COSC 2P95 – Lab 7 – Section 01/02 – Inheritance and Templates

For this lab, we'll be looking at reviewing the inheritance we learned in lecture, and reinforcing templates. This lab will be more of a 'worksheet' style, so feel free to spend time tinkering, or to skim towards the lab task, per your own preference.

Inheritance

We're going to start by looking at inheritance. This will largely just review what we learned in lecture, so it shouldn't take long, so long as you've been following along.

Let's start by defining something simple, a Vehicle base class:

```
class Vehicle {
private:
	std::string bumperSticker;
protected:
	void becomeSentient(){std::cout<<"Why was this added?"<<std::endl;}
public:
	int wheels; //Unicycle? Tricycle? Dodecacycle?
	double clearance; //Distance between chassis and ground
	Vehicle(int wheels=4, double clearance=18) : wheels(wheels), clearance(clearance) {}

	int getWheels() {return wheels;}
	double getClearance() {return clearance;}
	void rollOut() {std::cout<<"Brockobots! Roll out!"<<std::endl;}
	void recycle() {std::cout<<"You owe $45 for disposal."<<std::endl;}
	void setBumper(std::string b) {bumperSticker=b;}
	std::string getBumper() {return bumperSticker;}
};
```

First, note the three different *access specifiers*:

- `private`
  - *Only* the Vehicle class can directly access the bumper sticker
- `protected`
  - Both Vehicle and derived types may have access to becomeSentient, though external classes and code are still denied access to it
- `public`
  - Unrestricted access to clients

Let's create our first derived class, PedalPowered:

```
class PedalPowered : public Vehicle {
private:
	using Vehicle::wheels; //There's no need for public access to this
	bool spoked;
public:
	PedalPowered(int wheels=2, bool spoked=true, double clearance=18)
		: spoked(spoked), Vehicle(wheels,clearance) {}
	//This is only possible because becomeSentient was protected (or public) in Vehicle!
	using Vehicle::becomeSentient;

	void rollOut() {std::cout<<"*huff puff* Brock...*wheez*ob.... *thud*"<<std::endl;}
	//recycle is the same (inherited from Vehicle)
	void replaceChain(){std::cout<<"I have no 'funny' jokes for this one."<<std::endl;}
	bool getSpoked() {return spoked;}
};
```

First, note that PedalPowered can't initialize Vehicle's member variables directly, but it *can* delegate to Vehicle's constructor.

Also, we're employing two novel ways of controlling access to members:

- Setting an inheritance type to the class itself
  - When PedalPowered extends Vehicle, it provides the same access to members as their original access
  - More generally, this can be used to restrict outside scopes from accessing an otherwise accessible member
- We have two `using` lines to take the functionality from Vehicle, but change the visibilities of just those members
  - By listing a using line for the number of wheels under private, we don't change our internal access to it, but a client program or class can no longer directly access the variable. Also, any child types of PedalPowered won't be able to directly access that variable, either
  - We've also changed the becomeSentient method from being protected to public
    - Note that we couldn't widen the visibility of bumperSticker to protected or public, because (since it's private in Vehicle) it's entirely inaccessible. We can do it for becomeSentient because, as a protected, the PedalPowered class already has *some* access to it

Interjection:
Before we get any further, let's point something out. If we have a Vehicle v, and a PedalPowered p, then we should be able to guess what `v.rollOut()` and `p.rollOut()` should do.
However, what about the following?

```
PedalPowered pp;
Vehicle &v=pp;
v.rollOut();
pp.rollOut();
```

In this case, even though both `v` and `pp` are referencing the same PedalPowered object, only `pp` will actually use the PedalPowered version of the method.
As weird as it sounds, that does in fact make sense.

When you instantiate a PedalPowered, it runs constructors for both classes, and has allocated memory (instance variables/member variables) for both. Since everything you need for a Vehicle is included, there really isn't any harm in being able to invoke the Vehicle methods on the Vehicle data.

However, 'making sense' doesn't mean it's actually the behaviour that you wanted. There exists a pretty easy way to indicate to the compiler that we might be providing a new version of the rollOut method: simply prepend the `virtual` keyword in the *Vehicle* class:

```
virtual void rollOut() {std::cout<<"Brockobots! Roll out!"<<std::endl;}
```

It's up to you whether or not you wish to make that change for the duration of this example. Nothing we implement will rely on one version over the other.

Let's resume:
Let's add another type of Vehicle, Motorized:

```
class Motorized : protected Vehicle {
private:
      bool usesGas; //gas or diesel?
      double fuelLevel;
      double tankCap;
public:
      Motorized(bool usesGas=true, double tankCap=6)
            : usesGas(usesGas),tankCap(tankCap),Vehicle(2,20) {
                  fuelLevel=tankCap;
            }
      void rollOut() {std::cout<<"VROOM! VROOM!"<<std::endl;}
      bool hasGas() {return fuelLevel>tankCap;}
      using Vehicle::setBumper;
      using Vehicle::getBumper;
};
```

There's nothing particularly interesting here, other than listing the bumper sticker methods as being exceptions to our *protected* class inheritance, but here's a question: can we think of a vehicle that's both pedaled *and* motorized? A moped!

```
class Moped : public PedalPowered, public Motorized { };
```

But… wait… what happens if we want to rollOut? Which version would it use?
That is, what happens when we write:

```
Moped mo;
mo.rollOut();
```

It won't even compile. And that makes sense, because `mo.rollOut()` is *ambiguous*.
We can easily specify if we like:

```
mo.Motorized::rollOut();
```

However, what happens if we want to rely on something from the original Vehicle specification? It's a common ancestor to both Motorized and PedalPowered.

You might think this one isn't ambiguous, since only PedalPowered has a public `recycle` method, but the fact that `recycle` is protected in Motorized isn't enough for the compiler to ignore it completely. It still *has* that method.
So, `mo.PedalPowered::recycle();` works just fine.

But what happens if we want to rely on something *else* from Vehicle? Like, say, the bumper sticker methods?
This one's different, because they actually rely on state information (in the instance variables).
This is an example of a significant problem that relies from the *diamond problem*.

Let's demonstrate:

```
mo.PedalPowered::setBumper("Broccoli University or bust!");
mo.Motorized::setBumper("Live free. Die in massive student debt");
```

Quiz time!
Which version do you think is stored as the instance variable?
Trick question! Both!

… seriously?

Yep:
```
std::cout<<"Pedaled: "<<mo.PedalPowered::getBumper()<<std::endl;
std::cout<<"Motorized: "<<mo.Motorized::getBumper()<<std::endl;
```

Again, this *does* make sense when you consider how a single inherited class operates.
A PedalPowered *is-a* Vehicle, and that's effected by allocating memory for both, and running both constructors.
A Motorized *is-a* Vehicle, and that's effected through the same mechanism.

But, what if we don't want that?
Easy-peasy. Just change two lines of code (where the PedalPowered and Motorized classes are declared):

```
class PedalPowered : virtual public Vehicle {
```
and
```
class Motorized : virtual protected Vehicle {
```

Both of these establish Vehicle as a *Virtual Base Class*, which allows it to create the *same* common ancestor of both. The constructor need only be called once, and one set of member variables can be shared.

That isn't to say that you'll *always* want this. Sometimes individual base classes are necessary. If you're aggregating multiple components into a single type, you could very well have (and want) multiple copies of some base type.

For any given tool you're creating, decide how many copies of such a base type you'd have: one or more.

Templates

Recall what we did for Lab 5. In part, we created a simple *stack*.
That one used dynamic allocation, but we could have used an array just as easily.

What mattered was that we had a single data structure that would allow us to *push* elements on, to be later retrieved by *popping* them off. The sequence was First-In-Last-Out (FILO).

In particular, it stored long integer values. If we'd wanted to store something else, we'd have to rewrite a new copy of all of the code to accommodate the new type (and so on, and so on, for each new type to be stored).

There's an easy solution: *templates* can be used so that the nodes will store members of *some* type, with that type being filled in… eventually.

So let's look at a simple Stack (`Stack.h`):

```
//This uses a generic typename, but also an expression parameter
template <typename T, int capacity>
class Stack {
private:
	T arr[capacity]; //Always the desired size!
	int topIdx;
public:
	Stack() : topIdx(0) {}
	void push(const T item) {
		arr[topIdx++]=item;
	}

	T pop() {
		return arr[--topIdx];
	}
```

```
	T top() {
		return arr[topIdx-1];
	}

	bool isEmpty() {
		return topIdx<=0;
	}
};
```

Note that, even though we like separating specification from implementation, we won't be doing that with our templated class. That's because it's pretty unwieldy once templates are involved. We'll be better off just putting them in the same file.
To that end, we're actually just putting it into the header file. Possibly not recommended, but certainly simple.

Besides that, note that we're using templates for two things:

- We're using typename T to declare that we'll be substituting a different *type* in later.
- The `int capacity` portion is known as an *expression parameter*. In this case, it provides a very convenient means of defining the array size without needing to resort to dynamic allocation on the heap (which also means no need to worry about deleting the array, or using a destructor, later on)

There's one last thing worth noting: All of the types here will be copied values (e.g. push will use call-by-value); we aren't using any references.
Hypothetically, there would have been three possible choices for what to store:

- References
  - These are handy, but what happens if the original had been declared on a stack, to later be deallocated? Because the stack is genericized, we want to accommodate the most general usage
- Values
  - This *looks* like what we chose. It can be restrictive in limited cases (e.g. a stack of IO streams could be problematic, particularly if the copy constructor wasn't included). But, so long as the client software is appropriately written, this should be a reasonable choice
- Pointers
  - Pointers avoid the copy issues. But here's the catch: by choosing values, that value type *can be* a pointer. As such, this behaviour can be subsumed by choosing values

Using the stack is simple:

```
#include <iostream>
#include <string>
#include "Stack.h"

int main() {
	Stack<int,20> istk;
	istk.push(32);istk.push(20);istk.push(40);
	std::cout<<istk.pop()<<"\t"<<istk.pop()<<std::endl;

	Stack<std::string,50> sstk;
	sstk.push("hello");sstk.push("class!");sstk.push("templates are neat!");
	while (!sstk.isEmpty())
		std::cout<<sstk.pop();
	std::cout<<std::endl;
}
```

which, of course, yields:

```
20	40
templates are neat!class!hello
```

Submission Task

This is going to sound familiar…

Write a queue, and a simple program to use it. Refer to Lab 5 for a reminder of what a queue is (hint: like a Stack, but FIFO).

However, this time there are more requirements:

- Put your queue in a separate file (comparable to the Stack described above)
  - For the sake of simplicity, just put it in `Queue.h`
- Your queue *must* at least use a template to define the type it stores (as the stack example above does)
- It's up to you whether you want to write it to use an array or dynamically-allocated nodes
  - But if you do choose arrays, then define its size in an *expression parameter*, as in the Stack above

To demonstrate, write a client program of your choice.

Submission

When you're finished, simply demonstrate your sample program (showing the usage of your queue) to your lab demonstrator, to be checked off.