informit.com

Programming with Exceptions

Date: Apr 6, 2001 By Bjarne Stroustrup. Article is provided courtesy of Addison Wesley.

This article presents two series of examples of motivating the Standard C++ notion of a basic guarantee of exception safety, and shows how the techniques required to provide that basic guarantee actually lead to simpler programs.

Introduction

One of the nice things about Standard C++ is that you can use exceptions for systematic error handling. However, when you take that approach, you have to take care that when an exception is thrown, it doesn't cause more problems than it solves. That is, you have to think about exception safety. Interestingly, thoughts about exception safety often lead to simpler and more manageable code.

In this article, I first present concepts and techniques for managing resources and for designing classes in a program relying on exceptions. For this presentation, I use the simplest examples that I can think of. Finally, I explain how these ideas are directly reflected in the C++ standard library so that you can immediately benefit from them.

Resources and Resource Leaks

Consider a traditional piece of code:

```
void use_file(const char* fn)
{
        FILE* f = fopen(fn,"r");
        // use f
        fclose(f);
}
```

This code looks plausible. However, if something goes wrong after the call of `fopen()` and before the call of `fclose()`, it's possible to exit `use_file()` without calling `fclose()`. In particular, an exception might be thrown in the `use f` code, or in a function called from there. Even an ordinary `return` could bypass `fclose(f)`, but that's more likely to be noticed by a programmer or by testing.

A typical first attempt to make `use_file()` fault-tolerant looks like this:

```
void use_file(const char* fn)
{
        FILE* f = fopen(fn,"r");
        try {
                // use f
        }
        catch (...) {
                fclose(f);
                throw;
        }
        fclose(f);
}
```

The code using the file is enclosed in a `try` block that catches every exception, closes the file, and re-throws the exception.

The problem with this solution is that it's ad hoc, verbose, tedious, and potentially expensive. Another problem is that the programmer has to remember to apply this solution everywhere a file is opened, and must get it right every time. Such ad hoc solutions are inherently error-prone. Fortunately, there is a more elegant solution.

It's a fundamental rule that when a variable goes out of scope its destructor is called. This is true even if the scope is exited by an exception. Therefore, if we can get a destructor for a local variable to close the file, we have a solution. For example, we can define a class `File_ptr` that acts like a `FILE*`:

```
class File_ptr {
        FILE* p;
```

```
public:
        File_ptr(const char* n, const char* a) { p = fopen(n,a); }
        // suitable copy operations
        ~File_ptr() { if (p) fclose(p); }

        operator FILE*() { return p; }   // extract pointer for use
};
```

Given that, our function shrinks to this minimum:

```
void use_file(const char* fn)
{
        File_ptr f(fn,"r");
        // use f
}
```

The destructor will be called independently of whether the function is exited normally or exited because an exception is thrown. That is, the exception-handling mechanism enables us to remove the error-handling code from the main algorithm. The resulting code is simpler and less error-prone than its traditional counterpart.

The file example is a fairly ordinary resource leak problem. A resource is anything that our code acquires from somewhere and needs to give back. A resource that is not properly "given back" (released) is said to be *leaked*. Other examples of common resources are memory, sockets, and thread handles. Resource management is the heart of many programs. Typically, we want to make sure than every resource is properly released, whether we use exceptions or not.

You could say that I have merely shifted the complexity away from the `use_file()` function into the `File_ptr` class. That's true, but I need only write the `File_ptr` once for a program, and I often open files more frequently than that. In general, to use this technique we need one small "resource handle class" for each kind of resource in a system. Some libraries provide such classes for the resources they offer, so the application programmer is saved that task.

The C++ standard library provides `auto_ptr` for holding individual objects. It also provides containers, notably `vector` and `string`, for managing sequences of objects.

The technique of having a constructor acquire a resource and a destructor release it is usually called *resource acquisition is initialization*.

Class Invariants

Consider a simple vector class:

```
class Vector {
        // v points to an array of sz ints
        int sz;
        int* v;
public:
        explicit Vector(int n);          // create vector of n ints
        Vector(const Vector&);
        ~Vector();                       // destroy vector
        Vector& operator=(const Vector&); // assignment
        int size() const;
        void resize(int n);              // change the size to n
        int& operator[](int);            // subscripting
        const int& operator[](int) const; // subscripting
};
```

A class invariant is a simple rule, devised by the designer of the class, that must hold whenever a member function is called. This `Vector` class has the simple invariant `v points to an array of sz ints`. All functions are written with the assumption that this is true. That is, they can assume that this invariant holds when they're called. In return, they must make sure that the invariant holds when they return. For example:

```
int Vector::size() const { return sz; }
```

This implementation of `size()` looks clean enough, and it is. The invariant guarantees that `sz` really does hold the number of elements, and since `size()` doesn't change anything, the invariant is maintained.

The subscript operation is slightly more involved:

```
struct Bad_range { };

int& Vector::operator[](int i)
{
        if (0<=i && i<sz) return v[i];

        trow Bad_range();
}
```

That is, if the index is in range, return a reference to the right element; otherwise, throw an exception of type `Bad_range`.

These functions are simple because they rely on the invariant `v points to an array of sz ints`. Had they not been able to do that, the code could have become quite messy. But how can they rely on the invariant? Because constructors establish it. For example:

```
Vector::Vector(int i) :sz(i), v(new int[i]) { }
```

In particular, note that if `new` throws an exception, no object will be created. It's therefore impossible to create a `Vector` that doesn't hold the requested elements.

The key idea of the preceding section was that we should avoid resource leaks. So, clearly, `Vector` needs a destructor that frees the memory acquired by a `Vector`:

```
Vector::~Vector() { delete[] v; }
```

Again, the reason that this destructor can be so simple is that we can rely on `v` pointing to allocated memory.

Now consider a naive implementation of assignment:

```
Vector& Vector::operator=(const Vector& a)
{
        sz = a.sz;              // get new size
        delete[] v;             // free old memory
        v = new int[n];         // get new memory
        copy(a.v,a.v+a.sz,v);   // copy to new memory
}
```

People who have experience with exceptions will look at this assignment with suspicion. Can an exception be thrown? If so, is the invariant maintained?

Actually, this assignment is a disaster waiting to happen:

```
int main()
try
{
        Vector vec(10);
        cout << vec.size() << '\n';   // so far, so good
        Vector v2(40*1000000);         // ask for 160 megabytes
        vec = v2;                      // use another 160 megabytes
}
catch(Range_error) {
        cerr << "Oops: Range error!\n";
}
catch(bad_alloc) {
        cerr << "Oops: memory exhausted!\n";
}
```

If you hope for a nice error message `Oops: memory exhausted!` because you don't have 320MB to spare, you might be disappointed. If you don't have (about) 160MB free, the construction of `v2` will fail in a controlled manner, producing that expected error message. However, if you have 160MB, but not 320MB (as I do on my laptop), that's not going to happen. When the assignment tries to allocate memory for the copy of the elements, a `bad_alloc` exception is thrown. The exception handling then tries to exit the block in which `vec` is defined. In doing so, the destructor is called for `vec`, and the destructor tries to deallocate `vec.v`. However, `operator=()` has already deallocated that array. Some memory managers take a dim view of such (illegal) attempts to deallocate the same memory twice. One system went into an infinite loop when someone deleted the same memory twice.

What really went wrong here? The implementation of `operator=()` failed to maintain the class

invariant `v points to an array of sz ints`. That done, it was just a matter of time before some disaster happened. Once we phrase the problem that way, fixing it is easy: Make sure that the invariant holds before throwing an exception. Or, even simpler: Don't throw a good representation away before you have an alternative:

```
Vector& Vector::operator=(const Vector& a)
{
        int* p = new int[n];    // get new memory
        copy(a.v,a.v+a.sz,p);   // copy to new memory
        sz = a.sz;              // get new size
        delete[] v;             // free old memory
        v = p;
}
```

Now, if `new` fails to find memory and throws an exception, the vector being assigned will simply remain unchanged. In particular, our example above will exit with the correct error message: `Oops: memory exhausted!`.

Please note that `Vector` is an example of a resource handle; it manages its resource (the element array) simply and safely through the *resource acquisition is initialization* technique described earlier.

Exception Safety

The notions of resource management and invariants allow us to formulate the basic exception safety guarantee of the C++ standard library. Simply put, we can't consider any class exception safe unless it has an invariant and maintains it even when exceptions occur. Furthermore, we can't consider any piece of code to be exception-safe unless it properly releases all resources it acquired.

Thus, the standard library provides this guarantee:

- *Basic guarantee* for all operations: The basic invariants of the standard library are maintained, and no resources, such as memory, are leaked.

The standard library further defines these guarantees:

- *Strong guarantee* for key operations: In addition to providing the basic guarantee, either the operation succeeds, or has no effects. This guarantee is provided for key library operations, such as `push_back()`, and single-element `insert()` on a `list`.

- *Nothrow guarantee* for some operations: In addition to providing the basic guarantee, some operations are guaranteed not to throw an exception. This guarantee is provided for a few simple operations, such as `swap()` and freeing memory.

These concepts are invaluable when thinking about exception safety. Trying to add enough `try-blocks` to a program to deal with every problem is simply too messy, too complicated, and can easily lead to inefficient code. Structuring code as described earlier, with the aim of providing the strong guarantee where possible and the basic guarantee always, is easier and leads to more maintainable code. Note that the `Vector::operator=()` actually provides the strong guarantee. Often the strong guarantee comes naturally when you try not to delete an old representation before you've constructed a new one. The basic guarantee is used more when you're optimizing code to avoid having to duplicate information.

More Information

You can find a much more exhaustive discussion of exception safety and techniques for writing exception-safe code in Appendix E, "Standard-Library Exception Safety," in *The C++ Programming Language, Special Edition* (Addison-Wesley, 2000, ISBN 0-201-70073-5), here abbreviated *TC++PL* for simplicity. If you have a version of *TC++PL* without that appendix, you can download a copy of the appendix from my home pages at http://www.research.att.com/~bs.

If you're not acquainted with exceptions in C++, I strongly recommend that you learn about them and their proper use. Used well, exceptions can significantly simplify code. Naturally, I recommend *TC++PL*, but any modern C++ book—:meaning one that's written to take advantage of the ISO C++ standard and its standard library—:should have an explanation.

If you're not yet comfortable with standard library facilities such as `string` and `vector`, I strongly encourage you to try them. Code that directly messes around with memory management and elements in arrays is among the most prone to resource leaks and nasty exception-safety problems. Such code is rarely systematic and the data structures involved rarely have simple and

useful invariants. A very brief introduction to basic standard library facilities can be found in Chapter 3 of *TC++PL*, "A Tour of the Standard Library." That, too, can be downloaded from my home pages.