COSC 2P95

Objectively Optimal Objects (or... Calibrated Cool Classes!)

Week 6

Brock University

Remember Object Orientation in C++?

What we've seen so far certainly has value.

- We can achieve basic encapsulation
- We can better organize our solutions and model systems somewhat

However, modern design patterns, modelling approaches, and larger systems tend to include more complicated relationships.

What's more, we haven't addressed the issue of possibly having multiple closely-related classes, and the redundancy that entails.

Inheritance

(Or subtyping)

There are two common scenarios that can come up when designing classes:

1. One class seems like a small modification of an existing class
   - e.g. an emu is really just a particularly tasty bird
2. Two classes have so much in common that they're effectively cousins
   - e.g. be honest. Could *you* tell the difference bewteen a tasty emu and a scrumptious ostrich?
   - Note that, in this case, even if you only cared about delicious emus and ostriches, to the exclusion of all other avian edibles, there's still a common theme that one *could* define independently of them both

Inheritance (subtyping) is the process of defining a type in terms of how it differs from an existing type.

- This opens the door to things like *object polymorphism*

Nomenclature

First, a quick note on terminology:

- Though some languages tend to favour some terms over others, *parent class*, *superclass*, *supertype*, and *base class* are all effectively synonymous; the same goes for *child class*, *subclass*, *subtype*, and *derived class*
- Regardless of what you call them, a derived class allows us to add specialization, while retaining the original generalized case
- Another way to phrase it is to say that a derived type has an *Is-A* relationship with a base type

How inheritance works

A base class first defines what the generalized case entails:

- Member variables, member functions, etc.

Then, a derived class declares itself as a subtype of the base, and defines where it differs from the original:

- Additional member variables and functions may be defined; some existing member functions might be replaced

Derived classes may also act as base classes to further children, establishing an *inheritance chain*.

- e.g. one could create a Unicode string that *is a* string, which *is a* character sequence

We've already used at least one example: an `ifstream` *is a*n `istream`.

Simple introduction to inheritance

Let's just look at a really simple example, using inheritance.

Structure of classes

Classes are a little odd in C++.
A reasonable guess would be that derived classes copy the borrowed content from the base class, to create an entirely separate new class.

- Indeed, that would be reasonable
- It would also be wrong

Instead, a derived class is more like the portions of the newly-defined class, glued onto the base class.

- This can lead to some potentially unexpected behaviour
- For example, if I create a new instance of GraduateStudent, how many constructors will I run?

Let's find out!

Constructors

Okay, so all of the parent classes' constructors are still being called, but we don't seem to have *access* to them.
After all, how do you instantiate a single object with *two* constructors?
Well, you don't.

- This might be a more significant problem than it seems. e.g. Consider setting member constants: that needs to be done at initialization
- We may not be able to

Luckily, there's an easy solution:

- We can simply add an invocation of the base class's constructor to the member initialization list
  - (Is it me, or does Computer Science sometimes sound like completely made-up nonsense?)

Example time!

Access specifiers

Finer-grained information hiding

Now that we're making tangentially-related extensions of existing classes, we'll have to decide how much access we want to give to derived types. This decision should be separate from how much we grant to entirely independent components.

- First, we have a new modifier: `protected`
  - While a *private* member is hidden to all other scopes (including derived classes), a *protected* member is available not only to its class and *friends*, but also to derived types
  - It can be useful when you're writing a class with the expectation that it will end up a base class for further expansion

However, there's something else of note: when we've been extending classes, we've been prepending `public` to the base class's name. What happens if we use `private` or `protected`?

Inheritance access specifiers

To simplify, the access specifier we use when naming the base type becomes the maximum allowable accessibility of members to *other* classes/scopes. For example:

- If you use an inheritance access specifier of `protected`, then private members will still be private, but public members become only accessible to the class and its further derived types
- However, since it only affects how members appear externally, using a `private` modifier will only hide members from those other classes/scopes; the derived class can still access them
- If we want to redefine the access modifiers for a class, but still provide an exception, `using` is useful

If you don't specify an access modifier, it defaults to private.

Example Time! (It's like Adventure Time, but nobody sounds like Bender)

Controlling access to functionality in classes

That `using` trick was neat, but it raises the question of how else we can get creative with the underlying functionality and new definitions.

- We've already seen that we can add additional methods to derived classes that weren't present at all in their base types
  - That can include everything from accessors, to new mutators
- We can also provide a new implementation for an existing header
  - This is called *overriding* the method
- If we provide a new implementation, then we can still access the original base version; thus effectively *expanding* the behaviour, rather than strictly replacing it
- In the same way that we provided an exception to widen visibility, we can *hide* members we don't wish to keep available

Who doesn't like examples? Losers. Losers don't like examples.

Multiple inheritance

Diamonds aren't an Earl's best friend

Suppose you want a class to inherit properties from two completely different base classes.

- e.g. maybe one base class defines file IO, and another defines a list-based collection; you'd like a SerializableList that can be saved/loaded

In theory, that should be no problem, right? But it raises a few problems:

- What if both the collection and the file have an `insert(int)`? Which would the derived type inherit?

Even better, let's say that we have another base class, `Stringable`, that simply defines a `toString`, which the file and collection both override.

- We don't even need to take it that far. Remember how inheritance works: it creates a copy of the base type. In this case, how many copies of `Stringable` should be created for a single SerializableList?

The diamond problem

If you only have the initial problem (one class inherits the same name from two base classes), the solution doesn't have to be *too* painful.

- e.g. if `insert` is *ambiguous*, just be explicit: myList.File::insert(v) or myList.List::insert(v)
- Of course, that's a little ugly, so you might be better writing an `insert` for the SerializableList, that offloads to one or the other

But that doesn't solve the problem of one class, inheriting from two classes, which both inherit from the same single class. It's known as the *diamond problem*, because the dependencies are drawn as a diamond.

For that matter, we also still can't make effective use of polymorphism, because we don't have proper substitutability. We need more tools.

Types and inheritance

Before we get any further, we're going to need to make sure we understand how derived types actually work, and how they relate to their base types.

- If a Student is a derived class of a Person base class, then remember that that means a Student *is a* Person
- Does that mean a Person reference can receive a Student? How would the behaviour be affected by that?

I think we need another example.

Virtual functions

I can haz polymorphism?

The issue here is that the base class has no idea that some other class might come along and try redefining behaviour.

- When a variable is actually declared as Bird, why should the compiler scour the rest of the source code, trying to find hypothetical potential substitutions?
- For that matter, who's to say that, when you declare a variable as a Bird, it shouldn't *always* reliably behave as a Bird?

C++ offers a feature for letting you explicitly delineate which methods should be anticipated to be replaced

- The keyword is `virtual`
- Only the base type needs to declare the method virtual, but it doesn't hurt for the derived type as well

Normally, virtual function return types must match. The exception is for *covariant return types*. ... Bird Example Time?

Virtual destructors

(That sounds like a bad Atari 2600 game)

Ostensibly, based on what we've learned, we should be able to achieve proper polymorphism, but we've forgotten something.

Consider our previous problem, and think about what that might mean for destructors.

- Imagine if we had a simple array-based queue, and created a linked-list version as a derived type
  - If we were referencing it via the base queue type, then upon deallocation it wouldn't reclaim all of the heap memory!

To jump straight to the end: any time you use inheritance, if there's ever the possibility of *any* destructor being necessary, make them *all* virtual!

Final thought on virtual functions

Recall how, for normally-declared functions, with basic inheritance, you could still access a base class's members by explicitly specifying the scope?

- e.g. `Duck d; d.Bird::tweet();`

Well, being `virtual` doesn't interfere with that. You can still access the base type's version regardless.

Virtual base classes

I think we're ready to return to our *diamond problem*.

About half a dozen slides ago, we asked how many copies of the original base class would be created for the final derived class in the diamond.

- Since we bothered to ask, that's a hint: 2
- Minor inefficiency aside, imagine if that base type allocated memory, or started threads, or connected to an IO stream...

There's a pretty simple solution, though:

- For any derived type extending a base type (that might be shared with additional derived classes), prepend the keyword `virtual` before the access modifier
- This guarantees that only a single instance of a parent class will be created for a child class

Quick example? Just a trivial one...

Abstract types

What we've seen so far is great for when you want to make a new complete class based on another complete class.

But what if you only want to define a specification, or *part* of an implementation?

- A *pure virtual function* defines an abstract function
  - This is not the same as a forward declaration/prototype. This declares a virtual function header that must be implemented *in a derived class*
- A class that contains at least one abstract function is an *abstract base class*
  - Though it may define member variables, as well as some method implementations, it's still an incomplete class, and thus cannot be instantiated
- If a class consists *solely* of abstract functions, with no member variables or implementations, then it is an *interface class*
  - These act as a specification, or a trivial supertype

Probably best to look at another example.

Templates

Remember a couple of lectures ago, when we talked about using *function templates* to reuse functions in a more generic fashion (and thus avoid having to constantly rewrite redundant code just to accept different parameter types)?

Well, hopefully yes, because we're going to expand on that a little bit.

Template classes

We often encounter a similar problem when dealing with two scenarios: IO and collections. There are other reasonable ways around the IO issue, but collections can be a problem.

- It isn't generally feasible to write a separate version of a container/data structure for any thing you might want to hold
- There isn't really any useful supertype that would fit here, either
- Function templates aren't sufficient, because that doesn't help for member variables

However, we do have a reasonable solution: a *template class*

- If you put the `template` in front of the class, instead of the function, it applies to the whole class. This gives you an easy way to define generic member variables and functions

Honestly, the biggest concern is that you can't split up the specification and implementation as easily (and possibly shouldn't try).

Additional notes on templates

Ostensibly, we could also discuss *expression parameters* (which let you also provide values to parametrize template classes, for certain limited applications), *template specialization* (which lets you define special exceptions to parametrized templates, where special alternate implementations are necessary), and *partial template specialization* (where we provide specializations for at least one of the template parameters, but still require the rest to be filled in when called).

Heck, we could also talk about how to address bounded parametric types (bounded template specialization) in C++.

- Ahahahahahahaha
  - No
- This one isn't happening

For the rest, we might see if we have time at a later date. But for now we've seen more than enough to absorb in a single lecture.

Questions?

Comments?

- Aaaanyone looking forward to the Switch?