Inheritance and Polymorphism

Rong Li

What is happening here?

Because we are passing this object by value, we are not using a reference, this object has to be copied into this target parameter. As part of this copy operation, the compiler has to discard the stuff that we have in the textbox that doesn’t exist in the widget.
More precisely, that is the value attribute. So the size of a string object by default is 24 bytes, and that means when we pass this textbox object to the function the compiler has to discard 24 bytes of data.
This is called: **Object Slicing**

The **Object Slicing** doesn’t happen if we use a reference or a pointer.

For example: if we use a reference parameter here:

```
void showWidget(Widget& widget) {
    widget.getValue();
}
```

The warning goes away because no copy is done. We are passing a reference to an actual textbox here.

However, even though we are passing a textbox object here,
we don’t have access to any members of the textbox here:

We have the same scenario if we had a pointer here:

```
void showWidget(Widget* widget) {

}
```

Again, no slicing happens, and, in this function, we only have access to members of Widget class.

```
int main()
{
    TextBox box;
    showWidget(&box);

    return 0;
}
```

Here we have to use the address of operator(&):

Overriding Methods

Sometimes, we need to redefine a method in the derived class.

For example, let’s add a new method to Widget class: draw( )

The algorithm for drawing a textbox is definitely different from a generic widget. By the same token, every special kind of widget like checkbox, radio button and so on needs its own drawing algorithm, so here, we need to **redefine** this method in the TextBox class.

Pass the textbox object to the showWidget( ) function:

Question: what do you think we are going to see when we run this program?

Drawing a Widget.
or
Drawing a TextBox.

Drawing a Widget.

Here is the reason: in this case, even though we are passing a textbox object by reference to this function, the compiler uses a technique called: **static or early binding**

At **compile time**, it knows that because we are working with a widget object, the draw method after Widget class has to be called.

What if we want to draw a textbox?

This is where we use the **virtual** keyword.

With this, will tell the compiler to use a different technique called:
late or dynamic binding.

At **run time**, the compiler will decide what draw method to call depending on the kind of object we pass to the function.
If we pass a textbox, that draw method after TextBox class will be called.

Back to TextBox class:
Because we declared draw( ) method as **virtual** for the Widget class, we say, we are **overriding** the draw( ) method in the TextBox class instead of redefining.

Overriding means: redefining a virtual method

Practically speaking, you should not really redefine your method, you should override them. **If you have a method that might be changed in derived classes, you should always**

- **declare that method with virtual,**
- **and then override it in the derived classes**.

When overriding a virtual method, the best practice is to add the “ override “ keyword at the end of the method in the derived class:

```
class TextBox : public Widget
{
public:
// overriding
void draw() const override;

};
```

With this, we are telling the compiler that we are actually overriding the draw( ) method of the base class. If tomorrow we decide to make any changes in the signature of this method in the base class, the compiler will warn us.

If you have methods that are likely to change in the derived classes, you should declare them using the “ *virtual* “ keyword. And in the derived classes you should override them using the “ *override* “ keyword.

Polymorphism

It allows us to build applications that can be easily extended.

Refers to the situation where an object can take many different forms. This is a very powerful technique.

When we run the program:

```
Drawing a TextBox.
Drawing a CheckBox.
```

This is where **polymorphism** happens.

The showWidget( ) function, has a parameter of type Widget, at runtime, depending on the type of object we pass into the function, this Widget takes many different forms. In one example, taking the form of TextBox, in other example, taking the form of CheckBox. This is polymorphism in action.

Polymorphic Collections

Polymorphism becomes very interesting when we work with what we call: Polymorphic Collections of Objects.

Example:
Go to main file:
1. Declare a vector of Widgets
2. Add a TextBox and a CheckBox into this vector
3. Loop over this vector and draw each of them

Run the program we see the result:

```
Drawing a Widget
Drawing a Widget
```

What happens to our TextBox and CheckBox?

The reason we don’t see the proper messages because here we have a vector of Widget objects. Even though we are adding a TextBox or a CheckBox to this collection, these objects have to be copied and stored as Widget objects. More accurately, here we are experiencing **Objects Slicing**.

Even though the compiler is not warning about this. Because we have a vector of Widget objects when we call the draw method, obviously, the draw method of Widget class is called. Early I told you that this is called **static or early binding**, means, the compiler matches a function call with the proper definition of the function at **compile time**.

What we need here?

We need **dynamic or late binding**. We want the compiler to match the function call with the right function at **runtime**: if you have a TextBox object we want the draw( ) of the TextBox class to be called. Similarly, if you have a CheckBox object we want the draw( ) of the CheckBox class to be called.

How can we do that?

Using Pointers

1. Change this to a vector of Widget pointers.
2. When we done with these objects, we have to delete them.

Rewrite the program by using smart pointers:

- Create a vector of unique pointer of Widgets.
- Add a unique pointer of TextBox and a unique pointer of CheckBox into the vector.
- Loop over this vector and draw each of them.

Polymorphic Collections

Now, what we have here is called a polymorphic collection:

```
std::vector<std::unique_ptr<Widget>> widgets;
```

Each object in this collection can take a different form at runtime.

Virtual Destructors

If we have a pointer to a base class, a problem happens when these derived objects are destructed.

Add a destructor in each of these classes: Widget, TextBox and CheckBox.

Run the program and see the result:

```
Destructing a Widget
Destructing a Widget
```

Our destructors are not called properly.
Early, I told you that, in inheritance, the destructors of our classes are called in reverse order.
To destroy a TextBox object, first, the destructor of the TextBox class should have been called followed by the destructor of its base class.
And similarly, to destroy a CheckBox, first, the destructor of CheckBox should be called followed by the base class.
Obviously, the destructor of these derived classes are not called.

Can you tell why this is happening?

The problem is, again, **static or early binding.** So the compiler has decided that the destructor of the Widget class should be called.

How can we solve this problem?

By using **dynamic or late binding.** To make that happen, we need to declare the destructor of the Widget class as **virtual.**

Anytime you declare a method in a class as **virtual,** that means you are going to use **polymorphism** at some point. When you use **polymorphism**, you have to make sure that your objects are destroyed properly. So whenever you declare a method as **virtual,** you should always add a **virtual destructor** in that class as well. Even if that destructor is not going to do anything, you still need to implement it.

We are simply providing a message; we are not going to do that in a real application. So we will have an empty destructor. We don’t have to define this destructor explicitly; we can have the compiler generate for us:

```
class Widget
{

public:

virtual ~Widget() = default;

};
```

Abstract Classes

Sometimes we might have a virtual method, but there is really no meaningful way to define that method, example: draw( ) of the Widget class.
Look at the implementation of this method: purely for demonstration. But in a real framework, how can we draw a widget? We don’t know.
Each kind of widget should have its own drawing algorithm: how we draw a textbox is deferent from a checkbox.
But there is no meaningful way to define how a widget should be drawn. This is where we use a **pure virtual method**.

```cpp
class Widget
{

public:

// pure virtual function
virtual void draw() const = 0;

};
```

With this we can remove the definition of this function.

If a class has at least one pure virtual method, we refer to that class as **abstract**. So our Widget class is an **abstract class** now.

Abstract class cannot be instantiated. We cannot create new objects from them. So we cannot create Widget object from main ( ). Because this class is abstract, we don’t have an implementation for this method. So what should happen if we create a Widget object and call draw ( ) method doesn't make sense.

Abstract classes cannot be instantiated. Means: we cannot have a variable of type an abstract class.
And similarly, we cannot use them in function parameters.

If remove “&” here, we get the same completion error.

```
void showWidget(Widget& widget) {

    widget.draw();
}
```

But we can still use abstract classes as reference or pointers, because we need them for polymorphism behavior.

Abstract classes cannot be instantiated, they exist purely to be inherited, purely to provide some code to other classes.

When inheriting abstract classes, if we don't override their pure virtual method, our new class will be abstract as well.

For example: in TextBox class, if we don't override this pure virtual method:

```
void draw() const override;
```

The TextBox class will also be abstract. So in main, we will not be able to create TextBox object.

Recap, if we have a method that we cannot define in a meaningful way, we should declare it as a pure virtual method with no implementation.

Final Classes and Methods

Sometimes, we want to prevent a method from being overridden in a derived class. This is where we use the “final” keyword.

For example: in the TextBox class we have the implementation of the draw( ):

```
public:
// overriding draw()
void draw() const override;
```

Let’s imagine we don’t want any derivatives of this class: TextBox to override the draw( ) further, you want to make sure all of our TextBoxes have the same drawing algorithm:

```
void draw() const override final;
```

Now, we cannot create a new class that derives from the TextBox class and override the draw( ) further.

We can also declare a class as *final*.

Back to TextBox class: if you add “final” keyword here:

```
class TextBox final: public Widget
{
};
```

Then, we cannot create a new class that derives from the TextBox class:

```
class MaskedTextBox: public TextBox
{
};
```

We got a compilation error: based TextBox is marked final.

Deep Inheritance Hierarchies

The common mistake amongst people who start out with object-oriented programming:
They abuse inheritance and create deep, complex and fragile hierarchies that easily break.

The problem is that whenever we use inheritance between two classes, we are creating **dependency** or **coupling** between them. The higher in the hierarchy, we make a change, the costs of the change is going to be higher.

Deep Inheritance Hierarchies

Don’t go for deep inheritance hierarchies. Limit the inheritance to a maximum of 3 levels. Anything more than that will be going to get complicated and probably going to bite you later on.

Multiple Inheritance

In C++, a class can also have multiple base classes.
This is referred to as: Multiple Inheritance.

A great example of multiple inheritance is the iostream class.
Iostream has two parents: istream and ostream. That’s why it give s us both reading and writing capability.

What is the diamond problem?

The diamond problem happens in languages that support multiple inheritance. If two classes (B, C) derive from A and are also the parents
of another class (D), we see a diamond. If the top class (A) declares a method (eg toString) and its children (B and C) override this method, it’s not clear which implementation will be inherited by D.

Does Java support multiple inheritance?