Multiple Inheritance

We are returning once again to the topic of Object Oriented programming in C++. There are four topics we want to discuss:

1. Shadowing
2. Multiple Inheritance
3. Pure Virtual Classes (Interfaces)
4. Slicing

Let's start with shadowing, and remember what we have already said about the motivation for OO programming:

1. Encapsulation
2. Abstraction through Inheritance
3. Inheritance as Union.

Let's look at the union notion (you could also call it inclusion) in a little more detail. What happens if B inherits A, and A has a field a type Quagga called "pete". Is the following legal?

```
B b(…); // initialize instance b of B
b.pete;
```

Of course it is! Everything in A is in B, including pete.

But now what happens if I add to my definition of B:

```
class B : public A {
public:
      int pete;
};
```

Now is the previous line (`b.pete`) legal? Yes, but it now means something different: it refers to the int declared in B, not the one defined in A.

What if you want to reference the Quagga in A inside a method of B?

```
q = A::pete;
```

This is even uglier when not in a class method:

```
q = b.A::pete;
```

Note that this only works if A is a parent of the current class…

Reusing a variable name (or a method name & arguments, if the method is not virtual) is called *shadowing.* It goes without saying that shadowing, i.e. "hiding" a variable by duplicating the name of a public or protected field in a parent class, is a bad idea. Even though there is always a syntactic way is access a hidden variable, it is bad style and will lead to confusions.

OK, new topic: multiple inheritance. I want to start with the same caveat as with operator overloading: multiple inheritance is a feature of the language. You don't have to use it. I almost never do (unlike operator overloading, which I use more often).

But I want to talk about it anyway. Why? Two reasons. First, there is an informal design pattern where multiple inheritance is not only elegant, it is the best way to do certain things. Second, because multiple inheritance is different from anything in Java, so it makes you think about objects in C++.

The general idea is simple: one class can extend two different "parent" classes. Using the general idea that inheritance is union, this makes sense: class C is the union of two parents A & B, plus whatever is defined for C.

But when would you want to do this? The canonical (and as far as I know, original) example comes from computer graphics. Some of the first object oriented systems (in a language called SmallTalk) were developed for graphical windowing systems. Of course, back then most people still had gray scale monitors, and some had binary (black & white) monitors. A few people had color monitors, but they were rare. So they built a window class, and specialized it: color window, grayscale window, and black and white window.

At the same time, they started to figure out that how a window is framed is visually important. Some users liked a thin, flat boundary. Others liked a beveled or 3D boundary. Some applications needed boundaries with their own buttons (like X windows today).

The problem was, they wanted the cross-product of features. In other words, any window could be b&w, gray scale or color, and any window could have flat, beveled or buttoned boundaries. How could they accomplish this without replicating a lot of code?

Their solution was multiple inheritance. They built a window depth hierarchy (b&w, etc.) and they built a window boundary hierarchy. Then they used multiple inheritance to create a class for every combination.

This was elegant. Sure, they had a lot of classes, but the child classes had no code: they were just the union of their parents. No code was repeated.

This developed into the notion of "mixin's" (the SmallTalk developers liked ice cream). Every widget (as we now call them) was a mixture of multiple parents, implementing different aspects of the widget.

You can implement this pattern today in C++. When it applies, it is elegant. But now we have to think through the implications, because there are hidden dangers.

The first criteria for a mix-in design is that you need a cross-product of abilities. Heirarchy X has N choices, hierarchy Y has M choices, and all possible unions XxM (or at least most of them) are needed. Not common, but not unheard of.

But we have to be careful. Remember that inheritance is union. Let's say that class D extends both class B and class C. But assume also that B and C have a common parent A. Now what?

Well, every instance of B contains a copy of A. But every instance of C also contains a copy of A. D is the union of B and C, so it contains *two* copies of A!

This is a problem. Let's say A contains an int called value. Then D contains two ints called value. This is a syntactic problem: A::value is now ambiguous. Which of the two values do you want? It's ambiguous. This effectively makes some of the values in D (inherited from A) unreachable.

Note: there is a syntactic solution (B::A::value), but its ugly and you don't want to use it.

Why? The bigger problem is semantic. There are two copies of value in D. They may well get out of sink, i.e the values might be different. For example, if A is the top of the widget hierarchy and contains a field called 'width', then the copy of width in B might be 50 and the value of width in C might be 75. Now D becomes a window such that some of its methods think it is 50 pixels wide, while other methods think it is 75 pixels wide.

To continue with the analogy: if B is the class of color windows, and C is the class of beveled frame windows, and D extends B & C you have a problem. Only a 50 pixel wide window will be colored in, but the frame will be 75 pixels wide, creating a 25 pixel gap.

This leads to the most important rule of thumb for multiple inheritance: never use multiple inheritance if two (or more) parents share a common ancestor class.

Formally, this is known as 'orthogonality'. For multiple inheritance to work well, the parents of a class must be totally independent of each other. They should have no common ancestors. Even more, they should not contain independent representations of the same data. For example, I could modify the example above so that pixel depth and color did not have a common parent, but if they both represent the width of the window, you still have a problem.

So when to consider using multiple inheritance (i.e. mixins)? When you need the cross product of *orthogonal* properties.

This happens a lot with interfaces. If I am doing geometric modeling, for example, objects have color properties, reflectance properties, shape properties, histories, etc. But these are not universal: translucent and/or mirrored objects have no colors; liquid objects have no shape, etc. It's nice to have multiple interfaces, and different objects implement different subsets.

Multiple inheritance also has some interesting polymorphism implications.
Polymorphism, you may recall, is one of the main motivations for object oriented programming. The idea is that a parent class may have many derived classes (a.k.a. children or forms). It doesn't matter which one it is. If they support the same interface (as defined by the parent), you can substitute any of the children.

For example, imagine you have an Animal class with three derived classes: fish, lizard and mammal. It could have a "raise body temperature" method. The fish implements it by swimming upward (to warmer water). The lizard moves out of the shade. The mammal shivers and eats more. All accomplish the same goal; they all raise body temperature. The exact form of the Animal doesn't matter.

Let's say B extends A. If I have a method or function that takes a pointer to A as an argument, I can pass it a pointer to B, right? After all, every pointer to B is also a pointer to B.

From an implementation standpoint, this is because the first thing in B is a copy of A. What if B has multiple parents? They can't both be first in B. What if I pass a pointer to B to a function that takes a pointer to A's other parent?

Let me point out that this does work. You can always substitute a child for a parent in C++, even if the child has multiple parents. But how is it implemented?

The answer is that everything has a compile-time data type in C++, so the compiler knows when you are substituting a child for a parent (at least the first time). It knows the offset of the parent class relative to a child. When you pass a pointer to a child to a function expecting a pointer to a parent, this offset is added to the pointer.

For example, let's say that class C inherits from both class A and class B. Now I declare two vectors:

Vector<A*> foo;
Vector<B*> bar;

Now what happens when I fill these?

C* ptr = new C;
Foo.push_back(ptr);
Bar.push_back(ptr):

What happens? In the first case, ptr (the address of C) is pushed onto vector foo. In the second, the offset of B in C as added to ptr, and then the resulting address in pushed onto bar.

It's an interesting example of unwittingly putting an addition operation into your code. The type cast (upcast) from C* to B* may involve a pointer addition.

Now let's return to a theme of the course. C++ supports multiple inheritance. Java does not. Why? It's not quite the normal "because programmers could make mistakes" answer. This time the answer is even more basic. Java has a single inheritance hierarchy. This means that every pair of classes share a common ancestor. ('Object' if nothing else) Therefore the rule that says "never use multiple inheritance for non-orthogonal classes" always imply. There are no orthogonal classes in Java, so multiple inheritance is always a bad idea!

It's a non-obvious implication of a single object hierarchy: multiple-inheritance designs don't make sense. C++ does not enforce a single hierarchy, so multiple inheritance designs, i.e. mixins, can work. But not that often.