Lecture 5: Polymorphism and Virtual Functions

Introduction

The topic of polymorphism is given mystical status in some programming texts and is ignored in others, but it's a simple, useful concept that the C++ language supports. According to the standard, a "polymorphic type" is a class type that has a virtual function. From the design perspective, a "polymorphic object" is an object with more than one type, and a "polymorphic base class" is a base class that is designed for use by polymorphic objects.

Essentially classes present us with a way of expressing game entities that have attributes and behaviours. Inheritance was a way of logically extending classes, or game attributes. Polymorphism is another method of logically extending those attributes.

Aims

Today we are aiming to advance our knowledge of OOP programming, discussing the topic polymorphism, by having a look at virtual functions and using pure virtual functions to declare abstract classes.

What is a Virtual Member Function

From an OO perspective, it is the single most important feature of C++.

Essentially the aim is that a member functions will produce different results depending on the type of object being called.

This works in game this way. You get attacked by a wave of enemies, the enemies are made up off different types of enemies that are related through inheritance. You call the attack member function for all the enemies and the type of each object will determine the effects. Though this can be accomplished through overriding with polymorphism the results are determined at runtime.

A virtual function allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer. This allows algorithms in the base class to be replaced in the derived class, even if users don't know about the derived class.

The derived class can either fully replace ("override") the base class member function, or the derived class can partially replace ("augment") the base class member function. The latter is accomplished by having the derived class member function call the base class member function, if desired.

Declaring a Virtual Function in C++
A virtual function is declared in a base class of a program and can then be redefined in each derived class. The declaration in the base class acts as a kind of template which can be enhanced by each derived class.

In the base class, you must begin the function declaration with the keyword virtual.

The keyword virtual is not used in the derived functions.

The number and type of parameters in the derived function must match the number and type in the initial declaration of the virtual function.

The keyword virtual is used only in the base class declaration.

When a derived class redefines the virtual function in a base or predecessor class, this is referred to as overriding the function. Don't confuse this with overloading a function

Friend functions cannot be virtual, nor can constructor functions, although destructor functions can be virtual.

Checkout and compile the online code

- *initial_example.cpp*
- *VPlayer.cpp and VPlayer.h*

Redefinitions of a virtual function override the definition in the base class rather than overload it. This is because the resolution of the function is decided at run-time rather than during the compilation.

If a pointer to a base class object is assigned the address of a derived class object, the pointer remains pointing to the base class members and can only be used to access the base class members.

In the provided example *base_ptr* would still point to the instance of the base class, even though it has been assigned to the derived instance.
However we have declared our member function of the base class as virtual. The pointer can therefore be used to access the redefinition of the function in the derived class rather than the base class. This type of behaviour is referred to as "dynamic binding". The correct implementation of the function is selected at run-time.

Checkout and compile the online code – example2.cpp

In this example the functions aren't virtual so base_ptr still points to the instance of the base class.

**Note:*

When a virtual function is called by name and uses the dot operator, the reference is resolved at compile time and is referred to as "static binding". Virtual functions are declared using the virtual keyword.

Virtual Functions and Inheritance
A derived class does not have to redefine (override) a virtual function.
In other words, if a derived class does not specify code for the virtual function, then the code in the predecessor function is used.

A derived function inherits the virtual functions of its predecessors which may or may not be the base class.

Once a function has been defined as virtual, it remains virtual down the inheritance tree from that point, even if it's not declared virtual in sub-classes. However, it's considered good programming practice to explicitly declare these functions virtual to aid clarity.

If a derived class doesn't define the function, the derived class inherits the function from the class it's being derived from.

The redefinitions in the derived classes of virtual functions must have identical arguments as those in the base class. ANSI C++ has approved differing return types for virtual functions. Static members may not be declared virtual.

Overloaded operators may be declared virtual, apart from the "new" and "delete" operators as they become static, and static members can't be declared virtual.

Checkout and compile the online code – example3.cpp

In this example derived_class_2 prints out “first class” instead of “second class”. This is due to inheritance of virtual functions.

Pure Virtual Functions
As you've seen in the previous example, if a derived class does not override a virtual function, it is the function definition in a predecessor class that is used. If the immediate predecessor has not overridden its predecessor, then the one before that is referenced to find a function definition. This search could continue (if predecessors don't override the virtual function) until, ultimately, it is the function definition in the base class that is used.

However, in some cases, it does not make sense for the base class to define the virtual function at all. The actual method should really be implemented in each one of the descendent classes. The base function should simply provide a kind of placeholder for the virtual function and leave it up to the descendents to specify the individual methods.

A virtual function that is declared but not defined in a base class is referred to as a pure virtual function. Here is the general syntax for declaring a pure virtual function:

virtual type function_name(parameter_list)=0;

A pure virtual function in declared in C++ by initializing a virtual function to zero.
Now when the base class uses a PURE virtual function, each descendent must override that function, or you will get a compile error. This makes sense, because the function has to be defined somewhere.

Any class containing any pure virtual functions is an abstract data type, ADT. Because there is no definition for the function, you cannot create an object of that class. You can however, declare pointers to it.

Think of an abstract class as a general class that lays the foundation for descendent classes that define their own methods. This is the heart of polymorphism - let each class define its own operation.

<u>Checkout and compile the online code – abstract.cpp and abstract.h</u>

Constructors and Destructors
Constructors may not be declared virtual, but destructors may be declared virtual to ensure an orderly de-allocation of class objects.

If the base class destructor is declared virtual, then all derived classes automatically have virtual destructors.

Destructors for non-virtual base classes are executed before the destructors for virtual base classes. If memory is allocated in a class that contains one or more virtual functions, then a virtual destructor should be used to ensure that the memory is deallocated correctly. This is because the derived classes constructor will automatically invoke the base classes destructor.

<u>Virtual Function FAQs</u>

When are Virtual and Non-virtual Member Functions Called:
Non-virtual member functions are resolved statically. That is, the member function is selected statically (at compile-time) based on the type of the pointer (or reference) to the object.
In contrast, virtual member functions are resolved dynamically (at run-time). The member function is selected dynamically (at run-time) based on the type of the object, this is called dynamic binding.
Dynamic binding means that the address of the code in a member function invocation is determined at the last possible moment: based on the dynamic type of the object at run time. It is called "dynamic binding" because the

binding to the code that actually gets called is accomplished dynamically (at run time). Dynamic binding is a result of virtual functions.

Are Virtual Functions (Dynamic Binding) Central to Object Oriented C++:
Without virtual functions, C++ is just a syntactic variant of C. Operator overloading and non-virtual member functions are great, but they are, after all, just syntactic sugar for the more typical C notion of passing a pointer to a struct to a function.

From a business perspective, C++ without virtual functions has very little value-add over straight C. Technical people often think that there is a large difference between C and non-OO C++, but without OO, difference usually isn't enough to justify the cost of training developers, new tools, etc. In other words, if I were to advise a manager regarding whether to switch from C to non-OO C++ (i.e., to switch languages but not paradigms), I'd probably discourage him or her unless there were compelling tool-oriented reasons. From a business perspective, OO can help make systems extensible and adaptable, but just the syntax of C++ classes without OO may not even reduce the maintenance cost, and it surely adds to the training cost significantly.

Bottom line: C++ without virtual is not OO. Programming with classes but without dynamic binding is called "object based," but not "object oriented." Throwing out virtual functions is the same as throwing out OO

Why do Virtual Functions (dynamic binding) make a Big Difference:
Overview: Dynamic binding can improve reuse by letting old code call new code.

Before OO came along, reuse was accomplished by having new code call old code. For example, a programmer might write some code that called some reusable code such as printf().

With OO, reuse can also be accomplished by having old code call new code. For example, a programmer might write some code that is called by a framework that was written by their great, great grandfather. There's no need to change great-great-grandpa's code. In fact, it doesn't even need to be recompiled. Even if all you have left is the object file and the source code that great-great-grandpa wrote was lost 25 years ago, that ancient object file will call the new extension without anything falling apart.

That is extensibility, and that is OO.