Chapter 12: Error Handling

> *Forty years ago, `goto`-laden code was considered perfectly good practice. Now we strive to write structured control flows. Twenty years ago, globally accessible data was considered perfectly good practice. Now we strive to encapsulate data. Ten years ago, writing functions without thinking about the impact of exceptions was considered good practice. Now we strive to write exception-safe code.*
>
> *Time goes on. We live. We learn.*
>
> – Scott Meyers, author of *Effective C++* and one of the leading experts on C++. [Mey05]

In an ideal world, network connections would never fail, files would always exist and be properly formatted, users would never type in malformed input, and computers would never run out of memory. Realistically, though, all of the above can and will occur and your programs will have to be able to respond to them gracefully. 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 *exception handling*, a C++ language feature that redirects program control in case of emergencies.

Exception handling is a complex topic and will have far-reaching effects on your C++ code. This chapter introduces the motivation underlying exception handling, basic exception-handling syntax, and some advanced techniques that can keep your code operating smoothly in an exception-filled environment.

A Simple Problem

Up to this point, all of the programs you've written have proceeded linearly – they begin inside a special function called `main`, then proceed through a chain of function calls and returns until (hopefully) hitting 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.

Suppose, for example, that we'd like to write our own version of the CS106B/X `StringToInteger` function, which converts a `string` representation of an number into an `int` equivalent. One possible (partial) implementation of `StringToInteger` might look like this:*

```
int StringToInteger(const string &input) {
    stringstream converter(input);
    int result; // Try reading an int, fail if we're unable to do so.

    converter >> result;
    if (converter.fail())
        // What should we do here?

    char leftover; // See if anything's left over.  If so, fail.
    converter >> leftover;
    if (!converter.fail())
       return result;
    else
       // What should we do here?
}
```

* This is based off of the `GetInteger` function we covered in the chapter on streams. Instead of looping and reprompting the user for input at each step, however, it simply reports errors on failure.

If the parameter `input` is a `string` with a valid integer representation, then this function simply needs to perform the conversion. But what should our function do if the parameter doesn't represent an integer? One possible option, and the one used by the CS106B/X implementation of `StringToInteger`, is to call a function like `Error` that prints an error and terminates the program. This response seems a bit drastic and is a decidedly suboptimal solution for several reasons. First, calling `Error` doesn't give the program a chance to recover from the problem. `StringToInteger` is a simple utility function, not a critical infrastructure component, and if it fails chances are that there's an elegant way to deal with the problem. For example, if we're using `StringToInteger` to convert user input in a text box into an integer for further processing, it makes far more sense to reprompt the user than to terminate the program. Second, in a very large or complicated software system, it seems silly to terminate the program over a simple string error. For example, if this `StringToInteger` function were used in an email client to convert a string representation of a time to an integer format (parsing the hours and minutes separately), it would be disastrous if the program crashed whenever receiving malformed emails. In essence, while using a function like `Error` will prevent the program from continuing with garbage values, it is simply too drastic a move to use in serious code.

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." In our case, we might want to have `StringToInteger` return -1 to indicate an error, for example. Compared with the "drop everything" approach of `Error` this may seem like a good option – it reports the error and gives the calling function a chance to respond. However, there are several major problems with this method. First, in many cases it is not possible to set aside a value to indicate failure. For example, suppose that we choose to reserve -1 as an error code for `StringToInteger`. In this case, we'd make all of our calls to `StringToInteger` as

```
if (StringToInteger(myParam) == -1) {
    /* ... handle error ... */
}
```

But what happens if the input to `StringToInteger` is the string `"-1"`? In this case, whether or not the `StringToInteger` function completes successfully, it will still return -1 and our code might confuse it with an error case.

Another serious problem with this approach is that if each function that might possibly return an error has to reserve sentinel values for errors, we might accidentally check the return value of one function against the error code of another function. Imagine if there were several constants floating around named `ERROR`, `STATUS_ERROR`, `INVALID_RESULT`, etc., and whenever you called a function you needed to check the return value against the correct one of these choices. If you chose incorrectly, even with the best of intentions your error-checking would be invalid.

Yet another shortcoming of this approach is that in some cases it will be impossible to reserve a value for use as a sentinel. For example, suppose that a function returns a `vector<double>`. What special `vector<double>` should we choose to use as a sentinel?

However, the most serious problem with the above approach is that you as a programmer can ignore the return value without encountering any warnings. Even if `StringToInteger` returns a sentinel value indicating an error, there are no compile-time or runtime warnings if you choose not to check for a return value. In the case of `StringToInteger` this may not be that much of a problem – after all, holding a sentinel value instead of a meaningful value will not immediately crash the program – but this can lead to problems down the line that can snowball into fully-fledged crashes. Worse, since the crash will probably be caused by errors from far earlier in the code, these sorts of problems can be nightmarish to debug.

Surprisingly, experience shows that many programmers – either out of negligence or laziness – forget to check return values for error codes and snowball effects are rather common.

We seem to have reached an unsolvable problem. We'd like an error-handling system that, like `Error`, prevents the program from continuing normally when an error occurs. At the same time, however, we'd like the elegance of sentinel values so that we can appropriately process an error. How can we combine the strengths of both of these approaches into a single system?

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 regions of code designated as areas 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 `StringToInteger` with an invalid 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 perfectly legal 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 {
       // Do something
    }
    catch(int myInt) {
       // If the code throws an int, execution continues here.
    }
    catch(const vector<int>& myVector) {
       // Otherwise, if the code throws a vector<int>, execution resumes here.
    }
    catch(const 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. The code inside the `try` block begins with the ball and continues talking as long as possible. If an error occurs, the `try` block throws the ball to the appropriate `catch` handler, which begins executing.

Let's return to our earlier example with `StringToInteger`. 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 `StringToInteger` as

```
int StringToInteger(const string& input) {
   stringstream converter(input);
   int result; // Try reading an int, fail if we're unable to do so.

   converter >> result;
   if (converter.fail())
      throw invalid_argument("Cannot parse " + input + " as an integer.");

   char leftover; // See if anything's left over.  If so, fail.
   converter >> leftover;
   if (!converter.fail())
      return result;
   else
      throw invalid_argument(string("Unexpected character: ") + leftover);
}
```

Notice that while the function itself does not contain a `try`/`catch` pair, 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 `StringToInteger` that looks like this:

* `what` is a poor choice of a name for a member function. Please make sure to use more descriptive names in your code!

```
try {
    int result = StringToInteger(myString);
    cout << "The result was: " << result;
}
catch(const invalid_argument& problem) {
    cout << problem.what() << endl; // Prints out the error message.
}
cout << "Yay!  We're done." << endl;
```

Here, if `StringToInteger` 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 `StringToInteger` 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 control 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 (`const invalid_argument&` instead of `invalid_argument`). As with parameter-passing, exception-catching can take values either by value or by reference, and by accepting the parameter by reference you can avoid making an unnecessary copy of the thrown object.

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. As its name implies, 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 recommended that you avoid doing so. Throwing an exception is *much* slower than returning a value because of the extra bookkeeping required, so be sure that you're only using exception handling 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 program from crashing 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 C++ code. Most notably, when using exception handling, unless you are absolutely certain that the classes and functions you use never throw exceptions, you must treat your code as though it might throw an exception

* If you use Microsoft's Visual Studio development environment, you might notice that various errors like null-pointer dereferences and stack overflows result in errors that mention "unhandled exception" in their description. This is a Microsoft-specific feature and is different from C++'s exception-handling system.

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() {
    int* myArray = new int[128];
    DoSomething(myArray);
    delete [] myArray;
}
```

Here, we allocate space for a raw array, 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 [] myArray` would never execute. As a result, our program will leak the array. If this program runs over a sufficiently long period of time, eventually we will run out of memory and our program will crash.

There are three main ways that we can avoid these problems. First, it's completely acceptable to just avoid exception-handling all together. This approach might seem like a cop-out, but it is a completely valid option that many C++ developers choose. Several major software projects written in C++ do not use exception handling (including the Mozilla Firefox web browser), partially because of the extra difficulties encountered when using exceptions. However, this approach results in code that runs into the same problems discussed earlier in this chapter with `StringToInteger` – functions can only communicate errors through return values and programmers must be extra vigilant to avoid ignoring return values.

The second approach to writing exception-safe code uses a technique called "catch-and-rethrow." Let's return to the above code example with a dynamically-allocated character buffer. We'd like to guarantee that the array we've allocated gets deallocated, but as our code is currently written, it's difficult to do so because the `DoSomething` function might throw an exception and interrupt our code flow. If there is an exception, what if we were able to somehow intercept that exception, clean up the buffer, and then propagate the exception outside of the `SimpleFunction` function? From an outside perspective, it would look as if the exception had come from inside the `DoSomething` function, but in reality it would have taken a quick stop inside `SimpleFunction` before proceeding outwards.

The reason this method works is that *it is legal to throw an exception from inside a `catch` block*. Although `catch` blocks are usually reserved for error handling, there is nothing preventing us from throwing the exception we catch. For example, this code is completely legal:

```
try{
    try {
        DoSomething();
    }
    catch(const invalid_argument& error) {
       cout << "Inner block: Error: " << error.what() << endl;
       throw error; // Propagate the error outward
    }
}
catch(const invalid_argument& error) {
    cout << "Outer block: Error:  " << error.what() << endl;
}
```

Here, if the `DoSomething` function throws an exception, it will first be caught by the innermost `try` block, which prints it to the screen. This `catch` handler then throws `error` again, and this time it is caught by the outermost `catch` block.

With this technique, we can almost rewrite our `SimpleFunction` function to look something like this:

```
void SimpleFunction() {
    int myArray = new int[128];

    /* Try to DoSomething.  If it fails, catch the exception and rethrow it. */
    try {
        DoSomething(myCString);
    }
    catch (/* What to catch? */) {
        delete [] myArray;
        throw /* What to throw? */;
    }

    /* Note that if there is no exception, we still need to clean things up. */
    delete [] myArray;
}
```

There's a bit of a problem here – what sort of exceptions should we catch? Suppose that we know every sort of exception `DoSomething` might throw. Would it be a good idea to write a `catch` block for each one of these types? At first this may seem like a good idea, but it can actually cause more problems than it solves. First, in each of the `catch` blocks, we'd need to write the same `delete []` statement. If we were to make changes to the `SimpleFunction` function that necessitated more cleanup code, we'd need to make progressively more changes to the `SimpleFunction catch` cascade, increasing the potential for errors. Also, if we forget to catch a specific type of error, or if `DoSomething` later changes to throw more types of errors, then we might miss an opportunity to catch the thrown exception and will leak resources. Plus, if we don't know what sorts of exceptions `DoSomething` might throw, this entire approach will not work.

The problem is that in this case, we want to tell C++ to catch *anything* that's thrown as an exception. We don't care about what the type of the exception is, and need to intercept the exception simply to ensure that our resource gets cleaned up. Fortunately, C++ provides a mechanism specifically for this purpose. To catch an exception of any type, you can use the special syntax `catch(...)`, which catches any exception. Thus we'll have the `catch` clause inside `DoSomething` be a `catch(...)` clause, so that we can catch any type of exception that `DoSomething` might throw. But this causes another problem: we'd like to rethrow the exception, but since we've used a `catch(...)` clause, we don't have a name for the specific exception that's been caught. Fortunately, C++ has a special use of the `throw` statement that lets you throw the current exception that's being processed. The syntax is

```
throw;
```

That is, a lone `throw` statement with no parameters. Be careful when using `throw;`, however, since if you're not inside of a `catch` block the program will crash!

The final version of `SimpleFunction` thus looks like this:

```
void SimpleFunction() {
        int myArray = new int[128];

        /* Try to DoSomething.  If it fails, catch the exception and rethrow it. */
        try {
            DoSomething(myCString);
        }
        catch (...) {
            delete [] myArray;
            throw;
        }

        /* Note that if there is no exception, we still need to clean things up. */
        delete [] myArray;
    }
```

As you can tell, the "catch-and-rethrow" approach to exception handling results in code that can be rather complicated. While in some circumstances catch-and-rethrow is the best option, in many cases there's a much better alternative that results in concise, readable, and thoroughly exception-safe code – object memory management.

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 memory 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 likelihood will have to use a special tool to check memory usage. You can also run into trouble where two objects each point to a shared object. If one of the objects 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 much 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

* 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`.

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>.  The 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 source `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 hold 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 ensures that there can be at most one `auto_ptr` 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.* Finally, because each `auto_ptr` can assume that it has sole ownership of the resource, `auto_ptr` can be implemented extremely efficiently and has almost zero overhead.

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:

---

* For those of you interested in programming language design, C++ uses what's known as *copy semantics* for most of its operations, where assigning objects to one another creates copies of the original objects. `auto_ptr` seems strange because it uses *move semantics*, where assigning `auto_ptr`s to one another transfers ownership of some resource. Move semantics are not easily expressed in C++ and the code to correctly implement `auto_ptr` is surprisingly complex and requires an intricate understanding of the C++ language. The next revision of C++, C++0x, will add several new features to the language to formalize and simply move semantics and will replace `auto_ptr` with `unique_ptr`, which formalizes the move semantics.

`explicit 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`.

`T* get() const`	`SomeFunction(myPtr.get()); // Retrieve stored resource` Returns the stored pointer. Useful for passing the managed resource to other functions.

Of course, dynamically-allocated memory isn't the only C++ resource that can benefit from object memory management. For example, when working with OS-specific libraries like Microsoft's Win32 library, you will commonly have to manually manage handles to system resources. In spots like these, writing wrapper classes that act like `auto_ptr` but that do cleanup using methods other than a plain `delete` can be quite beneficial. 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 are almost as ubiquitous as 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 – we allocated a `nodeT` object, but we didn't clean 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`. Even if we prematurely exit the function from an exception in `SomeComplicatedFunction`, the `auto_ptr` destructor will ensure that our resources are cleaned up. 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.

Documenting Invariants with `assert`

The exception-handling techniques we've covered so far are excellent ways of handling and recovering from errors that can only be detected at compile-time. If a network connection fails to open, or your graphics card fails to initialize correctly, you can use exceptions to report the error so that your program can detect and recover from the problem.

However, there is an entirely different class of problems that your programs might encounter at runtime – logic errors. As much as we'd all like to think that we can write perfect software on the first try, we all make mistakes when designing programs. We pass `NULL` pointers into functions that expect them to be non-`NULL`. We make accidental changes to linked lists while iterating over them. We pass in values by reference that we meant to pass in by value. These are normal errors in the programming process, and while time and experience can reduce their frequency, they can never entirely be eliminated. The question then arises – given that you are going to make mistakes during development, how can you design your software to make it easier to detect and correct these errors?

When designing software, at various points in the program you will expect certain conditions to hold true. You might expect that a certain integer is even, or that a pointer is non-`NULL`, etc. If these conditions don't hold, it's often a sign that your program contains a bug.

One trick you can use to make it easier to detect and diagnose bugs is to have the program check that these invariants hold at runtime. If they do, then everything is going according to plan, but if for some reason the invariants do not hold it could signal the presence of a bug. If the program can then report that an invariant failed to hold, it will make it significantly easier to debug. For this purpose, C++ provides the `assert` macro. `assert`, exported by the header `<cassert>`, checks to see that some condition holds true. If so, the macro has no effect. Otherwise, it prints out the statement that did not evaluate to true, along with the file and line number in which it was written, then terminates the program. For example, consider the following code:

```
void MyFunction(int *myPtr) {
    assert(myPtr != NULL);
    *myPtr = 137;
}
```

If a caller passes a null pointer into `MyFunction`, the assert statement will halt the program and print out a message that might look something like this:

```
Assertion Failed: 'myPtr != NULL': File: main.cpp, Line: 42
```

Because assert abruptly terminates the program without giving the rest of the application a chance to respond, you should not use `assert` as a general-purpose error-handling routine. In practical software development, `assert` is usually used to express programmer assumptions about the state of execution that can only be broken if the software is written incorrectly. If an `assert` fails, it means that the programmer made a mistake, not that something unusual occurred at runtime. For errors that might arise during normal execution, such as missing files or malformed user input, user exception handling. For errors that represent a bug in the original code, `assert` is a much better choice.

Let's consider a concrete example. Assume we have some enumerated type `Color`, which might look like this:

```
enum Color {Red, Green, Blue, Magenta, Cyan, Yellow, Black, White};
```

Now, suppose that we want to write a function called `IsPrimaryColor` that takes in a `Color` and reports whether that color is a primary color (red, green, or blue). Here's one implementation:

```
bool IsPrimaryColor(Color c) {
    switch(c) {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Here, if the color is `Red`, `Green`, or `Blue`, we return that the color is indeed a primary color. Otherwise, we return that it is not a primary color. However, what happens if the parameter is not a valid `Color`, perhaps if the call is `IsPrimaryColor(Color(-1))`? In this function, since we assume that that the parameter is indeed a color, we might want to indicate that to the program by explicitly putting in an `assert` test. Here's a modified version of the function, using assert and assuming the existence of a function `IsColor`:

```
bool IsPrimaryColor(Color c) {
    assert(IsColor(c)); // We assume that this is really a color.
    switch (c) {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Now, if the caller passes in an invalid `Color`, the program will halt with an assertion error pointing us to the line that caused the problem. If we have a good debugger, we should be able to figure out which caller erroneously passed in an invalid Color and can better remedy the problem. Were we to ignore this case entirely, we might have considerably more trouble debugging the error, since we would have no indication of where the problem originated.

While assert can be used to catch a good number of programmer errors during development, it has the unfortunate side-effect of slowing a program down at runtime because of the overhead of the extra checking involved. Consequently, most major compilers disable the `assert` macro in release or optimized builds. This may seem dangerous, since it eliminates checks for inconsistent state, but is actually not a problem because, in theory, you shouldn't be compiling a release build of your program if assert statements fail during execution.* Because `assert` is entirely disabled in optimized builds, you should use `assert` only to check that specific relations hold true, never to check the return value of a function. If an `assert` contains a call to a function, when assert is disabled in release builds, the function won't be called,

* In practice, this isn't always the case. But it's still a nice theory!

leading to different behavior in debug and release builds. This is a persistent source of debugging headaches.

More to Explore

Exception-handling and RAII are complex topics that have impressive ramifications for the way that your write C++ code. However, we simply don't have time to cover every facet of exception handling. In case you're interested in exploring more advanced topics in exception handling and RAII, consider looking into the following:

1. **The Standard Exception Classes**: In this chapter we discussed `invalid_argument`, one of the many exception classes available in the C++ standard library. However, there are several more exception classes that form an elaborate hierarchy. Consider reading into some of the other classes – some of them even show up in the STL!

2. **Exception Specifications**. Because functions can throw exceptions at any time, it can be difficult to determine which pieces of code can and cannot throw exceptions. Fortunately, C++ has a feature called an *exception specification* which indicates what sorts of exceptions a function is allowed to throw. When an exception leaves a function with an exception specification, the program will abort unless the type of the exception is one of the types mentioned in the specification.

3. **Function `try` Blocks**. There is a variant of a regular try block that lets you put the entire contents of a function into a try/catch handler pair. However, it is a relatively new feature in C++ and is not supported by several popular compilers. Check a reference for more information.

4. **`new` and Exceptions**. If your program runs out of available memory, the `new` operator will indicate a failure by throwing an exception of type `bad_alloc`. When designing custom container classes, it might be worth checking against this case and acting accordingly.

5. **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.

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. Additionally, there is a most excellent reference on `auto_ptr` available at http://www.gotw.ca/publications/using_auto_ptr_effectively.htm that is a great resource on the subject.

Practice Problems

1. Explain why the `auto_ptr` constructor is marked `explicit`. *(Hint: Give an example of an error you can make if the constructor is not marked `explicit`).*

2. The `SimpleFunction` function from earlier in this chapter ran into difficulty with exception-safety because it relied on a manually-managed C string. Explain why this would not be a problem if it instead used a C++ `string`.

3. Consider the following C++ function:

```
void ManipulateStack(stack<string>& myStack) {
    if (myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    /* This might throw an exception! */
    DoSomething(myStack);

    myStack.push(topElem);
}
```

This function accepts as input a C++ `stack<string>`, pops off the top element, calls the `DoSomething` function, then pushes the element back on top. Provided that the `DoSomething` function doesn't throw an exception, this code will guarantee that the top element of the `stack` does not change before and after the function executes. Suppose, however, that we wanted to absolutely guarantee that the top element of the stack never changes, even if the function throws an exception. Using the catch-and-rethrow strategy, explain how to make this the case.

5. Write a class called `AutomaticStackManager` whose constructor accepts a `stack<string>` and pops off the top element (if one exists) and whose destructor pushes the element back onto the `stack`. Using this class, rewrite the code in Problem 4 so that it's exception safe. How does this version of the code compare to the approach using catch-and-rethrow?