Week 11

Runtime Polymorphism

Runtime Polymorphism

- C++ Runtime 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

- Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.

Polymorphism Again

- In OO model, polymorphism means that different objects can behave in different ways for the same message.

- Sender of a message does not need to know the exact class of receiver

- It doesn't say "If you're a **Triangle**, do this, if you're a **Circle**, do that, and so on." If we write that kind of code, which checks for all the possible types of a **Shape**, it will soon become a messy code, and we need to change it every time we add a new kind of **Shape**. Here, however, we just say "You're a **Shape**, I know you can move(), draw(), and shrink( ) yourself, do it, and take care of the details correctly."

Binding

- Connecting a function call to a function body is called *binding*. Decide which definition to use based on calling object
  - Early Binding/Compile-time Binding / Static Binding
  - Late -Binding/Runtime Binding/ Dynamic Binding

Static vs Dynamic Binding

- Early binding/ Static binding means that target function for a call is selected at compile time. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed.

- Late binding/Dynamic binding means that target function for a call is selected at run time

Upcasting / Downcasting

- **Upcasting:**

  C++allows that a derived class pointer (or reference) can be treated as base class pointer. This is **upcasting**.

  Parent* p = new Child(); //Implicit casting

- **Downcasting**

  is an opposite process, which is converting base class pointer (or reference) into derived class pointer.

  Child* c= (child*)new Parent(); //Needs to type-casting explicitly

Virtual Functions

- Target of a virtual function call is determined at run-time
- Defining a virtual function in a base class, with another version in a derived class, signals to the compiler that we don't want static linkage for this function rather we do want the selection of the function to be called at runtime.
- This sort of operation is referred to as **dynamic linkage**, or **late binding**.

```
class Shape {
      …
      virtual void draw();
  }
```

Example

```
class beverages{
public:
    void addHotItems(){
            addHotWater();
            addMilk();
    }
 virtual void addSachets ()
    {
            cout<<"Sachets of beverage"<<endl;
    }
```

```
private:
   void addHotWater(){
         cout<<"\tAdd Hot Water "<<"\n";
   }
   void addMilk(){
         cout<<"\tAdd Milk "<<"\n";
   }
};
```

```
class Tea:public beverages
{
public:
    void addSachets ()
    {
        cout<<"\tAdd Tea "<<"\n";
    }
    Tea(){
        cout<<"Preparing Tea... "<<"\n";
    }
};
```

```
class Coffee :public beverages
{
public:
    void addSachets (){
        cout<<"\tAdd Coffee "<<"\n";
    }
    Coffee(){
        cout<<"Preparing Coffee... "<<"\n"; } };
```

```cpp
int main() {
	beverages *tea = new Tea(); //Upcasting
	tea->addHotItems();
	tea->addSachets();
	beverages *coffee = new Coffee();
	coffee->addHotItems();
	coffee->addSachets();
	return 0;
}
```

Static vs Dynamic Binding

```
Line _line;
_line.draw();      // Always Line::draw
                   // called
Shape* _shape = new Line();
_shape->draw(); // Shape::draw called
        // if draw() is not virtual

Shape* _shape = new Line();
_shape->draw(); // Line::draw called
            // if draw() is virtual
```

Abstract Classes Vs Concrete Class

Abstract Classes

- a class that contains at least one pure virtual function or abstract function.
- a virtual functions with no implementation is pure virtual function.
- The class may also contain non-virtual functions and member variables.

- Pure virtual functions implemented in derived class.
- We cannot create an object of an abstract class
- But we create a pointer. And, in this pointer, we assign the object of derived classes.

Concrete Classes

- Conversely, a class with no pure virtual function is a concrete class !

Pure Virtual Functions

- A pure virtual represents an abstract behavior and therefore may not have its implementation (body)

- A function is declared pure virtual by following its header with “= 0”

```
virtual void draw() = 0;
```

… Pure Virtual Functions

- A class having pure virtual function(s) becomes abstract

```
class Shape {
  …
public:
  virtual void draw() = 0;
}
  …
Shape s;      // Error!
```

… Pure Virtual Functions

- A derived class of an abstract class remains abstract until it provides implementation for all pure virtual functions

Example- Abstract Class

```
class AbstractBase
{
 public:
 virtual void Display() = 0; //Pure Virtual
   Function declaration
};
```

```
class Derived:public AbstractBase
{
 public:
 void Display()
 {
    cout << "pure virtual function
   implementation\n"; }
 };
```

```
int main() {
 AbstractBase *basePointer = new Derived();
 basePointer->Display();
 AbstractBase * bPtr;
 Derived dObj;
 bPtr = &dObj;
 bPtr->Display();
}
```

Virtual Destructors

```
class Base
{
 public:
 ~Base() {
cout << "Base Destructor\t"; }
};

class Derived: public Base
{
 public:
 ~Derived() {
cout<< "Derived Destructor"; }
};
```

```
int main()
{
 Base* b = new Derived;
//Upcasting
 delete b;
}
```

Output :

Base Destructor

Need for Virtual Destructor

- In the above example, **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak.

```
class Base
{
 public:
 virtual ~Base() {
cout << "Base Destructor\t";
}
};

class Derived:public Base
{
 public:
 ~Derived() {
cout<< "Derived
Destructor"; }
};
```

```
int main()
{
 Base* b = new Derived;
//Upcasting
 delete b;
}
```

Output :

Derived Destructor

Base Destructor

Avoiding Memory Leak – Virtual Destructor

- When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behavior.

Review