COSC 2P95

Ooh, OO!

Week 6

Brock University

Organizing information

Recall that we've already learned a method for organizing multiple pieces of data into a single coherent record: the *struct*.

But how do we integrate that with behaviours that are naturally (and inextricably) linked to those records?

- We could add a bunch of function pointers to each struct
  - Unless you're using pure C, please don't
- Suppose we were to write a function for each operation
  - The first parameter of each function could be a pointer to the struct
  - So long as we had all of those functions assembled, and each could dereference the pointers to access members of the structs, we could assign functionality to a struct and make it reasonable useable
  - Of course, that wouldn't be completely easy to use with multiple files
- We actually *can* add functions to structs, but that's generally a bad idea

So, what do we want to do?

Adding some class to our records

We can extend the idea of a struct into a *class*.

- A class represents a concept for an entity; it's the underlying idea behind a type
- A class defines both the state information, and the behaviours, that an *instance* of such a type would require
- Such an instance is known as an *object*

The power of *object orientation* — the support for using objects to store instance variables (member variables) and support methods (member functions) — lies in the ability to *model a system*. It's an entirely different approach to software design that facilitates software engineering and advanced design patterns.

Our first class

(but not, like, kindergarten. software class)

Declaring classes is *very* similar to how we made structs. We're just going to eventually be adding some additional interesting features.

- Let's look at the simplest of examples to start

Information hiding

Or encapsulation, or the Macarena, or whatever you want to call it...

Since Object Orientation deals with modelling, and establishing roles, etc., it makes sense to assign responsibilities for dealing with specific state information and behaviours.

That is, if you define a *Student* record, then it makes sense that a *Student* object would "know" that Student's name, and be best-suited to handle behaviours associated with Students (e.g. enrolling in courses). Further, oneo could argue that *only* Students should have direct knowledge of Student names.

- If we're going to say that certain member properties and behaviours *should* only be used by the classes that own them, then it would make sense to guarantee that only those classes *can* directly access them
  - To that end, we can employ *access modifiers*
  - We can explicitly define which members should be *public* for external use, and which should be *private*
  - How could we choose which members should be which?

Example time!

Constructors

When an object is first instantiated, it's expected that it will be 'made ready for use'; this means we need a *constructor*.

A constructor header is written similarly to a function's, but with the name of the class.

- If you don't provide a constructor, the compiler provides the zero-argument constructor
- Otherwise, you simply write a header that matches the signature of expected usage
  - You can write several, to match different signatures
- Member initializer lists can make for useful shorthand
- You have more options for both writing and using them if you stick with at least C++11

Note: Outside of C++11 (and higher), don't try to directly initialize member variables unless they're declared `static`.

- Though with C++11 you can, which can then be overruled by explicit assignments in constructors

Constructor chaining

Suppose you have one constructor that accepts one set of parameters, and another constructor that accepts another set. e.g. a Fraction that accepts a numerator, or numerator and denominator.

Much of the code between the two constructors would be the same, right? How do we avoid this redundancy?

- If using an older C++ standard, you'd separate as much of the behaviour as possible out into a third method, and have both constructors call that method
- If using C++11 or higher, we have the option to use *constructor chaining* (often called *delegated constructors*)
  - One constructor can call another (but do it from the member initializer list)

`this`

Remember how we said we *could* have behaviour apply to a struct? By simply also including a point to the struct?

Well, that's actually already (basically) what we've been doing! The compiler prepends an initial parameter that's a pointer to the object. This object can still be referenced as the pointer `this` (of course, members are accessed via `->`; you can even access private members).

- It can be handy if you want to use a parameter with the same name as a member variable
- You can also use this to return a reference to the same object, to facilitate cascading method calls on the same object

The first point's trivial, but maybe an example for the latter?

Static member variables and functions

So, by now, we understand how instance variables and methods are tied to objects, yes?

But what about when they're not?

- With the `static` keyword, we can attach variables and functions to a *class*, rather than as a member of an *object*
- Within the class, you can just use the name as normal; outside, you can access it by prepending `ClassName::`
- For static member functions, there's no *this* pointer!
- Typical uses include defining constants, standardized functions that only require parameters, or defining counters to be used across multiple instances of the same class (e.g. so each new instance can be incremented relative the last)

Destructors

Set classes aside for a moment. Suppose you declare a local `int` variable. When will that variable be deallocated?

The same is true when you declare an object. Similarly, if you dynamically allocate an object via `new`, you can `delete` it in the same way as any other dynamic variable.

- But... what about anything you might have dynamically allocated within the object itself? And what about any streams it might have internally opened?
  - Cross your fingers and hope nobody notices?

C++ provides you with a default *destructor*: a trivial block of code that executes upon deallocation of the object (whether due to automatic deallocation upon leaving the block, or explicit deallocation from deletion).

You can define your own destructor using ~.
Example!

Specification vs Implementation

All of our examples have been so short that there's really no way for them to be unreadable. But what happens when we start needing more elaborate classes? Sooner or later, we'll end up with somewhat of a 'sea of code'. We may still be able to read it, but when it comes to actually using tools, what matters is often just *how to use it*, rather than *how it works*.

The point: We need to be able to define usage separately from implementation. We've already seen something similar for functions.

- Actually, we can do mostly the same thing for classes. We simply separate our class declaration from our class definitions.
- It's not uncommon to use a header file for the declaration, and the accompanying `.cpp` file for the definition
  - It's good convention, for a class *Class*, to declare the header in `Class.h`, and to define the implementation in `Class.cpp`
  - That header than can easily be imported by other files

Quick example time? Quick example time.

Constant classes and member functions

If you declare an object as `constant`, then (after the constructor's initialization) you can't modify the member variables.

- This includes not only trying to directly change member variables, but also via updater methods
- Because of this, for the sake of safety, don't try invoking a member *function* of a constant object unless that function has `const` added between its parameter list and open brace (which indicates a promise to not make changes)

Since an object can also be treated as constant if you have a `const` reference to it, this can come up more often than you'd think. For that reason, it's not uncommon to declare accessors as `const`.

Why can't we be friends?

Why can't we be friends?

We understand why we (generally) employ information hiding. It's good practice. But, on occasion, you'll encounter the need for one class to have access to another's members, but that shouldn't preclude you from still being otherwise cautious with information.

C++ provides a mechanism for defining special exceptions to the visibility rules.

- You can list an external function header as a `friend` to indicate that the function can access that object's members
- If the function is within another class, you can simply identify the function's class by prepending `ClassName::`
- If you want to make a class's members avaialble to the entirely of another class, you can simply list the other class as a friend

Just be careful with the little details: there's no *this* in another function, class friendship isn't transitive, etc. (Example? Yay/nay?)

Anonymous objects

Just a minor point: if you need a temporary object to use for an intermediate stage (e.g. constructing an argument for, or a return from, a function), you can simply use the class name and the parameters to create an object.

- e.g. `return Dealie(3);` creates and returns a `Dealie` object

Interjection!

Who here writes their math like this?

`x=add(2,3); y=times(x,4); z=sub(x,y);`

No? Nobody? Why not? You can still read it, right?

Except it's cumbersome.

Similarly, for the sake of cleaner algorithms and shorter code, we'd like to consolidate and condense.

The solution is largely the same as what we do for written math.

Overloading operators

C++ actually allows us to overload operators — that is, we can define operators that can be used directly with objects (or other similar types). Note that most operator precedences still apply from their original usages, so be careful with expected behaviour if you're reusing operator symbols for completely different uses.

- You might prefer to define operators as member functions, or as friend functions
- Always be aware of what your return type is

Super-quick example? Maybe two?

Streamlining streams

Query: What does the << operator do?

- If you intend to eventually print out records, or possibly be able to read them in, you may wish to do the same trick that we've already been seeing from the beginning
- Just be *super* careful about the return type!
- (Also, for overloaded member function operators, you're defining relative the 'left operand', so think of what that would mean here)

Really quick example time!

Overloading typecasts

Though possibly not something you'd think of as an operator, you can also overload type conversions.

This can be good for, for example, converting a fraction or currency amount to a double; or reducing a countable value into an int.

- Put a space between `operator` and the type conversion
  - e.g. `operator int() {...`
  - Note the lack of a return type

Additional thoughts on overloading and operators

There's plenty more we can do with overloading operators (and a few more caveats to consider). Take a look here:

- `http://en.cppreference.com/w/cpp/language/operators`
- `http://en.cppreference.com/w/cpp/language/cast_operator`
- `http://en.cppreference.com/w/cpp/memory/new/operator_new`
- `http://en.cppreference.com/w/cpp/memory/new/operator_delete`

Copy constructors and elision

Do you remember what happened when we tried passing a struct to a procedure as an argument?

Effectively the same thing happens with objects. However, that can be particularly bad, considering several things, including the streams you might have open, as well as deeper-allocated structures.

- If desired, you can overload the copy operator
- Depending on the compiler, there could be some instances where copying will be *elided*

Additional Notes

Suppose you wanted to give a fraction to a procedure. Would 9 suffice?

- Actually, there's a good chance it would (and with C++11 and higher, you can get slightly more complicated than that)
- The issue that arises is how to handle unexpected/atypical types (e.g. a char)
  - e.g. if you wanted to write a *string* class that could either initialize on a passed `char` array, or allocate a blank array of integral size, how should it handle a single `char`?

Also, you can overload the *assignment* operator, but we likely won't need to. It's rarely handled differently from *copy*, and the only particularly interesting thing to note is that you might get unexpected behaviour if you're assigning an object to itself.
Also, do we know the difference between a *shallow copy* and a *deep copy*?

- Because we should totally know that

Questions?

Comments!

- EXCITEMENT!