Introduction to Exception Handling

Introduction

When writing C++ code, at some point you'll run into a situation where if a specific condition isn't met, a function cannot continue execution. For example, if your code expects a specific data file to be formatted correctly and discovers that the file is malformed, chances are you cannot proceed with the current operation. Also, you might find that you've allocated so many objects that the memory manager cannot satisfy a call to `new`. In these scenarios, the normal function-call-and-return mechanism is not robust enough to signal and report errors and you will have to rely on the *exception handling system*, a C++ language feature that redirects program control in case of emergencies.

Exception handling is an immensely complicated subject and it would not be unreasonable to claim that we could spend an entire quarter simply talking about its nuances. However, to understand the C++ language, you should have at least a basic exposure to exception handling. This handout introduces the fundamentals of exception handling and serves as a launching point for a further exploration of exception handling topics.

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 hitting the `return 0` statement at the end of `main`. 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(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 is 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. A truly robust piece of software should be able to detect and handle problems so that it can respond before the errors snowball into a runtime crash. 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 program. 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 meaning "this value indicates that the function failed to execute correctly." Initially, this may seem like a good idea – it reports the error and gives the calling function a chance 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. If you checked the return value against the wrong sentinel, your code would appear totally correct but would never respond to errors. 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. Finally, if the programmer doesn't check the return value against this sentinel (and experience shows that this is overwhelming the case), then the program will continue with garbage data, the exact problem we were trying to avoid in the first case.

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

```
bool ConcatNCopies(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. Unfortunately, this runs into the exact same problem as before, since you can entirely ignore the return value and wind up proceeding with invalid data. Also, later in the quarter when we cover operator overloading, we'll see examples of functions where we cannot legally change the number of arguments. Clearly, all of these systems are flawed, and we will need to use a new language feature to resolve our 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 to the exception handling system to 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 pieces of code designated as regions that runtime errors 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. When this occurs, you can report the error by using 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. For example, here are three statements that each throw objects of different types:

```
throw 0;                        // Throw an int
throw new vector<double>;       // Throw a vector<double> *
throw 3.14159;                  // Throw a double
```

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 */
}
```

Here, `ParameterType` represents the type of variable this `catch` clause is capable of catching. `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)
{
   // If the code throws an int, execution continues here.
}
catch(vector<int> &myVector)
{
   // Otherwise, if the code throws a vector<int>, execution resumes here.
}
catch(string &myString)
{
   // Same for string
}
```

Now, if the code inside the `try` block throws an exception, control will pass to the correct `catch` block. You can visualize exception handling as a room of people and a ball. Only the person holding the ball can talk, and normally one person will keep speaking the whole time. If, however, the speaker wants to throw the ball, she can toss it to one of the many catchers. The person who catches the ball then talks a bit, and hands the ball back to the original speaker.

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 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(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
{
      string result = ConcatNCopies(myString, myInteger);
      cout << "The result was: " << result;
      cout << "It has length " << result.length() << endl;
}
catch(invalid_argument &problem)
{
    cout << problem.what() << endl; // Prints out the error message.
}
cout << "Yay!  We're done." << endl;
```

Here, if `ConcatNCopies` encounters an error and throws an exception, control will jump out of the `try` block into the `catch` clause specialized to catch objects of type `invalid_argument`. Otherwise, code continues as normal in the `try` block, then skips over the catch clause to print “Yay! We're done.”

There are several things to note here. First, if `ConcatNCopies` throws an exception, control immediately breaks out of the `try` block and jumps to the `catch` clause. Unlike the problems we had with our earlier approach to error handling, here, if there is a problem in the `try` block, we're guaranteed that the rest of the code in the `try` block will not execute, preventing runtime errors stemming from malformed objects. Second, if there is an exception and control resumes in the `catch` clause, once the `catch` block finishes

running, control does not resume back inside the `try` block. Instead, control resumes directly following the `try`/`catch` pair, so the program above will print out "Yay! We're done." once the `catch` block finishes executing. While this might seem unusual, remember that the reason for exception handling in the first place is to halt code execution in spots where no meaningful operation can be defined. Thus if leaves a `try` block, chances are that the rest of the code in the `try` could not complete without errors, so C++ does not provide a mechanism for resuming program control. Third, note that we caught the `invalid_argument` exception by reference (`invalid_argument &` instead of `invalid_argument`). For somewhat technical reasons, it's always best to catch exceptions by reference, though it's by no means a requirement.

`catch(...)`

Normally, `catch` blocks must specify the type of object they want to catch. However, if you don't care about what type of object was thrown, or simply want to provide a "last-resort" catch clause in case something goes wrong with your program, you can use the special syntax `catch(...)`. The `catch(...)` clause is a "catch-all" block that will catch any type of exception. However, since you do not know what type of object you're catching, you cannot access the thrown object inside a `catch(...)` clause.

When defining cascading `catch` blocks, 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. Similarly, if you have a `catch(int)` block and a `catch(...)` block, provided that the `catch(int)` block precedes the `catch(...)` block, the `catch(int)` block will execute instead of the more general `catch(...)`.

A Word on Scope

Exception handling is an essential part of the C++ programming language because it provides a system for recovering from serious errors. Thus, 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.

Also, the exception handling system will *only* respond when manually triggered. Unless a code snippet explicitly `throw`s a value, a `catch` block cannot respond to it. This means that you cannot use exception handling to prevent your programming from crashes from segmentation faults or other pointer-based errors, since pointer errors result in operating-system level process termination, not C++-level exception handling.

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 function:

```
void SimpleFunction()
{
    char *myCString = new char[128];
    DoSomething(myCString);
    delete [] myCString;
}
```

Here, we allocate space for a C string, pass it to a function, then deallocate the memory. While this code seems totally safe, when you introduce exceptions into the mix, this code can be very dangerous. What happens, for example, if `DoSomething` throws an exception? In this case, control would jump to the nearest `catch` block and the line `delete [] myCString` would never execute. As a result, our program will leak 128 bytes of memory. If this program runs over a sufficiently long period of time, eventually we will run out of memory and our program will crash. Later in this handout, we'll see a way to resolve this problem, but nonetheless it should be apparent that exceptions can complicate your program layout.

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 program state will not be corrupted (that is, global variables and local objects will not contain garbage values). Note that this does *not* mean that the program 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 at all 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++. For the most part, we will ignore exception safety in CS106L because it greatly complicates even the simplest of tasks. Nonetheless, exception handling is incredibly useful and is deeply embedded into C++. The rest of this handout addresses one of the major techniques you can use to write exception-safe code, but for a full treatment of exception handling you should consult a reference textbook.

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 probability 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 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 an exception.

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 that acquire a resource when created and that clean up the resource when destroyed. 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++ header file `<memory>` 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 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.

---

* 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`.

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_ptrs in STL containers, since when the containers reallocate or balance themselves behind the scenes they might assign auto_ptrs 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.

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.

Let's suppose you're given the following linked list cell struct:

```
struct nodeT
{
    int data;
    nodeT *next;
};
```

Now, consider this function:

```
nodeT *GetNewCell()
{
    nodeT *newCell = new nodeT;
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell;
}
```

This function allocates a new `nodeT` cell, then tells it to hold on to the value returned by `SomeComplicatedFunction`. If we ignore exception handling, this code is totally fine, provided of course that the calling function correctly holds on to the `nodeT *` pointer we return. However, when we add exception handling to the mix, this function is a recipe for disaster. What happens if `SomeComplicatedFunction` throws an exception? Since `GetNewCell` doesn't have an associated `try` block, the program will abort `GetNewCell` and search for the nearest `catch` clause. Once the `catch` finishes executing, we have a problem – in the first line we allocated a `nodeT` object, but we haven't cleaned it up. Worse, since `GetNewCell` is no longer running, we've lost track of the `nodeT` entirely, and the memory is orphaned.

Enter `auto_ptr` to save the day. Suppose we change the declaration `nodeT *newCell` to `auto_ptr<nodeT> newCell`. Now, if `SomeComplicatedFunction` throws an exception, we won't leak any memory since when the `auto_ptr` goes out of scope, it will reclaim the memory for us. Wonderful! Of course, we also need to change the last line from `return newCell` to `return newCell.release()`, since we promised to return a `nodeT *`, not an `auto_ptr<nodeT>`. The new code is printed below:

```
nodeT *GetNewCell()
{
    auto_ptr<nodeT> newCell(new nodeT);
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell.release(); // Tell the auto_ptr to stop managing memory.
}
```

This function is now wonderfully exception-safe thanks to `auto_ptr`. However, we can make this code even safer by using the `auto_ptr` in yet another spot. What happens if we call `GetNewCell` but don't store the return value anywhere? For example, suppose we have a function like this:

```
void SillyFunction()
{
    GetNewCell(); // Oh dear, there goes the return value.
}
```

When we wrote `GetNewCell`, we tacitly assumed that the calling function would hold on to the return value and clean the memory up at some later point. However, it's totally legal to write code like `SillyFunction` that calls `GetNewCell` and entirely discards the return value. This leads to memory leaks, the very problem we were trying to solve earlier. Fortunately, through some creative use of `auto_ptr`, we can eliminate this problem.

Consider this modified version of `GetNewCell`:

```
auto_ptr<nodeT> GetNewCell()
{
    auto_ptr<nodeT> newCell(new nodeT);
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell; // See below
}
```

Here, the function returns an `auto_ptr`, which means that the returned value is itself managed. Now, if we call `SillyFunction`, even though we didn't grab the return value of `GetNewCell`, because `GetNewCell` returns an `auto_ptr`, the memory will still get cleaned up.

More to Explore

This handout has barely scratched the surface of exception handling , 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.
1. **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++.
2. **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.
3. **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`*.
4. **`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.