Virtual Function and Type Casting

Today's notes are primarily about casting, but first I want to talk about pure virtual functions and interfaces. Interfaces are a common and powerful form of abstraction. In some sense, they are the logical extreme of polymorphism. Polymorphism says the same general idea (or object) can take many shapes (hence the buzzword). Interfaces say they don't care about the implementation (or "shape") of the idea; they only care about the abstract services it provides.

In Java, interfaces and objects are two different things. They have to be, since multiple inheritance is not allowed. A class cannot inherit both an interface and an implementation, so interfaces can't be objects that have to be inherited. C++ is not burdened with this restriction.

In C++, an interface is simply a class with one or more pure virtual methods. The idea of a pure virtual method is simple. We already talked about virtual methods, and how they are implemented through the virtual function pointer table (VFPT). This supports polymorphism by allowing a method to be invoked according to the type of an object as constructed, as opposed to the type of the object as lexically declared.

But go back to my recurring polymorphism example: we have an Animal type, which is inherited by both Mammal and Fish. I defined a warm_up() method for Animal, because every animal has a way to warm up if needed. Mammals warm up by shivering; fish warm up by swimming upward toward warmer water. So each child class redefines warm_up, and as long as warm_up is virtual, every Animal warms up using the method that is appropriate for them.

But how do I define the warm_up method for the Animal class? I need one, otherwise I can’t call warm_up for an Animal, defeating the polymorphism. In essence, I want to declare warm_up as part of an interface: all Animals must implement warm_up. So what do I put in this function?

I could put the following in the Animal class:

```cpp
virtual void warm_up() {throw std::exception();}
```

This is better than doing nothing, because if the Animal version of warm_up gets called, something is wrong. But I won’t find the error until run-time, and then only if a successfully test the case that generates it. Compile time errors are better.

So C++ defines the pure virtual function. A pure virtual function creates an entry in the VFPT, but puts NULL into that entry. The syntax is:

```cpp
virtual void warm_up() = 0;
```

Semantically, it says that there is a warm_up method that must be implemented before any instance of this class can be created. As a result, the compiler will throw an error if you try to make an instance of an Animal. You can, however, make an
instance of a class that inherits Animal, if and only if it redefines the warm_up method.

For example, if Mammal and Fish both redefine warm_up, it is possible to make instances of Mammals and Fish, and even treat them as Animals. But you can’t make an instance of Animal. (Instances of classes that inherit Mammal or Fish are also OK, since they will inherit a valid version of warm_up.)

Pure virtual functions are why C++ doesn’t need interfaces. An interface is just a class with one or more pure virtual functions. The pure virtual functions define the set of methods that anything inheriting it must provide.

But they are quite powerful, since not all the methods in an interface class need to be pure virtual. There can other methods associated with Animal that are implemented for Animals; they might not even be virtual. There can also be data fields. After all, it’s a class, and can have anything a class has. It just has one or more pure virtual methods that stop instances of it from being created until the interface is satisfied.

OK, new topic: type conversion, a.k.a. type casting.

What is type casting? Type casting is where you take an object of one type (for example integer), and make it another type (for example double). We often type cast primitives. We will also type cast up and down object hierarchies. Polymorphism relies on implicit type casting. Rarely, we will type cast other things.

What is really going on when we type cast? It depends on what data type we are converting to what data type. If I type cast an int to a double, the underlying bit pattern has to change in order to keep the semantic value the same. Therefore when I type case an int to a double, the compiler inserts code (usually a function call) to change the underlying representation.

On the other hand, if I cast an Animal* to a Mammal*, the underlying representation doesn’t change. No code is generated. The compiler just internally notes at compile time that the type of the pointer has changed.

Type casting is made even more confusing because there are four ways to do it in C++: implicit, C-style, functional-style, and the new “best” style. In general, having too many ways to do the same thing is poor language style, but hey (not my fault).

My recommendation is to limit yourself to implicit casting in certain safe situations, and new style casting otherwise.

So let’s go over new style casting, which has 4 different types of casts. This can be confusing, but the idea is that the underlying semantics are exposed, making the code easier to read. And then I will tell you when I think they are necessary.

Case #1: static casting. This is the most common cast. It is used to cast one related data type to another. For example, you can cast a double to an int, or vice-versa. The syntax is:
Double d = ...;
Int I = static_cast<int>(d);

Static casting always converts the data type, using the best conversion function available. In the case of double to int, it will truncate the number and change its representation to be an integer.

Using static_cast with pointers or references is dangerous: it will perform the compile-time type conversion whether it makes sense or not. This is equivalent to just reinterpreting the pointer, and is a bad idea. For pointers and references, use dynamic_cast.

Static casting is usually not safe, but it will not generate warnings. The assumption is you know you are explicitly doing something unsafe.

Static casting usually generates run-time code to change underlying representations (except for pointers and references).

My recommendation: safe up-casting (e.g. int to double) can be left implicit, but any static casts that generate a warning (like double to int) should be explicitly written as static_cast's.

Case #2: dynamic casting. This is common because of polymorphism: you will want to use a child class as one of its parents, and you will sometimes need to cast them back again. So you will need to cast pointer and references up and down the class hierarchy.

Note that dynamic casting makes no sense with actual objects, just pointers to them. (An object is what it is.)

The syntax is:
B* b_ptr = ...;
A* a_ptr = dynamic_cast<A*>(b_ptr);

The semantics is that if b_ptr points to an instance of A* (because it was created as an A* or as a child of A*), then a_ptr will be set to B* with the new compile-time type. Otherwise, a_ptr will be set to NULL.

Stylistically, every dynamic_cast should be followed by a test to see if it returned NULL, for obvious reasons.

This is sometimes referred to as RTTI (run-time type inference), and it is relatively new to C++ (borrowed from Java). It is very convenient. If I have a vector of Animal* and I want to check the color of just the birds, I can use this to skip Processing of all my non-bird Animals.

Note that a dynamic_cast operates both at compile time (changing the declared type) and run-time (generates code to check the virtual function table).

My advice: its OK to cast up the hierarchy (child -> parent) implicitly, because such casts are always safe. Always use dynamic_cast when casting down the hierarchy.

Now, there are some new types of casting:
Case #3: reinterpret_cast. This is dangerous, but sometimes necessary. This cast says to reinterpret the compile-time type of a piece of data without touching it. For example, I could write:

```cpp
Double b = 3.1;
Int a = reinterpret_cast<int>(b);
```

What would this do? It doesn't change the bits in \( b \) at all, but it reinterprets them as an int. As a result, \( a \) gets whatever integer value is the same bit pattern as the double number \( b \).

I don't have to tell you how dangerous this is. But sometimes it is necessary, for example when writing device drivers. Never use it unless you absolutely have to, and always make it explicit with reinterpret_cast when you do.

Case #4: const_cast can be used to convert type A to const A and vice-versa. When converting from a non-const to a const, it is OK to leave it implicit however (since this is safe, and generates no run-time code).

Const-casting the other way if exceedingly dangerous. I can declare a const method (for example), and then internally const_cast one of its fields to be non-const and then change it. But as far as the compiler or anyone else knows, the method is still const. Very dangerous. There is, however, one good reason to have it in the language. It is common in large C++ projects to have make calls to old C libraries. Compatibility with C libraries is one of our original reasons to program in C++, after all. Unfortunately, const isn’t used in old C libraries, even when they don’t side-effect their arguments. Therefore, IF you need to call a C library function AND it doesn’t side-effect its arguments, THEN you can use const_cast to (temporarily) remove the const so that you can pass the argument to the C function. Any other use is very, very, very strongly discouraged.

Not done yet. There is another proposed type of cast that has not been adopted yet, but implementations exist (e.g. Boost) and they are in Stroustrup’s book:

Narrow cast. Sometimes you need to convert from a large numeric type (like long int) to a smaller one (like int or even short). This is different from rounding, as in the double to int case. If the value in the long int fits in the smaller representation, then this cast is safe. If it doesn’t fit in the range of the target, it’s a disaster.

Narrow_cast makes sure it fits. If it does, it puts the value in the smaller target. Otherwise, it calls an error.

Narrow_cast is a really good idea, particularly since the sizes of data types of implementation dependent in C++. I recommend using it always when this situation occurs.

One snag: it’s not officially in the language yet. You can grab it in include files from the web if you need it, however, or use Stroustrup’s implementation in your auxiliary text book.