25.x — Chapter 25 summary and quiz

And so our journey through C++'s inheritance and virtual functions comes to an end. Fret not, dear reader, for there are plenty of other areas of C++ to explore as we move forward.

Chapter summary

C++ allows you to set base class pointers and references to a derived object. This is useful when we want to write a function or array that can work with any type of object derived from a base class.

Without virtual functions, base class pointers and references to a derived class will only have access to base class member variables and versions of functions.

A virtual function is a special type of function that resolves to the most-derived version of the function (called an override) that exists between the base and derived class. To be considered an override, the derived class function must have the same signature and return type as the virtual base class function. The one exception is for covariant return types, which allow an override to return a pointer or reference to a derived class if the base class function returns a pointer or reference to the base class.

A function that is intended to be an override should use the override specifier to ensure that it is actually an override.

The final specifier can be used to prevent overrides of a function or inheritance from a class.

If you intend to use inheritance, you should make your destructor virtual, so the proper destructor is called if a pointer to the base class is deleted.

You can ignore virtual resolution by using the scope resolution operator to directly specify which class's version of the function you want: e.g. `base.Base::getName()`.

Early binding occurs when the compiler encounters a direct function call. The compiler or linker can resolve these function calls directly. Late binding occurs when a function pointer is called. In these cases, which function will be called can not be resolved until runtime. Virtual functions use late binding and a virtual table to determine which version of the function to call.

Using virtual functions has a cost: virtual functions take longer to call, and the necessity of the virtual table increases the size of every object containing a virtual function by one pointer.

A virtual function can be made pure virtual/abstract by adding “= 0” to the end of the virtual function prototype. A class containing a pure virtual function is called an abstract class, and can not be instantiated. A class that inherits pure virtual functions must concretely define them or it will also be considered abstract. Pure virtual functions can have a body, but they are still considered abstract.

An interface class is one with no member variables and all pure virtual functions. These are often named starting with a capital I.

A virtual base class is a base class that is only included once, no matter how many times it is inherited by an object.

When a derived class is assigned to a base class object, the base class only receives a copy of the base portion of the derived class. This is called object slicing.

Dynamic casting can be used to convert a pointer to a base class object into a pointer to a derived class object. This is called downcasting. A failed conversion will return a null pointer.

The easiest way to overload operator<< for inherited classes is to write an overloaded operator<< for the most-base class, and then call a virtual member function to do the printing.

Quiz time

1. Each of the following programs has some kind of defect. Inspect each program (visually, not by compiling) and determine what is wrong with the program. The output of each program is supposed to be “Derived”.

1a)

```
#include <iostream>

class Base
{
protected:
	int m_value;

public:
	Base(int value)
		: m_value{ value }
	{
	}

	const char* getName() const { return "Base"; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base{ value }
	{
	}

	const char* getName() const { return "Derived"; }
};

int main()
{
	Derived d{ 5 };
	Base& b{ d };
	std::cout << b.getName() << '\n';

	return 0;
}
```

Show Solution

1b)

```
#include <iostream>

class Base
{
protected:
	int m_value;

public:
	Base(int value)
		: m_value{ value }
	{
	}

	virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base{ value }
	{
	}

	virtual const char* getName() const { return "Derived"; }
};

int main()
{
	Derived d{ 5 };
	Base& b{ d };
	std::cout << b.getName() << '\n';

	return 0;
}
```

Show Solution

1c)

```
#include <iostream>

class Base
{
protected:
	int m_value;

public:
	Base(int value)
		: m_value{ value }
	{
	}

	virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base{ value }
	{
	}

	const char* getName() override { return "Derived"; }
};

int main()
{
	Derived d{ 5 };
	Base b{ d };
	std::cout << b.getName() << '\n';

	return 0;
}
```

Show Solution

1d)

```
#include <iostream>

class Base final
{
protected:
	int m_value;

public:
	Base(int value)
		: m_value{ value }
	{
	}

	virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base{ value }
	{
	}

	const char* getName() override { return "Derived"; }
};

int main()
{
	Derived d{ 5 };
	Base& b{ d };
	std::cout << b.getName() << '\n';

	return 0;
}
```

Show Solution

1e)

```
#include <iostream>

class Base
{
protected:
	int m_value;

public:
	Base(int value)
		: m_value{ value }
	{
	}

	virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base{ value }
	{
	}

	virtual const char* getName() = 0;
};

const char* Derived::getName()
{
	return "Derived";
}

int main()
{
	Derived d{ 5 };
	Base& b{ d };
	std::cout << b.getName() << '\n';

	return 0;
}
```

Show Solution

1f)

```
#include <iostream>

class Base
{
protected:
	int m_value;

public:
	Base(int value)
		: m_value{ value }
	{
	}

	virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base{ value }
	{
	}

	virtual const char* getName() { return "Derived"; }
};

int main()
{
	auto* d{ new Derived(5) };
	Base* b{ d };
	std::cout << b->getName() << '\n';
	delete b;

	return 0;
}
```

Show Solution

2a) Create an abstract class named Shape. This class should have three functions: a pure virtual print function that takes and returns a std::ostream&, an overloaded operator<< and an empty virtual destructor.

Show Solution

2b) Derive two classes from Shape: a Triangle, and a Circle. The Triangle should have 3 Points as members. The Circle should have one center Point, and an integer radius. Overload the print() function so the following program runs:

```
int main()
{
    Circle c{ Point{ 1, 2 }, 7 };
    std::cout << c << '\n';

    Triangle t{Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }};
    std::cout << t << '\n';

    return 0;
}
```

This should print:

```
Circle(Point(1, 2), radius 7)
Triangle(Point(1, 2), Point(3, 4), Point(5, 6))
```

Here's a Point class you can use:

```
class Point
{
private:
        int m_x{};
        int m_y{};

public:
        Point(int x, int y)
                : m_x{ x }, m_y{ y }
        {

        }

        friend std::ostream& operator<<(std::ostream& out, const Point& p)
        {
                return out << "Point(" << p.m_x << ", " << p.m_y << ')';
        }
};
```

Show Solution

2c) Given the above classes (Point, Shape, Circle, and Triangle), finish the following program:

```
#include <vector>
#include <iostream>

int main()
{
        std::vector<Shape*> v{
          new Circle{Point{ 1, 2 }, 7},
          new Triangle{Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }},
          new Circle{Point{ 7, 8 }, 3}
        };

        // print each shape in vector v on its own line here

        std::cout << "The largest radius is: " << getLargestRadius(v) << '\n'; //
write this function

        // delete each element in the vector here

        return 0;
}
```

The program should print the following:

```
Circle(Point(1, 2), radius 7)
Triangle(Point(1, 2), Point(3, 4), Point(5, 6))
Circle(Point(7, 8), radius 3)
The largest radius is: 7
```

Hint: You'll need to add a getRadius() function to Circle, and downcast a Shape* into a Circle* to access it.

Show Solution