References

Chapter 15 – Inheritance, Polymorphism, and Virtual Functions

Starting Out With C++. From Control Structures through Objects (Eighth Edition)

Interview Questions:

- What are the differences among **overloading**, **overriding** and **redefining**?
- (1) What is virtual function? (2) How do you use virtual functions? (3) If you were the designer of C++, how do you implement a virtual function?

Polymorphism in C++

Encapsulation, **Inheritance** and **Polymorphism** are the three characteristics of Object-Oriented Programming. Polymorphism is considered as one of the most important features of Object Oriented Programming, since it enables coding to be written in an easier and much more elegant manner. The word polymorphism means having multiple forms. It can be defined as the ability to perform 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, and an employee. So the person may have different behavior in different situations. This is called polymorphism.

Story

During an interview, the interviewer asked John: “So how do you describe yourself?”

John said: “Well, I am OOP person.”

“What does that mean?”

“Oh! I mean that I have the three characteristics of OOP.”

“This is the most creative description which I even heard. Please explain.”

"I **inherit** the good genes from my parents, so that I am proactive and productive. Secondly, I am rather **polymorphic** since I am not just a C++ programmer, but also a guitar and tennis player. On the top of that, I am **encapsulated** by family trusts and my finance is well-protected."

"No wonder you are such a …"

Two types of Polymorphisms

In C++ polymorphism is mainly divided into two types:

- **Runtime Polymorphism** - This type of polymorphism is achieved by **function overriding**.
- **Compile time Polymorphism** - This type of polymorphism is achieved by (1) **template** or (2) **function overloading** or **operator overloading**.

Overloading, Redefining and Overriding

Let us first clarify the differences among **Overloading, Redefining, and Overriding**, before looking into these two types of Polymorphisms.

Overloading	Horizontal relationship: the involved functions are in the same class.	An *overloaded* function is a function that shares its name with one or more other functions in **the same class**, but which has a different parameter list. The compiler chooses which function is desired based upon the arguments used.
Redefining	Vertical relationship: the involved functions are in	A *redefined* function is a method in a derived class that has a different definition than **a *non*-virtual function** in an ancestor class. Don't do this. Since the

	the parent and child.	method is not virtual, the compiler chooses which function to call based upon the static type of the **object reference** rather than the actual type of the object.
Overriding		An *overridden* function is a method in a descendant class that has a different definition (implementation) than **a virtual function** in an ancestor class. The compiler chooses which function is desired based upon the type of the **object** being used to call the function.

.

	Overloading	**Redefining**	**Overriding**
Signatures	Different	Same	Same
Relationship	Same class	Base class and derived class	Base class and derived class
Binding	Static (compile time)	Static (compile time)	Dynamic (run time)
Keyword	none	none	virtual
Is member functions selected based on object or object reference?	N/A	**Object reference**	**Object**

Runtime Polymorphism

Runtime Polymorphism is only possible by using virtual function or pure virtual function.

Virtual member functions: They are the functions in base class that expect to be redefined in derived class. Without virtual member functions, C++ uses **static**

binding. (The compiler will not bind the function to calls in compile time). With virtual functions, C++ uses **dynamic binding.** (The compiler will not bind the function to calls in compile time. Instead, the program will bind them at runtime.)

Exercise 1 (Sample_Poly1.txt)

Step 1:

Predict the output and run the program to check your prediction.

Step 2:

Add another class C, which inherits from B. Modify main() to see how the print() work. When aPtr points to a C object, will the print() of B or C be executed?

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

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

class derived:public base
{
public:
   void print () //print () is already virtual function in derived class, we could
also declared as virtual void print () explicitly
   { cout<< "print derived class" <<endl; }

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

//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;
}
```

Exercise 2: (Sample_Timer.txt)

Step 1: Draw the class hierarchy, list their data members and member functions, and explain the purposes that these classes are implemented for. (The class hierarchy is the big picture of the entire system, which will help us understand and trace the program. The drawing will help you solve the problems below. You do not have to present it if this exercise is given in a quiz or homework.)

Step 2: We purposely left a bug in the program which causes an infinite looping. Fix the infinite looping problem.

Step 3: Create a new class named ExpoCounter. With the following code in main(),

```
ExpoCounter ec1(100);
ec1.Go();
```

the screen displays:

1*4*9*16*25*36*49*64*81*ding!

```
#include <iostream>

using namespace std;

class Timer
{
protected:
	int setting;
	int current;
public:
	Timer(int s){
		setting = s;
		current = s;
	}
	virtual void OneTick(){
		current--;
	}
	bool Done() const{
		return (current == 0);
	}
	void Reset(){
		current = setting;
	}
	void Go()
	{
		while (this->Done() == false)
		{
			cout << current << "*";
			this->OneTick();
		}
		cout << "ding!\n";
	}
};
class UpwardCounter : public Timer
{
public:
	UpwardCounter(int s) : Timer(s){
```

```
                current = 0;
        }
        virtual void OneTick(){
                current++;
        }
        virtual bool Done() const{
                return (current >= setting);
        }

};
class MultCounter : public UpwardCounter
{
protected:
        int mult;
public:
        MultCounter(int s, int m) : UpwardCounter(s){
                mult = m;
                current = 1;
        }
        virtual void OneTick(){
                current = current * mult;
        }

};
int main()
{
        Timer timerA(4);
        timerA.Go();
        cout << endl << endl;

        UpwardCounter upA(4);
        upA.Go();
        cout << endl << endl;


        MultCounter multC(8, 2);
        multC.Go();
```

```
        MultCounter multD(20, 3);
        multD.Go();
        cout << endl << endl;

        timerA.Reset();
        timerA.Go();
        cout << endl << endl;

        upA.Reset();
        upA.Go();
        cout << endl << endl;

        multD.Reset();
        multD.Go();

        return 0;
}
```

Pure virtual function

A virtual member function is a function that **must** be overridden in a derived class that has objects. The following example is what a pure virtual function looks like.

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

- Must have no function definition in the base class.
- The = 0 indicates a pure virtual function.

Question

When designing a system, we can choose between virtual function or pure virtual function. Could you give an example that pure virtual function is needed?

Exercise 3

Reuse any sample program above which has virtual function.

Step 1

Change the virtual function to pure virtual function. Can the program compile and run?

Step 2

Comment out the corresponding function in the derived class. Can the program compile and run?

Remarks on Runtime Polymorphism

Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer

1. The pointer variable *p* is declared as the pointer to a base class.
2. The object is of a direct or indirect derived class.
3. The address is assigned to *p.*
4. The function in the base class must be virtual or pure virtual.
5. In the run time the function in the derived class is called.