method and calculate and print the area of a rectangle. Implement the above program using runtime polymorphism

Assignment

Create a class called Player with name of the player as data member and getdata(), displaydata() as member functions. Player class is further inherited by two classes- CricketPlayer and FootballPlayer. CricketPlayer has getRuns() method to get the runs scored by player andFootballPlayer has getGoals() method to get goals of the player. Make displaydata() function as virtual in base class and overload it in derived classes to display name and run/goals of respective players. Write a COMPLETE C++ program to achieve runtime polymorphism in the above example.

QNA Time

Any Questions ??

Any Questions??

Thank You!

See you guys in next class.