Data Structures and Object Oriented Programming

Lecture 21

Dr. Naveed Anwar Bhatti

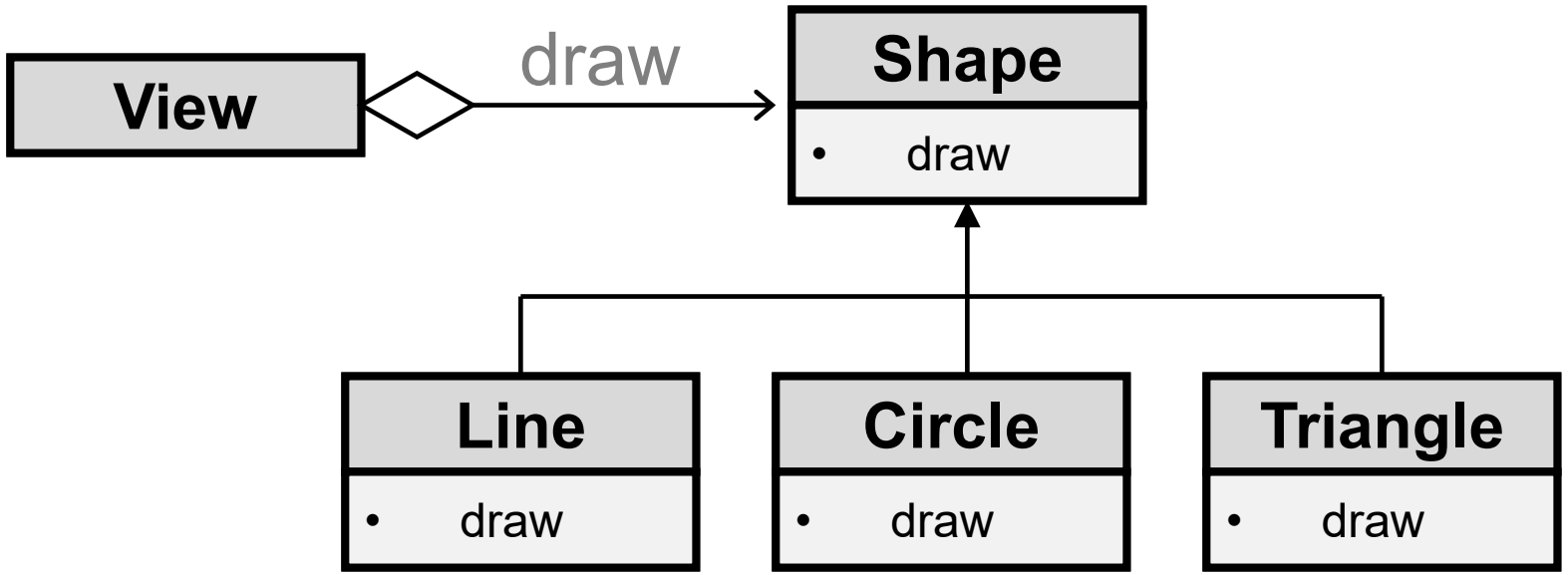
Webpage: naveedanwarbhatti.github.io

Polymorphism

& Related Concepts

Polymorphism Revisited

- In OO model, polymorphism means that different objects can behave in different ways for the same message (stimulus)
- Consequently, sender of a message does not need to know the exact class of receiver



Polymorphism

Technical Definition

" Polymorphism can be achieved through **Dynamic binding**. Dynamic binding is enabled when a **virtual function** is invoked through a **derived class object which is referred indirectly by either a base class pointer or reference** "

Pointers to the base class of derived objects

Pointers to the base class of derived objects

- Inheritance implies an **is-a relationship** between two classes
- When we create a Derived object, it contains a Base part (which is constructed first), and a Derived part (which is constructed second)
- C++ allow to set a Base pointer or reference to a Derived object.

```
class Parent {
public:
void Hello() { cout << "Hi from a Parent!\n"; }
// ...
};

class Child : public Parent {
public:
void Hello() { cout << "Hi from a Child!\n"; }
};

int main() {
    Parent* P;
    Child C;
    P = &C;
    P->Hello();
}
```

Note: **P** can only calls those members which are part of **Parent** class

Static binding vs. Dynamic Binding

Static binding (Compile-time binding)

- Static binding is to associate a function's name with the entry point of the function at compile time.

- Example:

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

void sayHi()
{
    cout << "Hello, World!\n";
}
int main()
{
    sayHi(); // the compiler binds any invocation of sayHi()
             // to sayHi()'s entry point.
}
```

In C++, all **non-virtual** functions are bound at compile-time (static). However, **virtual functions** can have **dynamic binding**

Dynamic binding (Run-time binding)

- Run-time binding is to associate a function's name with the entry point at run-time.

- C++ supports run-time binding through **virtual functions**.

Virtual Function

- A virtual function is a member function that you expect to be **redefined** in **derived classes**.

- When you refer to a derived class object using a pointer to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

```
class Parent {
public:
virtual void Hello() { cout << "Hi from a
Parent!\n"; }
// ...
};

class Child : public Parent {
public:
virtual void Hello() { cout << "Hi from a
Child!\n"; }
};

int main() {
    Parent* P;
    Child C;
    P = &C;
    P->Hello();
}
```

Virtual Functions

- To declare a function virtual, we use the Keyword virtual.

```
class Parent {
public:
virtual void sayHi() { cout << "Just hi!\n"; }
};
```

- If the member function definition is outside the class, the keyword virtual must not be specified again.

```
class Parent {
public:
virtual void sayHi();
};
virtual void Parent::sayHi() { // error
cout << "Just hi!\n";
}
```

- Virtual functions can not be stand-alone functions or static methods.

Virtual Functions

- A virtual function can be used same as non-virtual member functions.

```
class Parent {
public:
    virtual void print() { cout << "Hello!\n"; }
};

int main() {
    Parent obj;
    obj.print();
}
```

- A virtual function can be inherited from a base class by a derived class, like other class member functions.

```
class Parent {
public:
    virtual void print() { cout << "Hello!\n";
    }
};
class Child : public Parent {};
```

```
int main() {
    Child obj;
    obj.print();
}
```

Virtual Function

- To let derived classes have their own implementation for the virtual function, we **override** base class virtual functions in derived class.
- In order for a derived class virtual function instance to override the base class virtual function instance, **its signature must match** the base class virtual function exactly.
- The use of keyword virtual is **optional in derived classes**.

```
class Parent {
public:
virtual void sayHi() { cout << "Just hi!\n"; }
};
```

```
class Child : public Parent {
public:
// overrides Shape::sayHi(), automatically virtual
void sayHi() { cout << "Hi from a Child!\n"; }
};
```

Example

```
class Parent
{
public:
    virtual void sayHi() { cout << "Just hi!\n"; }
};

class Child_1 : public Parent
{
public:
    virtual void sayHi() { cout << "Hi from a
    Child_1!\n"; }
};

class Child_2 : public Parent
{
public:
    virtual void sayHi() { cout << "Hi from a
    Child_2!\n"; }
};
```

```
int main() {
    Parent* p;
    int choice;
    cout << "1 -parent, 2 – child_1, 3 –child_2\n ";
    cin >> choice;
    switch (choice)
    {
        case 1: p = new Parent; break;
        case 2: p = new Child_1; break;
        case 3: p = new Child_2; break;
    }
    p->sayHi(); // dynamic binding of sayHi()
    delete p;
}
```

Example

```
class Parent
{
public:
    virtual void sayHi() { cout << "Just hi!\n"; }
};

class Child_1
{
public:
    virtual vo
    Child_1!\n
};

class Child_2 :
{
public:
    virtual void sayHi() { cout << "Hi from a
    Child_2!\n"; }
};
```

```
int main() {
    Parent* p;
    int which;
                                    -rectangle\n ";
                                    sayHi()
    delete p;
}
```

Polymorphism is thus implemented by virtual functions and run-time binding mechanism in C++. A class is called **polymorphic** if it contains virtual functions.

Polymorphism

A typical scenario of polymorphism in C++:

- There is an inheritance hierarchy
- The first class that defines a virtual function is the base class
- Each of the derived classes in the hierarchy must have a virtual function with same signature.
- There is a pointer of base class type; and this pointer is used to invoke virtual functions of derived class.

Virtual Tables

C++ uses the virtual table (vtable) mechanism to implement the dynamic binding of virtual functions.

- A class with virtual member functions has a virtual table which contains the address of its virtual functions.
- An object of such a class has a pointer(vptr) to point to the virtual table of the class.
- Dynamic binding is done by looking up the virtual table for the entry point of the appropriate function at run-time.

Virtual Functions

Constructors and Destructors

- A constructor cannot be virtual since it is used to construct an object.
- A destructor can be virtual. Virtual destructors are very useful when some derived classes have cleanup code.
- Example

```
class B {
public:
    virtual B(); // error
    virtual ~B(); // ok
    virtual void f(); // ok
};
```

Dynamic v.s. Static binding

The approach of using inheritance and run-time binding facilitates the following software quality factors:

- Reuse
- Transparent extensibility
- Delaying decisions until run-time
- Architectural simplicity

Compared to compile time binding, run time binding has overhead in terms of space and time.

- Extra space is needed for virtual table.
- Extra time for virtual table lookup is required at each polymorfic function call.

Dynamic v.s. Static binding

When to choose use different kinds of bindings:

- Use compile-time binding when you are sure that any derived class will **not want to override the function dynamically.**
- Use run-time binding when the derived class may be able to provide a different implementation **that should be selected at run-time.**

Pure Virtual Function

- A pure virtual function is a virtual function in base class that has no definition. It is declared using specifier “= 0 ”.

```
class B {
public:
    virtual void setX() = 0; //virtual
};
```

- Only a virtual member function can be pure.

```
void f() = 0; //error! f() is a stand alone
function
class B {
public:
void setX() = 0; //error! setX not virtual
// ...
};
```

- Declaring a virtual function pure is not the same as defining a virtual function with an empty body.

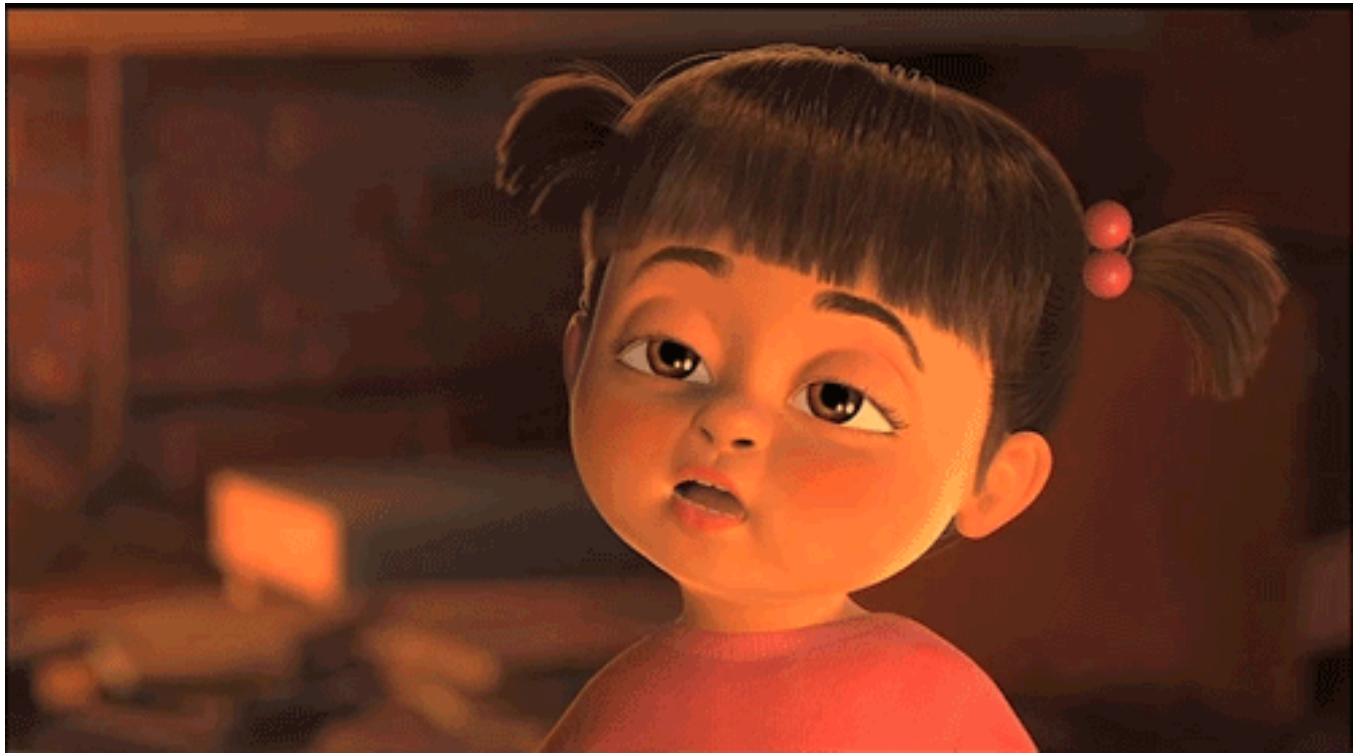
Abstract Class

- A class that has a pure virtual function is an abstract class. Abstract class is used as an interface for its derived classes.

- If a class derived from an abstract class, and this class doesn’t override all the pure virtual function in the base class, then this class is also an abstract class.

- No object can be created for an abstract class ! body.

Thanks a lot



If you are taking a Nap, **wake up**.....Lecture Over