Tute 12: Advanced Inheritance

Review: Inheritance

What is private/protected inheritance?

Consider the following scenario:

```
class Base { };
class Derived : public Base { };
```

Here any code that works on a pointer or reference of type Base will also work on type Derived, eg:

```
void polymorphic(Base *base) { }
polymorphic(new Derived());
```

This is because the inheritance relationship is public.

In the case of protected inheritance, only Derived and its children are aware that it inherits from Base, so while the above code would not compile, any member functions from children of Derived can use the Base class. This also means that any members from Base default to protected visibility inside Derived.

In the case of private inheritance, only Derived is aware of the inheritance relationship, not its children. This also means that any members default to private visibility inside Derived.

What is virtual inheritance?

Virtual inheritance is typically used in the context of multiple inheritance. Consider the following class hierarchy:

```
class Top
{
public:
        int a;
}

class Left : public Top
{
public:
```

```
        int b;
};

class Right : public Top
{
public:
        int c;
};

class Bottom : public Left, public Right
{
public:
        int d;
};
```

In this case Top is inherited from twice (once through Left, and once through Right). This means that Bottom will have two attributes called a:

```
Bottom::Left::a;
Bottom::Right::a;
```

This also means that to cast a Bottom to a Top, the type of Top needs to be explicitly stated (ie Left::Top or Right::Top):

```
Top t = (Top) (Left) bottom;
```

This also means that all operations inside Bottom that require data or operations from Top, must go through Left or Right.

To solve this problem, virtual inheritance can be used:

```
class Top
{
public:
        int a;
}

class Left : public virtual Top
{
public:
        int b;
};

class Right : public virtual Top
```

```
{
public:
        int c;
};

class Bottom : public Left, public Right
{
public:
        int d;
};
```

Now Bottom will be derived from Top only once, and only one copy of the data from Top will be present in Bottom. This also means that casting does not need to go through Left or Right, since a cast to type Top is no longer ambiguous. Likewise operations inside Bottom no longer need to go through Left or Right.

Programming Concepts: More Inheritance

For these questions refer to composition.cpp and diamond.cpp from the lecture 11 examples folder.

How can private inheritance be used to simulate composition?

The example depicts a Car class, which contains an instance of the Engine class (ie, Car has-a Engine relationship). This is called composition and should not look new. However, the above example can be altered to use private inheritance like so:

```
class Car : private Engine
{
public:
        Car() : Engine(8) {}
        using Engine::start;

};
```

Note the use of using Engine::start; If we did not do this and attempted to call car.start() we would have a compile error due to start being a private member of the Engine class. This is because when using private inheritance all members of the class we are deriving from are converted to private.

In both of the above cases there is exactly one Engine in every Car. Outsiders cannot convert a Car* to an Engine*, but using private inheritance members of Car can. The private inheritance

variant allows access to the protected members of the base class thanks to using inheritance, Car can override Engine's virtual member functions and may introduce unnecessary multiple inheritance, and in both cases the Car class has a start method that calls the Engine class' start method.

In the above case, there isn't a need for private inheritance and using composition works fine, however there are some cases where private inheritance provides a more elegant solution:

```
class Wilma
{
protected:
	void fredCallsWilma() { ... }
	virtual void wilmaCallsFred() = 0;
};

class Fred : private Wilma
{
public:
	void barney() { Wilma::fredCallsWilma(); }
protected:
	virtual void wilmaCallsFred() { ... }
};
```

How can virtual inheritance be used to allow siblings to call methods on each other?

Using the example, if we introduce virtual inheritance:

```
class Person;
class Student : public virtual Person;
class Employee : public virtual Person;
class Tutor : public Student, public Employee;
```

Tutor will now be derived from Person only once, instead of twice as in the previous example. This allows classes to delegate functionality to a sibling class:

```
class Person
{
public:
	virtual void foo() = 0;
	virtual void bar() = 0;
};

class Student: public virtual Person
```

```
{
public:
	virtual void foo() { bar() }
};

class Employee: public virtual Person
{
public:
	virtual void bar() { std::cout << "bar\n"; }
};

class Tutor: public Student, public Employee
{
};

int main()
{
	Tutor t;
	t.foo();
	return 0;
}
```

Notice that Student::foo() actually calls Employee::bar(), even though it knows nothing about that class. Note that this will only work if the Tutor class is also present and the member function foo() is called from the Tutor class.

Compilation: vtables

What is a vtable?

A virtual function table (vtable) is a mechanism used to support dynamic binding in C++. It is a table containing pointers to function locations that is created for a class whenever that class has a method declared as virtual.

Normally when the compiler encounters a method call on an object, it will insert the memory address of that method into that location in the executable (usually at link time). If the class has a vtable, instead the address of the method's entry in the vtable is inserted, and then at runtime the method at that entry location is called, eg:

```
class Base
{
public:
```

```
        virtual void foo() { }
        virtual void bar() { }
};

class Derived : public Base
{
public:
        virtual void foo() override { }
};

Base b;
Derived d;
```

Would produce classes that look like:

b:
    +0: pointer to vtable of Base

vtable of Base:
    +0: pointer to Base::foo
    +4: pointer to Base::bar

d:
    +0: pointer to vtable of Derived (from Base)

vtable of Derived:
    +0: pointer to Derived::foo // overridden
    +4: pointer to Base::bar // not overridden

Here b and d both container pointers to the vtables for classes Base and Derived, which in turn store the correct functions which need to be called at runtime. The numbers represent offsets from their current location (in bytes).

Note that a class can have more than one vtable if it inherits from multiple base classes, although this is compiler dependent.

How are vtables used in the context of virtual inheritance?

Using the classes defined earlier in the virtual inheritance question, the memory layout for non-virtual inheritance appears as:

```
    Bottom:
    Left::Top::a;
```

Left::b;
Right::Top::a;
Right::c;
Bottom::d;

Here it is clear that in order to access data from Top, either Left or Right must be used. In the case of virtual inheritance, the memory layout appears as:

Bottom:
Left::b;
Right::c;
Bottom::d;
Top::a;

Here Top::a has been moved to the bottom of the class. This is so it can be accessed by the vtable of Bottom which will have two vtable entries, one for Left and one for Right, both of which expect Top::a to appear after them, so that the offsets will still work (vtables cannot have negative offsets).

This means that any attempt to access a, even though it is a data member and not declared as virtual, must go through the vtable.

Errors: None this week

Debugging: None this week

Exercises: None this week