CS106L Handout #23
Fall 2007 December 5, 2007

Advanced C++ Topics

Introduction

Over the past ten weeks we've explored many aspects of C++, ranging from simple use of the IOStream library to advanced operator overloading and functional programming with the STL. Even so, we have hardly taken our first steps into the wondrous world of C++. There are numerous advanced programming techniques and language features we did not have time to cover in this class. This handout covers two of these important C++ topics – exception handling and object memory management.

A Simple Problem

Up to this point, all of the programs you've written have proceeded in a linear fashion – they begin inside a special function called `main`, then proceed through a chain of function calls and returns until (hopefully) ultimately returning. While this is perfectly acceptable, it rests on the fact that each function, given its parameters, can perform a meaningful task and return a meaningful value. However, in some cases this simply isn't possible. Consider, for example, this function:

```
string ConcatNCopies(const string &input, int numRepeats)
{
    string result;
    for(int i = 0; i < numRepeats; i++)
        result += input;
    return result;
}
```

At first glance, this function seems totally normal, and in almost all cases it will perform its stated task. However, there's a slight problem – what if `numRepeats` is negative? In the current implementation, since the inner loop will never execute, `ConcatNCopies` will simply return the empty string. While this won't crash at runtime, it is nonetheless a problem because it is a *silent error*. The function didn't work correctly, but the calling function has no way of knowing this.

There are several questions we have to ask ourselves at this point. First, is this even a problem? The answer should almost certainly be yes. If we write code that can fail without signaling that an error occurred, then the program could enter an invalid and potentially unstable state without knowing the cause. A truly robust piece of software should be able to detect and handle problems so that it can respond before they snowball into fully-fledged runtime errors. This then leads into the second question: how should we address the problem? Here, we have several options, many of which unfortunately introduce complications of their own. The first option, and the one you've encountered up to this point in CS106, is to call a function akin to the CS106 `Error` to report the error and terminate the problem. While this will indeed give the user notification of the problem, it is not particularly elegant. After all, a call to `Error` abruptly exits without giving the rest of the program or the user a chance to respond. And, in the case of functions like `ConcatNCopies`, it seems drastic to completely stop execution simply because of a bad parameter. Rather, we'd hope that somehow the calling function could identify the problem, correct it, then continue execution.

This approach suggests a second option, one common in pure C – *sentinel values*. The idea is to have functions return special values indicating "this value is invalid because the function failed to execute correctly." Initially, this may seem like a good idea – it reports the error and gives the calling function a change to respond. However, there are several major problems with this approach. First, it means that we have to designate a special return value that the function must never return except when an error occurs. In the case of `string`, this is infeasible since if we mark some `string` value as an error (let's call it `INVALID_PARAM`), then the user could write code like this:

```
ConcatNCopies(INVALID_PARAM, 1);
```

And the code would return `INVALID_PARAM` (since concatenating `INVALID_PARAM` with itself one time yields `INVALID_PARAM`) even though the function succeeded. Second, unless we define a standardized system for reserving invalid return values, we may end up with a huge number of functions, each with their own special "error" return codes, that could lead to problems if we checked the function's return value against the wrong sentinel. Imagine the chaos of constants `STRING_ERROR`, `INVALID_STRING`, and `INVALID_PARAM`, each corresponding to different functions. Third, this approach makes the code bulky, since whenever we'd call `ConcatNCopies`, we'd have to write something like this:

```
string result = ConcatNCopies(myString, myInteger);
if(result == INVALID_PARAM) { /* ... handle error ... */ }
```

This is unsightly and, above all, needlessly complicates the code.

Yet another option might be to change the function declaration so that it looks like this:

```
bool ConcatNCopies(const string &input, int numRepeats, string &output);
```

Instead of returning a value, instead we fill in a `string` specified as a reference parameter and then return whether the operation succeeded. This too has its flaws. Suppose our function is an overloaded operator. We cannot simply change the number of parameters to an overloaded operator, since all overloaded operators except for `operator ()` have a fixed number of arguments, and consequently this approach will not work. All of the approaches we've tried so far have some flaw, so how are we to resolve this problem?

Exception Handling

The reason the above example is such a problem is that the normal C++ function-call-and-return system simply isn't robust enough to communicate errors back to the calling function. To resolve this problem, C++ provides language support for an error messaging system called *exception handling* that completely bypasses function-call-and-return. If an error occurs inside a function, rather than returning a value, you can report the problem (called an *exception*) to the exception handling system to automatically jump to the proper error-handling code.

The C++ exception handling system is broken into three parts – `try` blocks, `catch` blocks, and `throw` statements. `try` blocks are simply regions of code where you inform the C++ compiler that a runtime exception might occur. To declare a `try` block, you simply write the keyword `try`, then surround the appropriate code in curly braces. For example, the following code shows off a `try` block:

```
try
{
    cout << "I'm in a try block!" << endl;
}
```

Inside of a `try` block, code executes as normal and jumps to the code directly following the `try` block once finished. However, at some point inside a `try` block your program might run into a situation from which it cannot normally recover – for example, a call to `ConcatNCopies` with a negative argument. To report an error, use the `throw` keyword to "throw" the exception into the nearest matching "catch" clause. Like `return`, `throw` accepts a single parameter that indicates an object to throw so that when handling the exception your code has access to extra information about the error. Unlike other languages like Java, in C++ you're allowed to throw objects of any type as exceptions, not just specialized classes. Thus all of the following pieces of code are legal:

```
throw 0;
throw new vector<double>;
throw 3.14159;
```

When you throw an exception, it can be caught by a `catch` clause specialized to catch that error. `catch` clauses are defined like this:

```
catch(ParameterType param)
{
    /* Error-handling code */
}
```

`catch` blocks must directly follow `try` blocks, and it's illegal to declare one without the other. Since catch clauses are specialized for a single type, it's legal (and recommended) to have cascading catch clauses, each designed to pick up a different type of exception. For example, here's code that catches exceptions of type `int`, `vector<int>`, and `string`:

```
try
{
}
catch(int myInt)
{
}
catch(vector<int> &myVector)
{
}
catch(string &myString)
{
}
```

Now, if the code inside the `try` block throws an exception, control will pass to the correct `catch` block. When defining cascading `catch` blocks as shown here, they will be evaluated in the order they're declared. Thus if you have two different `catch(int)` blocks, only the first will execute when catching an exception. You can also define a special "catch-all" catch clause that catches any type of exception by writing `catch(...)`. This catch clause will catch any exception, but because you cannot determine what type of exception it caught, you will not have access to the object that was thrown.

Let's return to our earlier example with `ConcatNCopies`. We want to signal an error in case the user enters an invalid parameter, and to do so we'd like to use exception handling. The question, though, is what type of object we should throw. While we can choose whatever type of object we'd like, C++ provides a header file, `<stdexcept>`, that defines several classes useful as thrown objects that let us specify what error triggered the exception.* One of these, `invalid_argument`, is ideal for the situation. `invalid_argument` accepts in its constructor a `string` parameter containing a message representing what type of error occurred, and has a member function called `what` that returns what the error was. We can thus rewrite the code for `ConcatNCopies` as

```
string ConcatNCopies(const string &input, int numCopies)
{
    if(numCopies < 0)
        throw invalid_argument("Number of copies must be positive.");

    string result;
    for(int i = 0; i < numRepeats; i++)
        result += input;
    return result;
}
```

Notice that while the function itself does not contain a `try`/`catch` system, it nonetheless has a `throw` statement. If this statement is executed, then C++ will step backwards through all calling functions until it finds an appropriate `catch` statement. If it doesn't find one, then the program will halt with a runtime error. Now, we can write code using `ConcatNCopies` that looks like this:

```
try
{
    cout << ConcatNCopies(myString, myInteger) << endl;
}
catch(invalid_argument &problem)
{
    cout << problem.what() << endl; // Prints out the error message.
}
```

A Word on Efficiency

Exception handling should be used only for *exceptional* circumstances – errors out of the ordinary that necessitate a major change in the flow of control. While you can use exception handling as a fancy form of function call and return, it is highly suggested that you avoid doing so. Throwing an exception is much, much slower than returning a value because of the extra bookkeeping required, so be sure that you're only using the exception handling system for serious program errors.

Programming with Exception Handling

While exception handling is a robust and elegant system, it has several sweeping implications for your C++ code. Most notably, when using exception handling, you must treat your code as though it might throw an exception at any point. In other words, you can never assume that an entire code block will be completed on its own, and should be prepared to handle cases where control breaks out of your functions at inopportune times.

For example, consider the following assignment operator code for a `Derived` object inheriting from `Base`:

* The `<stdexcept>` header exports a hierarchy of exceptions that encompass a wide number of error types. Be sure to read into them if you pursue this material in-depth.

```
Derived& Derived::operator = (const Derived &other)
{
    if(this != &other)
    {
        clear();
        Base::operator =(other);
        copyOther(other);
    }
    return *this;
}
```

Normally, this code should work without problems, but when you add exception handling to the mix this code is incredibly dangerous. Since the code may throw an exception at any point, there's no way of knowing that the entire body of the `if` statement will execute. In fact, a perfectly valid run of this code might be for the `clear` command to execute but have the `Base::operator =` function throw an exception. In this case, we have a serious problem, since we've cleared out the object without replacing it with any meaningful data. As a result, later in our program, if we try to use the malformed object, we'll probably get runtime errors. Clearly, this is a problem.*

If we maintain the mentality that any piece of code can throw an exception at any point, it would be virtually impossible to write exception-safe code. However, it's much easier if we could somehow make assumptions about what sorts of functions will and will not throw exceptions. C++ professionals tend to categorize C++ code into three classes based on their relative safety when mixed with exceptions: basic exception-safe code, strong exception-safe code, and nothrow exception-safe code. *Basic exception-safe code* is code that is minimally exception-safe, meaning that if an exception aborts it midway through execution, the object is not corrupted. Note that this does *not* mean that the object is in its original state, only that it's not corrupted. *Strong exception-safe code* promises that any exceptions at runtime will leave the program state unchanged. That is, any data existing before the function call will remain that way even if function flow aborts midway. Finally, *nothrow exception-safe code* is code that guarantees it will not throw any exceptions and that if it does, it is a serious error that should shut down the program immediately.

Writing code that meets even the weakest of these guarantees can be incredibly difficult, and indeed writing exception-safe code is one of the most challenging aspects of professional C++. We've ignored exception handling in both CS106X and CS106L for this very reason – it greatly complicates even the simplest of tasks. Nonetheless, exception handling is incredibly useful and is deeply embedded into C++. To learn more about writing exception-safe code, you should be sure to consult a reference.

Object Memory Management and RAII

C++'s memory model is best described as "dangerously efficient." Unlike other languages like Java, C++ does not have a "garbage collector" and consequently you must manually allocate and deallocate memory. At first, this might seem like a simple task – just `delete` anything you allocate with `new`, and make sure not to `delete` something twice. However, it can be quite difficult to keep track of all of the pointers you've allocated in a program. After all, you probably won't notice any symptoms of memory leaks unless you run your programs for hours on end, and in all probably will have to use a special tool to check memory usage. You can also run into trouble where two classes each have a pointer to a shared

* In fact, it's such a serious problem that in some cases you have to radically restructure your code to make sure it's exception-safe. Writing exception-safe assignment operators is one such challenge, and one of the only valid ways to make an exception-safe assignment operator is to switch to another strategy called "copy-and-swap" to perform the assignment. Consult a reference for more information on copy-and-swap.

object. If one of the classes isn't careful and accidentally `delete`s the memory while the other one is still accessing it, you can get some particularly nasty runtime errors where seemingly valid data has been corrupted. The situation gets all the more complicated when you introduce exception-handling into the mix, where the code to `delete` allocated memory might not be reached because of a runtime exception sometime earlier in the code.

In some cases having a high degree of control over memory management can be quite a boon to your programming, but most of the time it's simply a hassle. What if we could somehow get C++ to manage our memory for us? While building a fully-functional garbage collection system in C++ would be just short of impossible, using only basic C++ concepts it's possible to construct an excellent approximation of automatic memory management. The trick is to build *smart pointers*, objects whose constructors acquire a resource allocated by `new` and whose destructors clean up that resource through a call to `delete`. That is, when the objects are constructed, they wrap a newly-allocated pointer inside an object shell that cleans up the mess when the object goes out of scope. Combined with features like operator overloading, it's possible to create slick smart pointers that look almost exactly like true C++ pointers, but that know when to free unused memory.

The C++ Standard Library exports the `auto_ptr` type, a smart pointer that accepts in its constructor a pointer to dynamically-allocated memory and whose constructor calls `delete` on the resource.[*] `auto_ptr` is a template class whose template parameter indicates what type of object the `auto_ptr` will "point" at. For example, an `auto_ptr<string>` is a smart pointer that points to a `string`. Be careful – if you write `auto_ptr<string *>`, you'll end up with an `auto_ptr` that points to a `string *`, which is similar to a `string **`. Through the magic of operator overloading, you can use the regular dereference and arrow operators on an `auto_ptr` as though it were a regular pointer. For example, here's some code that dynamically allocates a `vector<int>`, stores it in an `auto_ptr`, and then adds an element into the `vector`:

```
/* Have the auto_ptr point to a newly-allocated vector<int>.  Constructor
   is explicit, so we must use parentheses. */
auto_ptr<vector<int> > managedVector(new vector<int>);

managedVector->push_back(137); // Add 137 to the end of the vector.
(*managedVector)[0] = 42; // Set element 0 by dereferencing the pointer.
```

While in many aspects `auto_ptr` acts like a regular pointer with automatic deallocation, `auto_ptr` is fundamentally different from regular pointers in assignment and initialization. Unlike objects you've encountered up to this point, assigning or initializing an `auto_ptr` to hold the contents of another destructively modifies the original `auto_ptr`. Consider the following code snippet:

```
auto_ptr<int> one(new int);
auto_ptr<int> two;
two = one;
```

After the final line executes, `two` will be holding the resource originally owned by `one`, and `one` will be empty. During the assignment, `one` relinquished ownership of the resource and cleared out its state. Consequently, if you use `one` from this point forward, you'll run into trouble because it's not actually holding a pointer to anything. While this is highly counterintuitive, it has several advantages. First, it assures that there can be at most one `auto_ptr` pointing to a resource, which means that you don't have

[*] Note that `auto_ptr` calls `delete`, not `delete []`, so you cannot store dynamically-allocated arrays in `auto_ptr`. If you want the functionality of an array with automatic memory management, use a `vector`.

to worry about the contents of an `auto_ptr` being cleaned up out from underneath you by another `auto_ptr` to that resource. Second, it means that it's safe to return `auto_ptr`s from functions without the resource getting cleaned up. When returning an `auto_ptr` from a function, the original copy of the `auto_ptr` will transfer ownership to the new `auto_ptr` during return-value initialization, and the resource will be transferred safely.

As a consequence of the "`auto_ptr` assignment is transference" policy, you must be careful when passing an `auto_ptr` by value to a function. Since the parameter will be initialized to the original object, it will empty the original `auto_ptr`. Similarly, you should not store `auto_ptr`s in STL containers, since when the containers reallocate or balance themselves behind the scenes they might assign `auto_ptr`s around in a way that will trigger the object destructors.

For reference, here's a list of the member functions of the `auto_ptr` template class:

`auto_ptr (Type *resource)`	`auto_ptr<int> ptr(new int);` Constructs a new `auto_ptr` wrapping the specified pointer, which must be from dynamically-allocated memory.

`auto_ptr(auto_ptr &other)`	`auto_ptr<int> one(new int);`
	`auto_ptr<int> two = one;`
	
	Constructs a new `auto_ptr` that acquires resource ownership from the `auto_ptr` used in the initialization. Afterwards, the old `auto_ptr` will not encapsulate any dynamically-allocated memory.

`T& operator *() const`	`*myAutoPtr = 137;` Dereferences the stored pointer and returns a reference to the memory it's pointing at.
`T* operator-> () const`	`myStringAutoPtr->append("C++!");` References member functions of the stored pointer.
`T* release()`	`int *regularPtr = myPtr.release();` Relinquishes control of the stored resource and returns it so it can be stored in another location. The `auto_ptr` will then contain a `NULL` pointer and will not manage the memory any more.

`void reset(T *ptr = NULL)`	`myPtr.reset();`
	`myPtr.reset(new int);`
	
	Releases any stored resources and optionally stores a new resource inside the `auto_ptr`.

An interesting trick with `auto_ptr` is to make a `const auto_ptr` to a resource. Since the `auto_ptr` is `const`, it can't be assigned (which would destructively modify it) or otherwise changed, so the resource it wraps is guaranteed not to be transferred or deallocated until the object is cleaned up. This trick is quite useful for ensuring that dynamically-allocated memory isn't cleaned up or orphaned until the end of a function call or object lifetime.

Of course, dynamically-allocated memory isn't the only C++ resource that can benefit from object memory management. For example, if you were writing C++ code to interface with older C code that did file access through the old-style `FILE *` system, you'd need to make sure to manually set up and clean up the file handles. In fact, the system of having objects manage resources through their constructors and destructors is commonly referred to as *resource acquisition is initialization*, or simply RAII.

Exceptions and Smart Pointers

Up to this point, smart pointers might seem like a curiosity, or perhaps a useful construct in a limited number of circumstances. However, when you introduce exception handling to the mix, smart pointers will be invaluable. In fact, in professional code where exceptions can be thrown at almost any point, smart pointers have all but replaced regular C++ pointers.

Consider the following function:

```
int EvaluateExpression(const string &expression)
{
    /* Assume ParseExpression returns dynamically-allocated memory. */
    Expression *e = ParseExpression(expression);
    int result = e->evaluate();
    delete e;
    return result;
}
```

Nothing in this code seems all that out-of-the-ordinary, and it seems like it won't leak any resources since the dynamically-allocated memory returned by `ParseExpression` will be cleaned up by the call to `delete e`. Unfortunately, the above code is a veritable recipe for disaster. After all, what happens if `e->evaluate()` throws an exception? If so, control breaks out of the function before the call to `delete e`, and the memory will be orphaned. To fix this problem, we can replace the regular `Expression *` pointer with an `auto_ptr<Expression>`. That way, even if `e->evaluate()` throws an exception, the `auto_ptr` destructor will correctly clean up the dynamically-allocated memory, preventing a problem. The new code thus looks like this:

```
int EvaluateExpression(const string &expression)
{
    auto_ptr<Expression> e(ParseExpression(expression));
    int result = e->evaluate();
    return result;
}
```

In general, if you ever dynamically allocate memory using `new`, you should strongly consider wrapping it in an `auto_ptr`, since it makes your code more exception-safe and lets you leave the deallocation to the `auto_ptr` destructor.

Exceptions in Constructors

One of the single most vexing and complicated parts of the C++ language arises when constructors throw exceptions. Suppose you have the following definition of the `MyClass` object:

```
class MyClass
{
    public:
        MyClass()
        {
            throw 0; // Throw an arbitrary exception.
        }
        ~MyClass()
        {
            cout << "MyClass destructor invoked." << endl;
        }
    private:
        int myInt;
        string myString;
        vector<int> myVector;
};
```

What will happen if we write the following code?

```
try
{
    MyClass mc;
}
catch(...)
{
}
```

When the `MyClass` object is instantiated, its constructor will invoke, throwing the number zero as an exception and passing control to the "catch-all" handler. However, the `MyClass` destructor will not invoke for the object `mc`, even though it has gone out of scope. This is a deliberate design decision and requires a bit of explanation. Suppose that if the `MyClass` constructor threw an exception, the `MyClass` destructor would clean up the `mc` object. This would mean that the destructor for `MyClass` would invoke, but there would be no guarantee that `mc` had been completely initialized. `MyClass` might, for example, contain several pointers that hadn't been initialized, and if the destructor were to `delete` them, you could very well get runtime errors inside the `catch` block. Consequently, C++ will not call the destructor of a class for which the constructor threw an exception. This does not mean, however, that all of `mc`'s data members will not be cleaned up. Although the `MyClass` object itself hasn't finished being constructed, its data members have been initialized to contain their default values (or possibly special values specified in the initializer list). Thus C++ *will* invoke the destructors for all of the `MyClass` data members that have been initialized. This is important – the data members must have been initialized for their destructors to invoke. Consider the following code snippet:

```
MyOtherClass::MyOtherClass() : myDataMember1(one), myDataMember2(two)
{
    /* ... */
}
```

If in the initializer list the `myDataMember1` constructor throws an exception, C++ will not call the destructor for any of `MyOtherClass`'s data members, since none of them have actually been initialized yet. In other words, C++ will only invoke object destructors if their constructors have completely finished running.

That C++ doesn't invoke destructors for incompletely-formed objects can be a great source of trouble when working with pointers. For example, consider the following implementation of a `BinaryTreeNode` class:

```
class BinaryTreeNode
{
    public:
        BinaryTreeNode(BinaryTreeNode *left, BinaryTreeNode *right)
            : leftChild(left), rightChild(right)
        {
            /* Initialization code */
        }
        ~BinaryTreeNode()
        {
            delete leftChild;
            delete rightChild;
        }
        /* Other member functions */
    private:
        BinaryTreeNode *leftChild, *rightChild;
};
```

If the initialization code in `BinaryTreeNode`'s constructor throws an exception, then C++ will not invoke the `BinaryTreeNode` destructor, but will instead call the default destructors for each of the children. However, since the children are pointers, there *is* no default destruction code, so if `leftChild` and `rightChild` point to dynamically-allocated memory, the memory will simply be lost. To resolve this problem, we can change `leftChild` and `rightChild` from simple pointers into smart pointers. That way, if the constructor throws an exception, the smart pointer destructors can clean up any leftover memory. This has the added advantage that we don't have to worry about forgetting to clean up the resources in the destructor. The new, exception-safe code thus looks like this:

```
class BinaryTreeNode
{
    public:
        BinaryTreeNode(BinaryTreeNode *left, BinaryTreeNode *right)
            : leftChild(left), rightChild(right)
        {
            /* Initialization code */
        }
    /* No destructor */
    private:
        auto_ptr<BinaryTreeNode> leftChild, rightChild;
};
```

This is the beauty of object memory management – it makes your code cleaner, more concise, and less error-prone.

More to Explore

This handout has barely scratched the surface of exception handling and smart pointers, so be sure to consult a reference for more information. Here are some interesting topics to explore to get you started:

1. **The Boost Smart Pointers**: While `auto_ptr` is useful in a wide variety of circumstances, in many aspects it is limited. Only one `auto_ptr` can point to a resource at a time, and `auto_ptr`s cannot be stored inside of STL containers. The Boost C++ libraries consequently provide a huge number of smart pointers, many of which employ considerably more complicated resource-management systems than `auto_ptr`. Since many of these smart pointers are likely to be included in the next revision of the C++ standard, you should be sure to read into them.
2. **The pImpl Idiom**: Writing exception-safe copy constructors and assignment operators is frustratingly difficult, and one solution to the problem is to use the *pImpl idiom*. pImpl stands for “pointer to implementation,” and means that rather than having a class have actual data members, instead it has a smart pointer to an “implementation object” that contains all of the class data. pImpl also can be used to reduce compile times in large projects. Since it arises so frequently in professional code, you should certainly look into pImpl if you plan on seriously pursuing C++.
3. **Exception Specifications**: You can explicitly mark what types of exceptions C++ functions are legally allowed to throw. While this introduces a bit of overhead into your program, exception specifications can greatly reduce the number of bugs in your code by identifying patches of exception safety in an otherwise uncertain world.
4. **Nothrow `new`**: The normal C++ `new` operator throws a `bad_alloc` exception if it is unable to obtain the needed amount of memory. In case this isn't what you want, you can use the *nothrow `new`*, a version of the `new` operator that cannot throw exceptions. The syntax is `new (nothrow) DataType`.
5. **`assert`**: Functions like the CS106 `Error` that halt execution at runtime are somewhat inelegant because they don't give your program a chance to respond to an error. However, when designing and testing software, `Error`-like functions can be useful to pinpoint the spots where errors occur in your code. For this purpose, C++ inherits the C `assert` macro, which evaluates an expression and halts execution if the value is false. However, unlike `Error`, when compiling the program in release mode, `assert` is disabled, so you can smoke out bugs during debug development, then leave error-handling in release mode to the exception-handling system. `assert` is exported by the `<cassert>` header, and might be worth reading in to.

Bjarne Stroustrup (the inventor of C++) wrote an excellent introduction to exception safety, focusing mostly on implementations of the C++ Standard Library. If you want to read into exception-safe code, you can read it online at http://www.research.att.com/~bs/3rd_safe.pdf.

Practice Problems

1. Write an `AutoIntPtr` class that acts a an `auto_ptr` for `int`s. *(Note: while you might be tempted to generalize it to a template version, there are all sorts of issues to consider with `auto_ptr`s of derived and base types. If you're interested in writing your own smart pointers, refer to a reference.)*
2. What happens if you write a `catch(...)` clause before a `catch(int)` clause?
3. The Java programming language has support for exceptions, but unlike C++, it produces compile-time errors if you call a function that might throw an exception without providing the proper `catch` block. Why do you think that C++ doesn't enforce this restriction?
4. Why might throwing an exception take longer than returning a value? *(Hint: think about constructors, and how C++ might locate an appropriate catch block)*

Final Thoughts

It's been quite a trip since we first started with the IOStream library almost three months ago. You now know how to program with the STL, write well-behaved C++ objects, and even implement functional programming constructs inside the otherwise imperative/object-oriented C++ language. But despite the immense volume of material we've covered this quarter, CS106L is only the tip of the iceberg when it comes to C++. There are volumes of articles and books out there that cover all sorts of amazing tips and tricks when it comes to C++, and by taking the initiative and exploring what's out there you can hone your C++ skills until problem solving in C++ transforms from "how do I solve this problem?" to "which of these many options is best for solving this problem?"

C++ is an amazing and beautiful language. It has some of the most expressive syntax of any modern programming language, and affords an enormous latitude in programming styles. Of course, it has its flaws, as critics are eager to point out, but it is undeniable that C++ is an incredibly important and useful language in the modern programming age.

I hope that you've enjoyed CS106L as much as I have. This class has been a big experiment, and I hope that you're satisfied with the results. We'll be offering the class again next quarter, so you may be the first class in a new Stanford computer science tradition!

Have fun with C++, and by all means, if you ever have any questions, feel free to email me.

Enjoy!