Virtual Functions

OK, today's topic in Object Oriented C++ is function dispatch. First, let's review overloading. What is function overloading? *Two methods or functions with the same signature.* Why do you overload a function? *For polymorphism.* Now, mechanically, how do you overload something? *Not only the method name, but the number and type of arguments and the return type must all match.* Why? *The name and arguments so that you know it's an overload; the return type so that the compiler always knows the data types at run time...*

OK, so let's create an example of overloading. Let's say I have an Animal class with a “warm up” method. It might look like:

```cpp
class Animal {
  public:
    bool Warm-up() const {return false;}
    void Behave() { if (cold()) warm-up(); }
};
```

Notice that is returns false, because there is no way for a generic Animal to warm up. But if we get more specific...

```cpp
class Mammal : public Animal {
  public:
    bool Warm-up() const {shiver(); return true;}
};
```

Notice that this is an example of function overloading, because the name, arguments and return type all match.

Now, what happens if I write the following code?

```cpp
Mammal m;
m.Behave();
```

In particular, which version of Behave() gets called? Notice that m is a mammal, but Behave() is a method of Animal, so inside Animal() the declared type of the object is Animal, not Mammal.

If I did this example in Java, the answer is that the mammal would shiver and return true. But as written, in C++ the animal does nothing and Warm_up returns false. But this is C++, I can do whatever I want.

To understand what is going on, you have to understand the difference between dynamic (a.k.a. virtual) dispatch and static (a.k.a. direct) dispatch.

In static dispatch, the compiler hardwires a jump to a method or function directly into the code. The method is selected based on the declared data types of the arguments. In the example above,
the Behave() method is compiled to a code that directly calls the Warm_up() method of Animal.
This is simple and fast.

Static dispatch is the default in C++. This is different from Java.

Dynamic dispatch, on the other hand, tests the data type of the arguments at run-time, and uses
them to determine what version of the function to call. In the example above, if dynamic dispatch
is used, the Behave() method would compile to code that checked the run-time data type of *this
(the instance it is called on) and jumps to the version of Warm_up*

This is how functions are dispatched in Java. If you want dynamic dispatch in C++ you can have it:
declare the method virtual. So once again, C++ gives you a choice: dynamic dispatch or static
dispatch.

Syntax: all methods are statically dispatched unless they are declared virtual. If a method is
declared virtual, it is virtual and any function that overloads it also virtual. You cannot declare a
statically dispatched function that overloads a dynamically dispatched one.

Thus, in the example above, if we declare Animal’s Warm_up() to be virtual, Mammal’s
Warm_up() is automatically virtual. But relying on that is not good style. It is better style to
declare Mammal’s Warm_up() to be virtual too.

Which is better? Most of the time, virtual dispatch. Why? Because if you didn’t want the
Mammal’s “warm_up” method to override the Animal’s, then why did you use the same function
signature?

But there are exceptions:
- When a method manipulates private data (because the child will not have access to that
data)
- When every last ounce of speed/space matters (static dispatch is faster)
- When you are confident no classes will be derived from the one you are writing.

Of these, only the first is likely to come up.

But this class is partly about “under the hood”. How are virtual functions implemented?
Remember that objects are compiled independently, so the Animal class doesn’t know that other
classes are derived from it. It doesn’t know about Mammals, and it certainly doesn’t know about
Quaggas or Horses.

Virtual methods are implemented through what is known as a virtual function pointer table. The
idea is simple: as soon as you declare one or more methods in a class to be virtual, the compiler
creates a virtual function pointer table for that class.

For example, as soon as the warm_up method of Animal is declared, a virtual function pointer
table is defined for Animals. The virtual function table maps function signatures to addresses. So
in this example, the warm_up() signature would be mapped to the address of the Animal version
of the warm_up method.

Note that Animal may have many virtual functions. If “Behavior()” is also a virtual method, the
table would also match “Behavior()” to its address.
So the compiler builds a virtual function pointer table for every class (not object), and that table is stored in memory at run-time. Furthermore, a new field is added at the head of every Animal instance, and this is the address to the virtual function table for Animals.

So when a virtual method like warm_up is called, the instructions generated by the compiler look like this:
- Follow the address in the first field of the instance to the virtual function table.
- Offset into this table an amount associated with the method name.
- Jump to the address found there.

So instead of a direct function call, there is an extra pointer follow and addition. The good news is that the virtual function table will usually be in cache, so this is pretty fast (but not quite as fast as direct dispatch).

OK, now how is Mammal implemented so as to make Warm_up() dynamically dispatched? Well, it gets its own virtual function pointer table, but it begins by inheriting Animal’s. This means that it inherits an entry for warm_up, and one for make_sound. If the Bird method overrides one of these virtual methods, it doesn’t make a new entry; it overrides the address the old one (remember: it has a different table than animal). And just like for Animal, the first thing in the class is the virtual function pointer table. Note that this is the same location as for Animal; the pointer inherited from Animal is replaced! This is the only exception I know of to the inheritance as union rule.

So now, the dispatch code written for Animals will have a different effect if handed a Bird. It will still follow the address in the first field of the instance, but now that will take it to the virtual function pointer table for Birds, not Animals. It will add the same offset, and jump to the Bird version of the warm_up method.

So this is how virtual functions are implemented, even though the parent class knows nothing about its children, and is compiled independently from them. Clever, huh?

Why teach you this? A couple of reasons:
- Some people find it easier to keep the semantics of direct and virtual dispatch straight if they know how they are implemented.
- If there is an error calling a virtual function, you will see the pointer to the virtual function pointer table in the debugger and/or error statement. Now you know what it is.
- Because I mentioned earlier that an instance of a class containing only a single int data field might be bigger than in int. This is why: if it has any virtual functions defined, it needs a pointer to the virtual function pointer table.
- Because it’s good to know what it going on under the hood (bridge the gap between CS200 and CS270).

It also brings me to a new topic: pure virtual functions.

Why do we have inheritance hierarchies? What type of relation do they reflect? Unions, which are good for Is-a relations.

So what happens as you go up the hierarchy? Objects become more abstract and less detailed. This is the natural result of the union operator.
This is one of those basic points about object-oriented programming that gets lost. The reason we have objects is to encapsulate closely-related data and code. The reason we have inheritance is to support abstraction.

So what happens as you go up the hierarchy from Quagga to Mammal to Animal? Abstraction. As you go up the hierarchy you are including a larger and larger set of related objects. Or, equivalently, loss of detail. The higher up the hierarchy you go the less information you have. But this is a good thing – abstraction is about the loss of unnecessary details.

Polymorphism is another buzzword here. Literally it means that an object can have many forms: a Quagga can be seen as Quagga or an Equine or a Mammal or an Animal…

OK, so what role do virtual functions play in this scheme? The idea is to take advantage of detail when possible, but not rely on it. For example, all Animals have a way to warm up or make sound – that’s why it was a property of the top-level class. But how does an Animal warm up? I know how a mammal does it and a bird does it and a fish does it and…

So this is the weakness of my running example: how do I implement the warm_up method for Animals in general? In the example above I did nothing and returned false. Not very useful.

So what do virtual functions do to the concept of abstraction? Well, the idea is that the general form of the abstraction is still valid – the function name, arguments and return type, but that the function can be replaced with a more specific, and therefore more valid, version.

In the extreme, it may not be possible to compute the value in general. One could argue that warm_up() is a valid abstraction because all Animals can do it, but that to ask a generic Animal is warm_up is meaningless. Birds can do it and fish can do it, but not generic Animals.

In this case, warm_up is an interface: every Animal must have a warm_up method, but it can’t be implemented in the abstract.

This is what pure virtual functions are for: interfaces. They declare that every child of the parent class (in this case, every type of Animal) must implement a function called LifeSpan, but that Animal itself can’t implement it.

Pure virtual functions are a requirement: if Animal declares LifeSpan to be pure virtual, you cannot make an instance of any class that inherits Animal but fails to implement LifeSpan. This is a compile-time error.

The syntax is simple:

```cpp
Class Animal {
    Public:
        Virtual int LifeSpan() const = 0;
    ...
};
```

At the level of implementation, what you are saying is this: there exists a virtual function called LifeSpan, so make an entry for it in the virtual function table.

But it isn’t defined yet, so set the pointer in the virtual function table to be null (0).
At compile-time, the compiler will not generate code to create an instance of any class with a null pointer in its virtual function table. So classes that extend Animal must define LifeSpan, or they can’t be instantiated.

Note that if mammal defines warm_up, but not fish or bird, then mammals can be created, but not fish and birds. Of course, once you have created a mammal you can use it as an Animal…

Does that make sense?