Tute 10: More Classes & Inheritance

Review: Inheritance

How do you extend a class?

You can extend a class by declaring:

```
class FlexiAccount : public Account
```

This means that the FlexiAccount class extends the Account class. You can think of this relationship as "FlexiAccount is a type of Account." Using the "is-a" relationship is a good way of thinking of inheritance. Another type of relationship is the "has-a" relationship which we will cover later. In the above case, the Account class is the base class and the FlexiAccount class is the derived class. Some ways we can think of this are:

- FlexiAccount is a type of Account
- FlexiAccount is derived from Account
- FlexiAccount is a subclass of Account

An important behaviour to note is that when you call the constructor of a derived class (ie, create an instance of a derived class) you will automatically call the constructor of the base class. If you want to call a specific constructor of your base class, you need to call it in the initializer list of the derived class like so:

```
FlexiAccount() : Account()
```

It is also possible to have a pointer to a base class that points to objects of any class derived from the base class, so in this example, it is possible to have an Account pointer that actually points to a FlexiAccount object (but not vice versa).

What is a virtual method?

A virtual method is a method whose definition can be overridden (to an extent) by an inheriting class, ie:

```
class A
{
        virtual void foo() { cout << “foo”; }
};
```

```
class B : public A
{
        virtual void foo() { cout << “bar”; }
};

A *a = new B();
a->foo();
```

This would print bar rather than foo, as the method has been overridden by B, which is the object type that is actually calling the method (even though it is using a pointer of type A). We can also cast a to type B (although this may give undesired behaviour if it is not actually of type B):

```
B *b = (B*) a;
```

An extension of this is a pure virtual method, which is a method that has no definition, eg:

```
class A
{
        virtual void food() = 0;
};

class B : public A
{
        virtual void foo() {}
};

A *a = new B();
a->foo();
```

A class which contains a pure virtual method is considered to be abstract, which means that it cannot be instantiated. Any class derived from an abstract class must provide definitions for the pure virtual methods of the abstract class, or else the derived class will also be abstract.

What is polymorphism?

Polymorphism (in the case of Object Oriented languages, which use subtyping) is where a function has been written to take a certain type of object T, but will also work if passed an object that is a subtype of T (ie, inherits from or extends T), for example:

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

```
void someFunction(A *a) { }
```

The above code will work if it is passed an object of type A or B. Now recall that methods on classes are also simply functions where the first argument is the object that is calling the method, and it becomes clear that polymorphism can also be used on class methods, ie:

```
class A {
        virtual void someFunction() {}
};
class B : public A {
        virtual void someFunction() {}
};
```

Here we can call someMethod on objects of type A or type B, and more importantly, we can use dynamic binding to determine which method is called at runtime (ie the choice of which function to execute depends on the type of object the method has been called on).

What is the difference between static and dynamic binding?

Static binding is where the program decides what method to call at compile time. An example of this is an ordinary function call or a method call on a non-virtual method, ie:

```
class A {
        void someFunction();
};

class B {
        void someFunction();
};

A *a = new B();
```

Here the code will call the someFunction() method on A, since the method has not been made virtual, and therefore static, rather than dynamic binding will be used.

In dynamic binding the program will decide what method to call at runtime by looking up the appropriate function address in the class vtable (more on this later) and calling that function (this means virtual method calls require two function calls as opposed to one).

```
class A {
        virtual void someFunction();
};
```

```
class B {
        virtual void someFunction();
};

A *a = new B();
```

Here the program will call the someFunction() method on B.

Programming Concepts: More inheritance

What is/is not inherited by a derived class?

Things that are inherited by the derived class include:

- Every data member defined in the parent class
- Every method in the parent class

Things that are not inherited include:

- The base class constructor/destructor
- The base class assignment operator
- The base class friend classes

Note: in c++11 and onwards we can explicitly request that constructors are inherited by the use of a “using” statement, eg:

```
class base
{
    int i_;
    public:
    base(int i) : i_(i){}
};

class derived : public base
{
    //inherit constructors from the base class
    using base::base;
};
```

What is multiple inheritance?

Multiple inheritance is where a class inherits from more than one base class, ie:

```
class C : public A, public B {};
```

The derived class will have the functionality of all the base classes combined. This is similar to how interfaces work in Java, but extended to a full class, rather than just const data and prototypes.

How are override/final used?

Note: these are both C++11 features.

Override is a keyword that is used to give a compile error when a virtual method doesn't actually override anything, ie:

```
class A {
	virtual void someFunction() {}
};

class B : public A {
	virtual void someFnction() override {}
};
```

This will give a compile error, as the someFnction() method on B doesn't override any methods on A since it has a typo, which means it is treated as a different method. Without the override keyword this code would compile, and potentially behave incorrectly if the programmer was assuming it was overriding the someFunction() method on A.

final is used to prevent a method from being overridden by any derived class, ie:

```
class A {
	virtual void someFunction() {}
};

class B : public A {
	virtual void someFunction() final {}
};

class C : public B {
	virtual void someFunction() override {}
};
```

This will also give a compile error, since someFunction() cannot be overridden by any class derived from B, as it has been declared final inside B.

What kind of casts are available in C++? What dangers are there in using vanilla C style casts?

We have the following casts in c++ that are intended to replace the C style casts:

- dynamic_cast: this is for safe casting up and down the inheritance hierarchy. Please see below for full examples of this.
- const_cast: constness in C++ is much stronger than in C and in fact it is often complained about that this is viral. Nevertheless, const correctness is an important concept in C++ and when you get this wrong, your data may be modified by chunks of code that really should not have the right to do it. However, there are times when you want to cast away that constness and override c++ const semantics and when you need to do this, this is the right operator.
- static_cast: this allows casting to valid types for which there is a conversion operator or conversion constructor.
- reinterpret_cast: ignores the type of the thing we are casting and forces it into the new format. This is rarely the right thing to do but sometimes we have to do it. For example, when doing binary I/o we force everything to be char* and in that context, that is correct because we are going to the low level and treating everything as just a bunch of bytes.

How do dynamic_cast and typeid work?

dynamic_cast and typeid are both used for Runtime Type Identification (ie, determining the type of an object at runtime). dynamic_cast works in a similar way to the instanceof operator in Java and can be used to check if an object is a specific type, eg:

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

Base *base = new Derived();
Derived *d = dynamic_cast<Derived*>(base);
```

This will return null if base is not an object of type Derived, otherwise it will cast base to type Derived (as it will in this case).

The typeid keyword is used to determine an object's class, without testing it against any specific types, eg:

```
auto typeInfo = typeid(*base);
```

```
cout << typeInfo.name();
```

Will print Derived if base is type Derived, or Base if it is type Base.

What is inlining?

Normally when a function is called, the compiler will insert the address of the function so the program can jump to that function to execute the code there, however, when an inline function is called, rather than the compiler inserting the address of the function, the function's code is inserted instead (similar to how a macro works). Note that when a class is implemented in the header its methods are implicitly inline and thus separating functionality between header and source files is important if you don't want the compiler to inline all of your function calls. Note that the compiler can decide whether functions will be inlined or not, even if you don't declare a function to be inlined (or even if you do).

What is operator overloading?

In C++ the overloading principle applies not only to functions, but to operators too, which means that operators can be extended to work not just with built-in types (int, float, etc) but also classes. This means you can provide an operator as a method in a class, and then use that operator with any object of that class type, ie:

```
class Decimal
{
        Decimal &operator + (const Decimal &d) {}
};

Decimal a, b;
Decimal c = a + b;
```

Most operators can be overloaded. In C++ the only operators that cannot be overloaded are:

- Member operator a.b
- Member indirection operator a.*b
- Scope resolution a::b
- Ternary conditional a ? b : c
- sizeof sizeof(a)
- typeid typeid(a)
- static/dynamic/const/reinterpret_cast static_cast<type>(a)

What is an enum?

An enum is the ability to assign an integer value a specific type, eg:

```
#define NORTH 0
#define SOUTH 1
#define EAST 2
#define WEST 3

void move(int dir);
```

Here any integer value can be passed to move(), even ones that aren't defined. Instead we can write this as:

```
enum Dir { NORTH, SOUTH, EAST, WEST };
void move(Dir dir);
```

Now only valid Dir values can be passed to move(). These can also be used in switch statements:

```
switch (dir)
{
case NORTH:
        break;
case SOUTH:
        break;
case EAST:
        break;
case WEST:
        break;
}
```

enums can be even more strongly typed by declaring them as:

```
enum class Dir { NORTH, SOUTH, EAST, WEST };
```

This means that an int can no longer be passed to the move() method, it must specifically be of type Dir. Dir::NORTH, Dir::SOUTH, etc, is the correct way to reference a member of the Dir enum class.

What is a virtual destructor?

Destructors are methods used to delete objects. When a destructor is called, all code in the destructor is executed and memory allocated for the object is freed. Destructors are declared like so:

```
~Object() {}
```

The destructor is automatically called for objects on the stack when their stack frame is popped (they go out of scope) and if no destructor has been declared for a class, a default destructor is provided.

A virtual destructor is a destructor that has been declared as a virtual method, and so uses dynamic binding:

```
virtual ~Object() {}
```

This is handy if you have created an object polymorphically:

```
class A {
        virtual ~A() {}
};

class B {
        virtual ~B() {}
};

A *a = new B();
delete a;
```

In this case it is necessary for the destructor for B to be called, which can only happen if the destructors in A and B have been made virtual.

Note on destructors in C++11 and onwards:
Any classes that you want to store in a safe pointer object such as unique_ptr and safe_ptr must have a virtual destructor implemented. It doesn't have to have any code inside it but it must be implemented as otherwise the c++ runtime cannot find the destructor and you will have memory leaks.

Compilation: None this week

Errors: None this week

Debugging: None this week

Exercises:

Write the class definitions and member functions for the following situations:

- Meat and Vegetables are Foods.
- Herbivores and Carnivores are Animals.
- All animals can eat Food.
- Herbivores eat Vegetables, and Carnivores eat Meat.
- If a Herbivore eats Meat, or if a Carnivore eats Vegetables, they abstain().
- An Omnivore is a special case of a Herbivore and a Carnivore - it can eat both Meat and Vegetables.

Notes:

- There are not a lot of differences in the implementations between Herbivore and Carnivore, and Vegetables and Meat, so if time is short, it is reasonable to assume that the students could work out how to do it themselves, and you can move on to the next bit.
- Animal should really be a virtual base class, so that there aren't two copies of it within Omnivore - but the main point of that question is to show how multiple inheritance syntax works, and resolving ambiguous calls to base classes.
- The whole Food::foodType() thing is a hack to avoid using exceptions to catch bad cast on the reference, and to make Food polymorphic.
- This question will take a while, which is why I haven't put constructors in most classes (the students can work them out). If you are running out of time, don't worry about doing the member function denitions for Herbivore and Carnivore - just show Omnivore, and perhaps let the students assume that the Food hierarchy exists already.

```
#include <iostream>
using namespace std;

class Food
{
public:
        enum Type { VEG, MEAT };
        Food();
        virtual Type foodType() = 0;
};
```

```
class Vegetables : public Food
{
public:
	Type foodType() { return VEG; }
};

class Meat : public Food
{
public:
	Type foodType() { return MEAT; }
};

class Animal
{
public:
	virtual void eat(Food &f) = 0;
	abstain() { cout << "Abstaining\n"; }
};

class Herbivore : public Animal
{
public:
	virtual void eat(Food &f)
	{
		if (f.foodType() != Food::VEG)
		{
			abstain();
			return;
		}
		// eat
	}
};

class Carnivore : public Animal
{
	public:
	virtual void eat(Food& f)
	{
		if (f.foodType() != Food::MEAT)
		{
			abstain();
			return;
		}
```

```
                // eat
        }
};

class Omnivore : public Herbivore, public Carnivore
{
public:
        Omnivore() : Herbivore(), Carnivore() { }

        void eat(Food& f)
        {
                if (f.foodType() == Food::VEG)
                {
                        Herbivore::eat(f);
                }
                else if (f.foodType() == Food::MEAT)
                {
                        Carnivore::eat(f);
                }
                else
                {
                        abstain();
                }
        }
};
```