Lecture 10: Polymorphism

Introduction

Polymorphism is the final fundamental concept in object-oriented programming that allows a single interface to represent different types. It enables code flexibility and reusability by allowing objects to behave differently based on their context, which is achieved using virtual functions.

Virtual Methods

Method overriding allows a derived class to provide a new implementation of an inherited method. However, if a derived class object is assigned to a base class object, pointer or reference, the base class implementation is used; any derived class-specific attributes or behaviors are lost. This process is called object slicing. To prevent this, enable dynamic dispatch by quantifying the base class method with the '`virtual`' keyword. This ensures that when a base class pointer or reference is used, the overridden method in the derived class is executed via the *virtual table* (vtable). The syntax for declaring a virtual method is:

`virtual` *`method-definition`*

An overridden method of a virtual method remains virtual. To explicitly indicate that a derived class method overrides a virtual method, use the '`override`' keyword in its definition:

`method-header` `override` {*`body`*}

And to prohibit further overrides, append the '`final`' keyword to its definition:

`method-header` `final` {*`body`*}

Example:

```
class A
{
  public:
  virtual string id() const {return "classA";}
  int value() const {return 21;}
};

class B : public A
{
  public:
  //overrides a virtual method but prevents further overriding
  string id() const override final {return "classB";}
  int value() const {return 13;}
};

std::string F(const A& obj)
{
  return obj.id() + std::to_string(obj.value()) + '\n';
}

int main()
{
  A a;
  B b;
  std::cout << F(a); //classA21
  std::cout << F(b); //classB21
  return 0;
}
```

Abstract Classes & Interfaces

A *pure virtual function* (also known as an *abstract function*) is a virtual function that acts as a template for derived classes. It forces subclasses to provide their implementation. The syntax for declaring a pure virtual function is:

```
virtual function-header = 0;
```

A class containing at least one pure virtual function is called an *abstract class*. Abstract classes cannot be instantiated directly—only pointers and references to them can be created. A derived class is abstract if all inherited pure virtual functions are not implemented.

Pure virtual methods can be overridden in derived classes regardless of their access specifier. An abstract class containing only pure virtual methods is called an *interface*. Inheriting multiple interfaces with at most one concrete or abstract class does not cause conflicts.

Example:

```
//Abstract Class
class RegularPolygon
{
  private:
  double len;
  virtual int sides() const = 0; //pure virtual method
  public:
  RegularPolygon() : len(1.0) {}
  double length() const {return len;}
  void length(double value) {if(value > 0) {len = value;}}
  virtual double area() const = 0; //pure virtual method
  double perimeter() const {return sides() * len;}
  virtual ~RegularPolygon() {}
};

//Concrete Class
class Square : public RegularPolygon
{
  public:
  Square(){}
  Square(double value) {length(value);}
  int sides() const override {return 4;}
  double area() const override {return pow(length(),2);}
};

//Abstract Class - fail to override sides()
class RegularTriangle : public RegularPolygon
{
  public:
  RegularTriangle(){}
  RegularTriangle(double value) {length(value);}
  double area() const override {return pow(length(),2) * sqrt(3);}
};
```

In the above example, *sides()* is private whenever a *RegularPolygon* pointer or reference is used, but public for *Square* objects.

Virtual Destructor

If a base class contains at least one virtual function, its destructor should also be virtual. This ensures that when a derived object is deleted through a base class pointer, its destructor is correctly invoked. Only the base class destructor runs without a virtual destructor, leading to potential memory leaks if the derived class dynamically allocates resources.

Example:

```
class A
{
  public:
  ~A() {std::cout << "A deleted";}
};

class B
{
  public:
  virtual ~B() {std::cout << "B deleted";}
};

class C : public A, public B
{
  public:
  ~C() {std::cout << "C deleted";}
};

int main()
{
  A* a = new C;
  B* b = new C;
  delete a; //A deleted
  delete b; //C deleted B deleted A deleted
  return 0;
}
```