C++ Exceptions

Tim Bailey

Version 1.3
March 30, 2006

1 Introduction

When code in a reusable library encounters an error, it should never unilaterally terminate the program. Nor should it print a diagnostic message to the end-user. These actions, while perhaps suitable in a stand-alone application, are not appropriate for a reusable module whose end-use is not known in advance. However, typically the module will not have sufficient information to properly respond to the error. Rather it must somehow transmit notification of the error back up the call-stack until it reaches a function with sufficient context to specify an appropriate action.

For example, the error might be divide-by-zero, out-of-memory, or an inability to open a file. None of these is necessarily fatal, but the library module does not know how to deal with them. Possible actions taken by a calling function might be to, respectively, specify the result as *not-a-number* (NaN), request a smaller block of memory, or attempt to create a new file. Thus, the library can detect the error but cannot determine a response; the calling function can respond but cannot detect the error and needs to be informed by the library module.

A common approach in the C language is to have functions return a *status* code. The calling function must then check the return value and respond accordingly. This strategy puts the onus on the programmer to perform all error checking, and problems may arise if it is not done systematically. Furthermore, error handling code is tightly coupled with "ordinary" code, increasing clutter and reducing readability.

C++ exceptions provide a mechanism to decouple error handling code from ordinary code and ensures that unhandled errors do not go undetected but make themselves known by terminating the program. (It is far easier to debug a program that crashes immediately when an error occurs than one that fails at some arbitrary later point.) A detailed rationale for the C++ exception handling mechanism is given in [Str00, Section 14.1].

When a module encounters an error to which it cannot respond, it throws an exception. The exception travels up the call-stack in a process known as "stack unwinding" until it reaches a calling function that requests to *catch* that particular exception type. Exceptions are thrown using the keyword `throw` and are caught using `try` and `catch` blocks. However, there is a lot more to exception handling than just writing appropriate try-catch blocks. It turns out that designing reliable software in the presence of exceptions is not as simple as it first appears. This paper explains the basics of the exception handling mechanism, and presents coding techniques developed to counter its pitfalls. In particular, the "resource acquisition is initialisation" idiom is emphasised as an elegant and, indeed, essential technique for resource management.

2 In the Event of an Exception

When an exception is thrown, the following operations transpire.

- The function call-stack "unwinds" as the exception is passed up through the stack in search of a handler.

- The destructors of all complete (i.e., fully constructed[1]) objects are called as they are popped off the stack during unwinding.
- If the exception reaches a try-catch block, and one of the `catch` blocks has a matching exception type, the exception is passed to the `catch` handler. The handler may then deal with the problem, and the program resumes normal operation at the next statement below the try-catch block, or the handler may rethrow the exception, whereupon stack unwinding continues.
- If no matching handler is found, or they all rethrow, the program will terminate once the stack is fully unwound.
- Finally, just prior to program termination, the destructors of any constructed *static-extent* objects are called. (Objects have static extent if they are declared globally or with the keyword `static`.) Dynamically allocated objects are not destructed.

For example, suppose we have a function `func1()`, which creates some objects and performs a series of operations. At some point, it evaluates a conditional expression and, if the expression is `true`, throws an exception.

```
void func1(MyClass1 &a)
{
    MyClass2 b;
    MyClass3 c;
    ...
    if (condition)
        throw MyException();
    ...
}
```

In the process of stack-unwinding, local object `c` is destructed followed by local object `b`. Suppose that `func1()` was called by `func2()`, some of whose operations are enclosed in a try-catch block.

```
void func2()
{
    MyClass1 x;
    try {
        MyClass4 y;
        func1(x);
    }
    catch (MyException& e) {
        /* do something for MyException type error */
        throw; // rethrow exception
    }
    catch (...) {
        /* do something for any other type of exception */
        throw; // rethrow exception
    }
}
```

The stack continues to unwind until it reaches the head of the `try` block, destructing local object `y` along the way. The exception is then passed to the first matching `catch` handler, which performs some error-handling operations and rethrows the exception. Stack unwinding proceeds, destructing `x`, and finally terminating the program.

[1]If an exception is thrown inside a constructor, the affected object is incomplete and its destructor is not called. However, any complete sub-object within this object *will* be properly destructed [Sut00].

There a few further points to note from this example. First, the *rethrow* operation in a `catch` handler is optional, and should be performed only if the handler is unable to completely resolve the problem. Second, `catch` parameters should always be references or pointers, as this allows exceptions to be caught according to their base-classes without slicing. Third, the `catch` handlers are considered in top-down order, and the *first* matching handler is chosen [Str00, Section 14.3.2.1]. So more derived exceptions should appear above base-class exceptions. And finally, the ellipsis handler `(...)` defines a "catch-all" criteria; it catches all exception types not handled by catch-blocks above it, and any catch-blocks below it will never be considered.

3 The C Standard Library and Exceptions

The C standard library provides several functions for error handling: `abort()`, `exit()` and `assert()`.[2] These functions do not integrate well with C++ exception handling and should be used with caution or avoided altogether. The key problem is that they are ignorant of C++ destructors and do not call local object destructors before terminating the program [Str00, Section 9.4.1.1].

The `exit()` function signifies "normal" program termination, and performs a series of clean-up operations before termination.

- Calls any functions registered with `atexit()`.
- Flushes output streams and closes any open files.
- Calls destructors of any `static` objects.
- Returns a status code to the program's execution environment.

The `abort()` function, on the other hand, triggers "abnormal" program termination, and ends the program without any cleanup. In particular, the destructors of `static` objects are not called. It is possible to prevent "aborted" program termination by catching the `SIGABRT` signal with the standard C signal handling facility. This can be used to mitigate the worst of `abort()`'s behaviour.

The `assert()` macro is included in debug code only; the preprocessor translates it as nothing in release-version code. In debug code, it takes a conditional expression as an argument and, if it evaluates `false`, prints a diagnostic message and calls `abort()`.

Since a library should never presume to terminate a program, `abort()` and `exit()` should be avoided.[3] The `assert()` macro is an extremely useful tool for debugging and its liberal use to help expose erroneous code is well advised. However, as it calls `abort()`, it should never be applied to situations that might trigger in well-behaved code. A good rule-of-thumb is to use `assert()` only to catch bugs for which you, the writer of the particular module, are directly responsible. That is, use `assert()` as sanity checks for invariants in your library, but do not use it for anything that might be due to incorrect use of the library by client code, such as invalid arguments. These errors are best addressed by throwing an exception.

4 Defining Exception Types

Exceptions are thrown using the keyword `throw`. It is possible to throw objects of any type, and textbook examples throwing integers and string literals abound. For example,

```
if (fp == NULL)
    throw "Error: File could not be opened";
```

[2]The C standard library also provides `setjmp()`, and `longjmp()`, which perform stack-unwinding. However, these functions also do not invoke local-object destructors.

[3]While `abort()` and `exit()` are not appropriate in reusable code-modules, they often apply in specific applications. In general, `exit()` is the preferred termination function as it performs some cleanup and permits `static` object destructors to execute.

However, such practice should be avoided [Dew03, Item 64]. Exceptions are caught based on their type, and primitive types (i.e., ints, chars, etc) provide little information about the nature of the actual error. A far better approach is to define a set of specific exception classes to suit your library.

A good design technique is to develop an exception hierarchy, with a base-class to represent all errors that may propagate from your library, and derived classes to define specific error types. It is also a good idea to derive the base-class from std::exception so as to inherit its basic interface. Consider the following example. A set of exception classes is defined for a C++ implementation of the LAPACK linear algebra library. We first define a base-class LapackError, which is derived from std::exception so that it inherits the standard what() member function.

```
class LapackError : public std::exception
{
    int info_;
    const char *message_;
protected:
    LapackError(const char *message, int info) :
         info_(info), message_(message) {}
public:
    int get_info() const { return info_; }
    const char* what() const throw() { return message_; }
};
```

LAPACK routines typically return an info value for diagnostic purposes, and we include this in our exception class interface.

There are two main error types that our library may throw. These are numerical errors, such as divide-by-zero, and logical errors, such as passing a non-square matrix to a Cholesky decomposition routine. For these errors, we derive exception classes NumericalError and LogicalError, respectively.[4]

```
class NumericalError : public LapackError
{
public:
    NumericalError(const char *message, int info=0) :
        LapackError(message, info) {}
};

class LogicalError : public LapackError
{
public:
    LogicalError(const char *message, int info=0) :
        LapackError(message, info) {}
};
```

A LAPACK exception may then be thrown as in

```
if (rows != cols)
    throw LogicalError("Matrix is not square");
```

Suppose we have a series of functions that use LAPACK and possibly other libraries as well. Exceptions might be caught in the following manner.

[4]Notice that the LapackError constructor is protected, so that we cannot throw LapackError exceptions directly, but only exceptions derived from it. LapackError is used solely to provide a common interface for *catching* exceptions.

```
try {
    f();
    g();
    h();
}
catch(NumericalError &n) {
    // do specific handling for a numerical error
}
catch (LapackError &m) {
    // handle any other LapackError
}
catch (std::exception &e) {
    // do some cleanup or logging operations
    throw; // and rethrow
}
```

Exception handlers are considered in top-down order so more specific exception types should be placed higher up. Thus, `NumericalError` exceptions are given specific treatment, while any other exception derived from `LapackError` is dealt with by a common handler. In this way, base classes may be used to catch groups of exceptions. The final block catches any exception derived from `std::exception`. It is rare that a module would have sufficient context to handle so general an exception, and here the handler might simply query `e.what()` and record an error message in a log-file. The exception is then rethrown.

A cautionary note regarding `catch` handlers. If a function does not have enough context to handle a particular exception type, it should not catch it, but should allow it to travel further up the call-stack to a function that does. Thus, the `catch (std::exception &e)` in the above example is not good practice in general.[5] A function should catch and deal with any errors that it can resolve, and leave all the rest to higher levels. This is advantageous as it means not every function needs to check for errors and only select functions need act as firewalls, that either complete successfully or fail in a well-defined manner [Str00, Section 14.9].

A very brief note on exception specifications [Str00, Section 14.6]. Contemporary expert advice is: "don't use them" [Sut02]. They tend to cause more problems than they solve. Importantly, exception specifications should *never* be written for template functions as the exceptions thrown by the template type are not known in advance. If used at all, they should be restricted to major interfaces, such as the public interface of a library [Str00, Section 14.9]. In general, the only useful exception specification is `throw()`, which expressly states that a function will *not* throw an exception.

5 Writing Code that is Exception Safe

Exceptions were originally added to the C++ language in 1989 [Str00, Appendix B.3] and were widely presumed to provide a safe and elegant error-handling mechanism that would easily retrofit to existing software designs. This attitude was challenged in 1994 when Tom Cargill wrote an article [Car94] in the *C++ Report* rebutting a previous article [Ree93] by David Reed. The thrust of his argument was that exception handling is much more than just try-catch blocks. Rather it concerns the effects on surrounding code as an exception propagates up the call-stack. Writing robust code in the presence of exceptions proved much harder than originally anticipated, and required a fundamental change in thinking about code organisation and class design, particularly with regard to the acquisition and release of resources.

> *This "extraordinary care" demanded by exceptions originates in the subtle interactions among language features that can arise in exception handling. Counter-intuitively, the hard part of coding exceptions is not the explicit throws and catches.*

[5]However, it is good practice to catch and report all exceptions in `main()`.

The really hard part of using exceptions is to write all the intervening code in such a way that an arbitrary exception can propagate from its throw site to its handler, arriving safely and without damaging other parts of the program along the way. Tom Cargill [Car94].

Consider the following example, taken from [AM00], of a simple message-server program. The program has a class `User`, which models per-user information, and maintains both a local cache and a server-side database of a user's friends.

```
class User {
    ...
    std::string GetName();
    void AddFriend(User& newFriend);
private:
    typedef std::vector<User*> UserCont;
    UserCont friends_;
    UserDatabase* pDB_;
};
```

The member function `AddFriend()` adds a new friend to the user's local storage and the friend's name to the server database.

```
void User::AddFriend(User& newFriend)
{
    // Add the new friend to the database
    pDB_->AddFriend(GetName(), newFriend.GetName());
    // Add the new friend to local cache
    friends_.push_back(&newFriend);
}
```

The problem with this code is that, if `vector::push_back()` throws, the database will have a new item, but the local cache will not. The system becomes inconsistent. If the database function is known to not throw, the problem can be fixed by simply reordering the operations.

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    pDB_->AddFriend(GetName(), newFriend.GetName());
}
```

Now, if `vector::push_back()` throws, the system is left in the same state as it was before `User::AddFriend()` was called. Suppose, however, that `UserDatabase::AddFriend()` may also throw. The function can be made exception safe using a try-catch block.

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    try
    {
        pDB_->AddFriend(GetName(), newFriend.GetName());
    }
    catch (...)
    {
        friends_.pop_back();
        throw;
    }
}
```

This implementation is correct, but unwieldy and verbose. A much cleaner and more convenient solution is presented in the next section.

The C++ standard library defines a set of exception-safety guarantees, which describe the state of an object after an operation throws an exception [Str00, Appendix E.2].

- No guarantee. If an exception is throw, the state of the object is undefined, possibly corrupted.
- Basic guarantee. The object remains in a valid state and no resources (e.g., memory) are leaked.
- Strong guarantee. The operation either succeeds or has no effect, such that the object is left in the same state as before the operation.
- No-throw guarantee. The operation will not throw an exception.

An exception-safe library should aim to meet the basic guarantee as a minimum and the strong and no-throw guarantees where feasible. For example, consider the following `Stack` container class.

```
template <typename T>
class Stack
{
public:
    void push(const T &t)
    {
        data.push_back(t); // may throw
    }
    T pop()
    {
        T t = data.back(); // may throw
        data.pop_back();
        return t; // may throw
    }
    std::size_t size() const throw()
    {
        return data.size();
    }
private:
    std::vector<T> data;
};
```

In the `push()` operation, `vector::push_back()` or `T`'s copy constructor may throw. In either case, the state of the `Stack` object is unchanged, and so `Stack::push()` satisfies the strong guarantee. In the `pop()` operation, `T`'s copy constructor may throw in two places. If the second copy throws, the returned `T` object is lost, and so `Stack::pop()` cannot provide the strong guarantee. No resources are leaked, however, and the operation satisfies the basic guarantee. The `size()` operation is guaranteed to not throw and provides a no-throw exception specification. In general, a template container class should be "exception transparent" in that exceptions generated by contained objects propagate up the call-stack without harming the integrity of the container.

Three class member functions warrant special mention. Copy constructors and copy-assignment can be very difficult to make exception-safe. It is best, if possible, to design them to provide a no-throw guarantee [Str00, Section 14.4.6.1]. Otherwise, it is possible to attain the strong guarantee by defining a no-throw `swap()` operation and implementing copying in terms of `swap()` [Sut, Item 59], [Str00, Appendix E.3.3]. The third, and most important, class operation is the destructor. A destructor should never throw an exception under any circumstances [Sut97]. Doing so invalidates all exception-safety guarantees provided by the standard library.

There are a number of techniques for implementing exception-safe code [Str00, Appendix E.3 and E.6]. Two are shown in the database example above. That is, if a single operation only may throw, the simplest solution is to reorder the operations so that the one that throws is executed first. More generally, the set of operations can be enclosed in a try-catch block, where a catch-all handler resets the state. For most situations, however, a more elegant and efficient solution is possible, involving an idiom called "resource aquisition is initialisation".

6 Resource Aquisition is Initialisation

Resource aquisition is initialisation (RAII) [Str00, Section 14.4] is a fundamental design idiom for exception safe code. It is a technique whereby constructors and destructors are used to automate release of resources—a resource is acquired in the constructor and released in the destructor. C++ stack-unwinding semantics ensure that the resource is automatically released in the event of an exception. For example, given a class `Mutex`, used to lock shared objects in multi-threaded code, a class `ScopedLock` is devised to manage the locking and unlocking operations.

```
class ScopedLock {
public:
    explicit ScopedLock (Mutex& m) : mutex(m) { mutex.lock(); locked = true; }
    ~ScopedLock () { if (locked) mutex.unlock(); }
    void unlock() { locked = false; mutex.unlock(); }

private:
    ScopedLock (const ScopedLock&);
    ScopedLock& operator= (const ScopedLock&);
    Mutex& mutex;
    bool locked;
};
```

The mutex is locked when the `ScopedLock` object is created, and unlocked either by an explicit `unlock()` or implicitly when the object goes out-of-scope.

```
{
    ScopedLock locker(mtx);
    // mtx is locked

} // mtx automatically unlocked
```

RAII is not just for exception-safety. It also protects against programmer error by ensuring symmetry between acquire-release operations; the programmer no longer has to remember to release resources. Common resource operations are obtaining-releasing dynamic memory, locking and unlocking mutexes, opening-closing files, registering-deregistering with services, etc. Dynamic memory management is arguably the most prominent of these, and the standard C++ library provides the `auto_ptr` class as an RAII class for memory.

```
{
    std::auto_ptr<MyClass> m(new MyClass);
    m->do_something();
} // memory released
```

The Boost C++ library [BCL] provides a number of more advanced forms of memory handle classes, such as `smart_ptr`.

When implementing RAII classes, one must ensure that constructors perform their own resource cleanup given an exception—constructors should not leak resources. It is also important that destructors do not throw. Exceptions do not nest, and if a destuctor throws an exception during stack-unwinding from another exception, the program will abort.

Writing small handle classes for every type of resource can be tedious and can introduce new bugs of their own (e.g., in their constructor, destructor, copy-constructor, etc). A recent article [AM00] presents a generic class to overcome this problem and, particularly, to simplify writing exception-safe resource management. The class is called `ScopeGuard` and is basically a generic implementation of RAII. As such, it may be used to perform normal automatic release operations. For example,

```
{
    void *buffer = std::malloc(1024);
    ScopeGuard freeIt = MakeGuard(std::free, buffer);
    FILE *fp = std::fopen("afile.txt");
    ScopeGuard closeIt = MakeGuard(std::fclose, fp);
    // ...
} // file closed and memory freed
```

However, `ScopeGuard` also extends the RAII concept; it provides a `Dismiss()` operation. This is a significant innovation, greatly enhancing its value as an exception tool. The `Dismiss()` operation permits roll-back procedures to be invoked in the event of an exception, so as to meet exception-safe guarantees, and all without resorting to try-catch blocks. A roll-back function is registered with the `ScopeGuard` object before performing an unsafe operation. If the operation succeeds, the roll-back function is dismissed but, if an exception is thrown before reaching `Dismiss()`, roll-back is invoked.

Resuming the database example from the previous section, `ScopeGuard` ensures the strong guarantee as follows.

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    ScopeGuard guard = MakeObjGuard(friends_, &UserCont::pop_back);
    pDB_->AddFriend(GetName(), newFriend.GetName());
    guard.Dismiss();
}
```

If no exception is thrown, `Dismiss()` is called at the end of the block and `UserCont::pop_back()` is not invoked. However, if `UserDatabase::AddFriend()` throws, `Dismiss()` is never called, and `UserCont::pop_back()` is called from `guard`'s destructor.

7 Conclusion

Exceptions are intended for errors that cannot be resolved within their local context.[6] They inform another part of the program that an error has occurred and permit higher-up functions to respond to the problem. They facilitate separation of error-handling code from ordinary code, and are particularly suited to modules and libraries developed independently.

Designing software with exceptions must account for exception-based effects from the beginning. It is not possible to retrofit exception-safety onto existing exception-unaware code. A robust design should aim for the basic safety guarantee as a minimum and the strong guarantee where feasible. The RAII idiom is a crucial part of exception-safe design, and is useful for robust resource management in general. Use it early and use it often. Further comprehensive advice on designing with exceptions in given in [Str00, Section 14.11 and Appendix E.7].

8 Further Reading

Bjarne Stroustrup's text [Str00] is the definitive reference on the C++ language, and exception handling in particular. A large quantity of information on exception safety is freely available on

[6]It is important to avoid overusing exceptions. Don't throw an exception when the error can be dealt with by local control structures [Str00, Section 14.5].

the web, and the following articles are highly recommended. The argument against exceptions by Tom Cargill [Car94] can be found in the online demo of Scott Meyer's *Exceptional C++ CD* [Mey99]. So too can articles by Jack Reeves [Ree96] and Herb Sutter [Sut97] written in reply to Cargill's challenge. Sutter has written numerous other papers on exceptions, including exceptions in constructors [Sut00] and exception specifications [Sut02]. He also maintains a very useful website called "Guru of the Week" [Sut], which examines C++ problems in a question and answer format; many are directly concerned with exceptions. The C/C++ Users Journal has an online feature named the "Experts Forum", which contains many excellent articles, some of which feature exception handling, such as those by Kevlin Henney [Hen02] and Andrei Alexandrescu [AM00]. Two useful C++ FAQ sites that deal with exceptions, among other things, are Marshal Cline's [Cli] and Bjarne Stroustup's [Str].

References

[AM00] A. Alexandrescu and P. Marginean. Generic programming: Simplify your exception-safe code *aka* Change the way you write exception-safe code—forever. *C/C++ Users Journal*, December 2000. `http://www.cuj.com/experts/`.

[BCL] The Boost C++ Libraries. `http://www.boost.org`.

[Car94] T. Cargill. Exception handling: A false sense of security. *C++ Report*, Nov-Dec 1994. Also published in [Mey99].

[Cli] M. Cline. C++ FAQ lite. `http://www.parashift.com/c++-faq-lite/index.html`.

[Dew03] S.C. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley, 2003.

[Hen02] K. Henney. From mechanism to method: The safe stacking of cats. *C/C++ Users Journal*, February 2002. `http://www.cuj.com/experts/`.

[Mey99] S. Meyers. *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1999. A free demo version is available at `http://www.awprofessional.com/content/downloads/meyerscddemo/`.

[Ree93] D. Reed. Exceptions: Pragmatic issues with a new language feature. *C++ Report*, October 1993.

[Ree96] J.W. Reeves. Coping with exceptions. *C++ Report*, March 1996. Also published in [Mey99].

[Str] B. Stroustrup. C++ style and technique FAQ. `http://www.research.att.com/~bs/bs_faq2.html`.

[Str00] B. Stroustrup. *The C++ Programming language (Special Edition)*. Addison-Wesley, 2000. Appendices B (Compatibility), D (Locales) and E (Exception Safety) are available online at `http://www.research.att.com/~bs/3rd.html`.

[Sut] H. Sutter. Guru of the week. `http://www.gotw.ca/gotw/index.htm`.

[Sut97] H. Sutter. Exception-safe generic containers. *C++ Report*, September 1997. `http://www.gotw.ca/publications/index.htm`.

[Sut00] H. Sutter. Sutter's mill: Constructor failures. *C/C++ Users Journal*, November 2000. `http://www.gotw.ca/publications/index.htm`.

[Sut02] H. Sutter. Sutter's mill: A pragmatic look at exception specifications. *C/C++ Users Journal*, July 2002. `http://www.gotw.ca/publications/index.htm`.