Multiple Inheritance in C++

by Andrei Milea

Using multiple inheritance in C++

Deriving directly from more than one class is usually called multiple inheritance. Since it's widely believed that this concept complicates the design and debuggers can have a hard time with it, multiple inheritance can be a controversial topic. However, multiple inheritance is an important feature in C++ and C++ programmers think of it as a very good structuring tool.

To get a taste of this, let's consider the following real world example:

```
class Animal
{
        //describes the behavior of the animal
}

class Drawing
{
        //contains the drawing properties of the entity like colors, size etc
}
```

Suppose we need to create a displayable snake object. We can inherit class Animal in class Drawing or class Drawing in class Animal and then use the derived one as a base class for snake, but neither of the solutions is appropriate because they both create a dependency between two independent concepts. So, instead, we should use multiple inheritance and derive both Drawing and Animal in the snake class:

```
class Snake : public Animal, public Drawing
```

As you may noticed, another solution in the above design could be to use composition instead of inheritance and have animal and drawing as members of class snake :

```
class Snake
{
        public:
        ...
        private:
        Animal *m_animal;
        Drawing *m_drawing;
}
```

But now the relation between animal and snake is that snake contains an animal instead of snake is an animal, which is conceptually wrong, and the lack of categorization makes the design unrealistic. This will make the code harder to understand and also harder to reuse because we lose the advantage of dynamic binding and polymorphism. The benefit of dynamic binding and polymorphism is that they help making the code easier to extend (make it possible to create operations that work on a class of objects that share the same interface). Suppose we want later to create another animal, a lion for example. The old code that operates on objects of type Animal will work without change, and in the new oneLion class, if it overrides one of the animal methods, dynamic binding ensures that its new methods are going to be executed properly instead of the Animal versions of those methods depending on the object type they are called from.

Using inheritance instead of composition, we can completely hide some members of the parent class, or allow access only for subclasses (classes derived from it) by specifying them as protected.

As with simple (single) inheritance the derived class has access to all the non-private members of the base classes. When the Snake's class constructor is executed it first initializes the base classes by calling their appropriate constructors in the order they are defined in appear in the list defining the list of inherited classes when they are declared (first Animal and then Drawing).

Note that our example has only two base classes, but C++ doesn't impose any constraint on the number of classes that can be inherited.

Some design patterns benefit from the use of multiple inheritance. One of them is the Adapter pattern: it uses multiple inheritance to adapt one interface to another. This pattern is usually used to provide implementation for an abstract class (when you want to use an existing class and its interface is not appropriate, so you need to .glue. another interface with that class). Another design pattern that can be created with

multiple inheritance is the Observer pattern. In this design pattern a class, called subject, maintains a list of observers that are notified by some change (user input) by calling one of their functions.

Pitfalls

The most difficult to avoid complication that arises when using multiple inheritance is that sometimes the programmers interested in using this technique to extend the existing code are forced to learn some of the implementation's details. The second trivial problem that might appear when using this technique is the creation of ambiguities:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : public A ,public B { void f(); };
```

This issue can be solved by using explicit qualification. We explicitly say to the compiler where to get the function that we need to call:

```
C* pc = new C;
pc->f();
pc->A::f(); //this calls f() from class A
pc->B::f(); //this calls f() from class B
```

Each base class can be uniquely identified by using the scope resolution operator :: .

If the C class didn't have the method f() the problem couldn't have been solved with explicit qualification. Instead we would have used implicit conversion :

```
A* pa = pc;
pc->f();
```

or we would have to make a cast in order to call the method from the parent class A.

A more complicated situation that arises when using multiple inheritance is the diamond problem, which occurs when two classes are derived from a base class and another class is obtained by joining the derived classes together. If the derived classes override the same method from the base class when calling the method from the merged class and the joining class does also override that method, an ambiguity will rise. A second problem that can occur with the diamond pattern is that if the two classes derived from the same base class, and that base class has one or more members, then those members will be duplicated in the joining class. For a more detailed discussion of this problem and its solution, check out solving the diamond problem with virtual inheritance.

Inheriting from one base class and a number of interface classes (classes that contain no members of method definitions) is called multiple interface inheritance. (It is also sometimes referred to as mixin . the interfaces are mixed in one class that implements them.) This type of programming is usually used in Java where full-fledged multiple inheritance is not allowed. It is an alternative to multiple inheritance and solves the problem of knowing which parent will be called because we need to implement those inherited methods (since the interface provides only method signatures). This can be a big limitation when the base classes we want to inherit already exist and they are not abstract, which usually may happens in programming because it can't always be known what things interfaces are going to be needed when making the design. In that situation, it is often appropriate to refactor the code to split the non-abstract base class into an interface class and an implementation class. But doing that is not always possible, if the code you are working with it is not all your own.