Exception Safety Issues and Techniques

Exception handling is a fundamental error reporting mechanism in modern languages, including C++. In *Exceptional C++* [Sutter00] and *More Exceptional C++* [Sutter02] we considered in detail many issues related to defining what exception safety is, how to go about writing exception-safe code, and language issues and interactions to be aware of.

In this section, we continue to build on that material by turning our attention to some specific exception-related language features. We begin by answering some perennial questions: Is exception safety all about writing **Try** and **catch** in the right places? If not, then what? And what kinds of things should you consider when developing an exception safety policy for your software?

Delving beyond that, it's worth spending an entire Item to lay out reasons why writing exception-safe code is, well, just plain good for you, because doing that promotes programming styles that lead to more robust and more maintainable code in general, quite apart from their benefits in the presence of exceptions. But there is a limit to goodness and to "if some is good, then more is better" thinking, and that limit is hit well and hard when we get to exception specifications: Why are they in the language? Why are they well motivated in principle? And why, despite all that, should you stop using them in your programs?

This and more, as we dip our cups and drink again from the font of today's most current exceptional community wisdom.

---

Chapter 11. Try and Catch Me

*Difficulty: 3*

*Is exception safety all about writing **Try** and **catch** in the right places? If not, then what? And what kinds of things should you consider when developing an exception safety policy for your software?*

**JG Question**

1. What is a try-block?
Guru Question

2. "Writing exception-safe code is fundamentally about writing \texttt{try} and \texttt{catch} in the correct places." Discuss.

3. When should \texttt{try} and \texttt{catch} be used? When should they not be used? Express the answer as a good coding standard guideline.

Solution

Playing \texttt{catch}

1. \textit{What is a try-block?}

A try-block is a block of code (compound statement) whose execution will be attempted, followed by a series of one or more handler blocks that can be entered to catch an exception of the appropriate type if one is emitted from the attempted code. For example:

```cpp
// Example 11-1: A try-block example
//
try {
  if(some_condition)
    throw string("this is a string");
  else if(some_other_condition)
    throw 42;
} catch(const string&) { // do something if a string was thrown
} catch(...) { // do something if anything else was thrown
```

In Example 11-1, the attempted code might throw a \texttt{string}, an integer, or nothing at all.

There's More to Life Than Playing \texttt{catch}

2. "Writing exception-safe code is fundamentally about writing \texttt{Try} and \texttt{catch} in the correct places." Discuss.
Put bluntly, such a statement reflects a fundamental misunderstanding of exception safety. Exceptions are just another form of error reporting, and we certainly know that writing error-safe code is not just about where to check return codes and handle error conditions.

Actually, it turns out that exception safety is rarely about writing `try` and `catch` and the more rarely the better. Also, never forget that exception safety affects a piece of code's design; it is never just an afterthought that can be retrofitted with a few extra catch statements as if for seasoning.

There are three major considerations when writing exception-safe code:

1. *Where and when should I throw?* This consideration is about writing `throw` in the right places. In particular, we need to answer:
   - What code should *throw*? That is, what errors will we choose to report by throwing an exception instead of by returning a failure value or using some other method?
   - What code shouldn't *throw*? In particular, what code should provide the no-fail guarantee? (See Item 12 and [Sutter99].)

2. *Where and when should I handle an exception?* This is the only consideration that is in part about writing `try` and `catch` in the right places, and even this can be automated most of the time. First, consider the questions we need to answer:
   - What code could *catch*? That is, what code has enough context and knowledge to handle the error being reported by the exception (possibly by translating the exception into another form)? In particular, note that the catching code also needs to have enough knowledge to perform any necessary cleanup, such as of dynamic resources.
   - What code *should* *catch*? That is, of the code that could catch the exception, which is best suited to do so?

Once we've answered those questions, note that using the "resource acquisition is initialization" idiom can eliminate many try-blocks by automating the cleanup work. If you wrap dynamically allocated resources in owner-manager objects, typically the destructor can perform automatic cleanup at the right time without any `try` or `catch` at all. This is clearly desirable, not to mention that it's also usually easier to code now and to read later.

**Guideline**

*Prefer handling exception cleanup automatically by using de-structors instead of try/catch.*

3. *In all other places, is my code going to be safe if an exception comes roaring through out of any given function call?* This consideration is about using good resource management to avoid leaks, maintaining class and program invariants, and other kinds of program correctness. Put another way, it's about keeping the program from blowing up just because an exception happens to pass from its throw site through code that shouldn't have to particularly care about it before arriving at
an appropriate handler. For most programmers I've encountered, it turns out that this is typically by far the most time-consuming and difficult-to-learn aspect of exception safety.

Notice that only one of these three considerations has anything to do with writing `try` and `catch`. And even that one can often be avoided with the judicious use of destructors to automate cleanup.

4. **When should `try` and `catch` be used? When should they not be used? Express the answer as a good coding standard guideline.**

Here's one suggestion. In brief:

1. **Determine an overall error reporting and handling policy for your application or subsystem, and stick to it.** In particular, the policy should cover the following basic aspects (and generally includes much more):
   - *Error reporting.* Define what kinds of errors are to be reported and how; prefer using exceptions as opposed to other error reporting methods. Generally it's good to choose the most readable and maintainable method for each case by default; for example, exceptions are most useful for constructors and operators that cannot emit return values or where the throw site and the handler are widely separated.
   - *Error propagation.* Among other things, define the boundaries that exceptions shall not cross; typically these are module or API boundaries.
   - *Error handling.* Among other things, mandate that owning objects and destructors be used to manage cleanup instead of `try/catch`, wherever possible.

2. **Write `throw` in the places that detect an error and cannot deal with it themselves.** (Clearly, code that can resolve an error immediately doesn't need to report it!)

   For every operation, document what exceptions the operation might throw, and why, as part of the documentation for every function and module. You don't need to actually write an exception specification on each function (and you shouldn't; see Item 13), but you do need to document clearly and rigorously what the caller can expect, because error semantics are part of the function's or module's interface.

3. **Write `try` and `catch` in the places that have sufficient knowledge to handle the error, to translate it, or to enforce boundaries defined in the error policy.** In particular, I've found that there are three main reasons to write `try` and `catch`:
   - *To handle an error.* This is the simple case: An error happened, we know what to do about it, and we do it. Life goes on (sans the original exception, which has been safely put to rest). Again, do this in a destructor if possible; if not, go ahead and use `try/catch`.
   - *To translate an exception.* This means catching one exception that reports a lower-level problem and throwing another that is couched in the context of the translating code's own higher-level semantics. Alternatively, the original exception can be translated to another representation, such as an error code.

   For example, consider a communications session utility class that works across many host
types and transport protocols: An attempt to open a session to another host can fail for any number of low-level reasons that the session class can detect (for example, a failure to detect the network or authentication/permission rejection from the remote host). The Open function can handle these conditions itself, and there's no use reporting them to the caller, who after all has no idea what a Foo packet is or what to do if it Barifies; the session class handles its internal low-level errors directly, keeps itself in a consistent state, and reports its own higher-level error or exception to inform its caller that the session could not be opened.

void Session::Open(/*...*/) {
    try {
        // entire operation
    }
    catch(const ip_error& err) {
        // - do something about an IP error
        // - clean up
        throw Session::OpenFailed();
    }
    catch(const KerberosAuthentFail& err) {
        // - do something about an authentication error
        // - clean up
        throw Session::OpenFailed();
    }

    // ... etc. ...
}

- To catch(...) on subsystem boundaries or other run-time firewalls. This usually also involves translating the error, usually to an error code or other nonexceptional representation. For example, when your stack unwinds up to a C API, you have only two choices: Return an error code right away for the current API function, or set an error state that the caller can query later via a complementary GetLastError API function.

Guidelines

Determine an overall error reporting and handling policy for your application or subsystem, and stick to it. Include a policy for error reporting, error propagation, and error handling.

Write throw in the places that detect an error and cannot deal with it themselves.

Write try and catch in the places that have sufficient knowledge to handle the error, to translate it, or to enforce boundaries defined in the error policy (e.g., to catch(...) on subsystem boundaries or other run-time firewalls).

Summary

A wise man once said:

*Lead, follow, or get the blazes out of the way!*
In exception safety analysis, we might say instead:

\texttt{throw, catch, or get the blazes out of the way!}

In practice, the last get-out-of-the-way case accounts for the bulk of exception safety analysis and testing. That's the major reason why exception-safe coding is not fundamentally about writing \texttt{Try} and \texttt{catch} in the right places. Rather, it's fundamentally about getting out of the bullet's way in the right places.

\section*{Chapter 12. Exception Safety: Is It Worth It?}

\textbf{Difficulty: 7}

Is it worth the effort to write exception-safe code? This should no longer be a seriously disputed and debated point... but sometimes it still is.

\begin{center}
\begin{tabular}{|p{\textwidth}|}
\hline
\textbf{Guru Question} \\
\hline
1. Recap: Briefly define the Abrahams exception safety guarantees (basic, strong, and nofail).
2. When is it worth it to write code that meets:
   \begin{enumerate}
     \item the basic guarantee?
     \item the strong guarantee?
     \item the nofail guarantee?
   \end{enumerate}
\hline
\end{tabular}
\end{center}

\section*{Solution}

The Abrahams Guarantees
1. **Recap: Briefly define the Abrahams exception safety guarantees (basic, strong, and nofail).**

The *basic guarantee* says that failed operations might alter program state, but no leaks occur and affected objects/modules are still destructible and usable, in a consistent (but not necessarily predictable) state.

The *strong guarantee* involves transactional commit/rollback semantics: Failed operations guarantee that program state is unchanged with respect to the objects operated upon. This means no side effects that affect the objects, including the validity or contents of related helper objects such as iterators pointing into containers being manipulated.

Finally, the *nofail guarantee* says that failure simply will not be allowed to happen. In terms of exceptions, the operation will not throw an exception. (Abrahams and others, including the earlier Exceptional C++ books, originally called the nothrow guarantee. I have switched to calling it the nofail guarantee because these guarantees apply equally to all error handling, whether using exceptions or some other mechanism such as error codes.)

### When Are Stronger Guarantees Worthwhile?

2. **When is it worth it to write code that meets:**

   a. the basic guarantee?

   b. the strong guarantee?

   c. the nofail guarantee?

It is *always* worth it to write code that meets at least one of these guarantees. There are several good reasons:

1. *Exceptions happen.* (To paraphrase a popular saying.) They just do. The standard library emits them. The language emits them. We have to code for them. Fortunately, it's not that big a deal, because we now know how to do it. It does require adopting a few habits, however, and following them diligently but then so did learning to program with error codes.

The big thorny problem is, as it ever was, the general issue of error handling. The detail of how to report errors, using return codes or exceptions, is almost entirely a syntactic detail where the main differences are in the semantics of how the reporting is done, so each approach requires its own style.

2. *Writing exception-safe code is good for you.* Exception-safe code and good code go hand in hand. The same techniques that have been popularized to help us write exception-safe code are, pretty much without exception, things we usually ought to be doing anyway. That is, exception-safety techniques are good for your code in and of themselves, even if exception safety weren't a consideration.

To see this in action, consider the major techniques I and others have written about to make exception safety easier:

- Use "resource acquisition is initialization" (RAII) to manage resource ownership. Using resource-owning objects such as `Lock` classes and `shared_ptr` (see [Boost, Sutter02a]) is just a good idea...
in general. It should come as no surprise that among their many benefits we should also find exception safety. How many times have you seen a function (here we're talking about someone else's function, of course, not something you wrote) where one of the code branches that leads to an early return fails to do some cleanup because cleanup wasn't being managed automatically using RAI?

- Use "do all the work off to the side, then commit using nonthrowing operations only" to avoid changing internal state until you're sure the whole operation will succeed. Such transactional programming is clearer, cleaner, and safer even with error codes. How many times have you seen a function (and naturally here again we're talking about someone else's function, of course, not something you wrote) where one of the code branches that leads to an early return fails to preserve the object's state, because some fiddling with internal state had already happened before a later operation failed?

- Prefer "one class (or function), one responsibility." Functions that have multiple effects, such as the Stack::Pop and EvaluateSalaryAndReturnName functions described in Items 10 and 18 of Exceptional C++ [Sutter00], are difficult to make strongly exception-safe. Many exception safety problems can be made much simpler, or eliminated without conscious thought, simply by following the "one function, one responsibility" guideline. And that guideline long predates our knowledge that it happens to also apply to exception safety; it's just a good idea in and of itself.

Doing these things is just plain good for you.

Having said that, then, which guarantee should we use when? In brief, here's the guideline followed by the C++ standard library, and one that you can profitably apply to your own code:

**Guideline**

A function should always support the strictest guarantee that it can support without penalizing callers who don't need it.

So if your function can support the nofail guarantee without penalizing callers who don't need that guarantee, it should do so. Note also that a handful of key functions simply must be nofail operations:

**Guideline**

Never allow a destructor, deallocation, or swap function to emit an exception, because otherwise it's often impossible to reliably and safely perform cleanup.

Otherwise, if your function can support the strong guarantee without penalizing some users, it should do so. Note that vector::insert is an example of a function that does not support the strong guarantee in general because doing so would force us to make a full copy of the vector's contents every time we insert an element, and not all programs care so much about the strong guarantee that they're willing to incur that much overhead. (Those programs that do can wrap vector::insert with the strong guarantee themselves, trivially: Take a copy of the vector, perform the insert on the copy, and once it's
successful, perform a **swap** with the original **vector**, and you're done.)

Otherwise, your function should support the basic guarantee.

For more information about these concepts, such as what a nonthrowing **swap** is all about or why destructors should never emit exceptions, see also *Exceptional C++* [Sutter00] and *More Exceptional C++* [Sutter02].

---

**Chapter 13. A Pragmatic Look at Exception Specifications**

**Difficulty: 6**

*Now that the community has gained experience with exception specifications, it's time to reflect on when and how they should best be used. This Item considers the usefulness, or lack thereof, of exception specifications and how the answers can vary across real-world compilers.*

---

**JG Questions**

1. What happens when an exception specification is violated? Why? Discuss the basic rationale for this C++ feature.

2. For each of the following functions, describe what exceptions the function could throw.

   ```cpp
   int Func();
   int Gunc() throw();
   int Hunc() throw(A,B);
   ```

---

**Guru Question**

3. Is an exception specification part of the function's type? Explain.


5. When is it worth it to write an exception specification on a function? Why would you...
choose to write one, or why not?

Solution

As we consider work now underway on the new C++ standard, C++0x, it's a good time to take stock of what we're doing with, and have learned from, our experience with the current standard [C++03]. The vast majority of standard C++'s features are good, and they get the lion's share of the print because there's not much point harping on the weaker features. Rather, the weaker and less useful features more often just get ignored and atrophy from disuse until many people forget they're even there (not always a bad thing). That's why you've seen relatively few articles about obscure features such as valarray, bitset, locales, and the legal expression 5[a] (although a version of the last one does show up in another Item later in this book) and the same is true, we will find, for exception specifications.

Let's now take a closer look at the state of our experience with standard C++ exception specifications.

Moving Violations

1. What happens when an exception specification is violated? Why? Discuss the basic rationale for this C++ feature.

The idea of exception specifications is to do a run-time check that guarantees that only exceptions of certain types will be emitted from a function (or that none will be emitted at all). For example, the following function's exception specification guarantees that f will emit only exceptions of type A or B:

```cpp
int f() throw(A, B);
```

If an exception would be emitted that's not on the invited-guests list, the function unexpected will be called. For example:

```cpp
// Example 13-1
int f() throw(A, B) { // A and B are unrelated to C
    throw C(); // will call unexpected
}
```

You can register your own handler for the unexpected-exception case by using the standard set_unexpected function. Your replacement handler must take no parameters and it must have a void return type. For example:

```cpp
void MyUnexpectedHandler() { /*...*/ }
```
The remaining question is, what can your unexpected handler do? The one thing it can't do is return via a usual function return. There are two things it may do:

- It could decide to translate the exception into something that's allowed by that exception specification, by throwing its own exception that does satisfy the exception specification list that caused it to be called. Then stack unwinding would resume from where it had left off.

- It could call `terminate`, which ends the program. (The `terminate` function can itself be replaced, but any replacement must likewise also always end the program.)

**The Story So Far**

The idea behind exception specifications is easy to understand: In a C++ program, unless otherwise specified, any function might conceivably emit any type of exception. Consider a function named `Func` (because the name `f` is so dreadfully over-used):

2. For each of the following functions, describe what exceptions the function could throw.

```cpp
// Example 13-2(a)
//
int Func();  // can throw anything
```

By default, in C++, `Func` could indeed throw anything, just as the comment added hereto says. Now, often we know just what kinds of things a function might throw, and then it's certainly reasonable to want to supply the compiler and the human programmer with some information limiting what exceptions could come tearing out of a function. For example:

```cpp
// Example 13-2(b)
//
int Gunc() throw();  // will throw nothing
int Hunc() throw(A, B);  // can only throw A or B
```

In these cases, the function's exception specification exists to say something about what the functions `Gunc` and `Hunc` could emit. The comments document colloquially what the specifications say. We'll return to that "colloquially" part in a moment, because as it turns out, these two comments are deceptively close to being correct.

One might naturally think that making a statement about what the functions might throw would be a good thing, that more information is better. One would not necessarily be right, because the devil is in the details: Although the motivation is noble, the way exception specifications are, well, specified in C++ isn't always useful and can often be downright detrimental.

**Issue the First: A "Shadow Type System"**
3. Is an exception specification part of the function's type? Explain.

John Spicer, of Edison Design Group fame and an author of large swathes of the template chapter of the C++ standard, has been known to call C++'s exception specifications a "shadow type system." One of C++'s strongest features is its strong type system, and that's well and good. Why would we call exception specifications a shadow type system instead of just part of the type system?

The reason is simple, and twofold:

- Exception specifications don't participate in a function's type.

- Except when they do.

Consider first an example of when exception specifications don't participate in a function's type. Reflect on the following code:

```c++
// Example 13-3(a): You can't write an exception specification in a typedef.
//
void f() throw(A,B);

typedef void (*PF)() throw(A,B); // syntax error

PF pf = f; // can't get here because of the error
```

The exception specification on the `typedef` is illegal. C++ doesn't let you write that, so the exception specification is not allowed to participate in the type of a function… at least, not in the context of a `typedef`, it's not. But in other cases, exception specifications do indeed participate in the function's type, such as if you wrote the same function declaration without the `typedef`:

```c++
// Example 13-3(b): But you can if you omit the typedef!
//
void f() throw(A,B);

void (*pf)() throw(A,B); // ok
pf = f; // ok
```

Incidentally, you can do this kind of assignment of a pointer to a function as long as the target's exception specification is no more restrictive than the source's:

```c++
// Example 13-3(c): Also kosher, low-carb, and fat-free.
//
void f() throw(A,B);

void (*pf)() throw(A,B, C); // ok
pf = f; // ok, pf's type is less restrictive
```

Exception specifications also participate in a virtual function's type when you try to override it:
 ISSUE the Second: (Mis)understandings

The second issue has to do with knowing what you're getting. As many notable persons, including the authors of the Boost exception specification rationale [BoostES], have put it, programmers tend to use exception specifications as though they be-haved the way the programmer would like, instead of the way they actually do do-be-have.

Hence the question:


Here's what many people think exception specifications do:

- Guarantee that functions will throw only listed exceptions (possibly none).
- Enable compiler optimizations based on the knowledge that only listed exceptions (possibly none) will be thrown.

These expectations are, again, deceptively close to being correct. Consider again the code in Example 13-2(b):

// Example 13-2(b) reprise, and two potential white lies:
//
int Gunc() throw(); // will throw nothing ←?
int Hunc() throw(A,B); // can only throw A or B ←?

Are the comments correct? Not quite. Gunc might indeed throw something, and Hunc might well throw something other than A or B! The compiler just guarantees to beat them senseless if they do... oh, and to beat your program senseless too, most of the time.

Because Gunc or Hunc could indeed throw something they promised not to, not only can't the compiler assume it won't happen, but the compiler is also responsible for being the policeman with the billy club who checks to make sure such a bad thing doesn't happen undetected. If it does happen, then the compiler must invoke the unexpected function. Most of the time, that will terminate your program.

Why? Because there are only two ways out of unexpected, neither of which is a normal return. You can pick your poison:
• Throw instead an exception that the exception specification does allow. If so, the exception propagation continues as it would normally have. But remember that the unexpected handler is global - there is only one for the whole program. A global handler is highly unlikely to be smart enough to Do the Right Thing for any given particular case, and the result is to go to terminate, go directly to terminate, do not pass catch, do not collect $200.

• Throw instead (or rethrow) an exception that the exception specification (still) doesn't allow. If the original function allowed a bad_exception type in its exception specification, okay, then it's a bad_exception that will now get propagated. But if not, then go to terminate, go directly to terminate...

Because violated exception specifications end up terminating your program the vast majority of the time, I think it's legitimate to call that "beating your program senseless."

Earlier, we saw two bullets stating what many people think that exception specifications do. Here is an edited statement that more accurately portrays what they actually do do [sic]:[19]

[19] Yes, this is a sic joke.

• Guarantee **Enforce at run-time** that functions will throw only listed exceptions (possibly none).

• Enable or prevent compiler optimizations based on the knowledge that only listed exceptions (possibly none) will be thrown having to check whether listed exceptions are indeed being thrown.

To see what a compiler has to do, consider the following code, which provides a body for one of our sample functions, **Hunc**:

```cpp
// Example 13-4(a)
//
int Hunc() throw(A,B) {
    return Junc();
}
```

Functionally, the compiler must generate code like the following, and it's typically just as costly at runtime as if you'd hand-written it yourself (though less typing because the compiler generates it for you):

```cpp
// Example 13-4(b): A compiler's massaged version of Example 13-4(a)
//
int Hunc()
try {
    return Junc();
} catch (A) {
    throw;
} catch (B) {
    throw;
} catch (...) {
    std::unexpected(); // won't return! but might throw an A or a B if you're lucky
}
```
Here we can see more clearly why, rather than letting the compiler make optimizations by assuming only certain exceptions will be thrown, it's exactly the reverse: The compiler has to do *more work to enforce* at run-time that only those exceptions are indeed thrown.

### The Scoop on Exception Specifications

Most people are surprised to discover that exception specifications can cause performance penalties. One reason this is true has now been amply demonstrated: The exception specification incurs the overhead for the implicitly generated `try/catch` blocks, although this might be minor on efficient compilers.

There are at least two other ways that exception specifications can commonly cost you in run-time performance:

- Some compilers will automatically refuse to inline a function having an exception specification, just as they can apply other heuristics such as refusing to inline functions that have more than a certain number of nested statements or that contain any kind of loop construct.

- Some compilers don't optimize exception-related knowledge well at all and will add the compiler-generated `try/catch` blocks even when the function body provably can't throw. (I mean it; that's not a typo.)

Moving beyond run-time performance, exception specifications can cost you development time because they increase coupling. For example, removing a type from the base class virtual function's exception specification is a quick and easy way to break lots of derived classes in one swell foop (if you're looking for a way). Try it on a Friday afternoon check-in and start a pool to guess the number of angry emails that will be waiting for you in your inbox on Monday morning.

Hence our natural next question would be:

5. *When is it worth it to write an exception specification on a function? Why would you choose to write one, or why not?*

Here's what seems to be the best advice we as a community have learned as of this writing:

#### Guidelines

**Moral #1:** Never write an exception specification.

**Moral #2:** Except possibly an empty one, but if I were you, I'd avoid even that.

Boost's experience is that a throws-nothing specification on a non-inline function is the only place where an exception specification "may have some benefit with some compilers." That's a rather underwhelming statement in its own right but a useful consideration if you have to write portable code that will be used on more than one compiler platform.
It's actually even a bit worse than that in practice, because it turns out that popular implementations vary in how they actually handle exception specifications. At least one popular C++ compiler (Microsoft's, up to the current version as of this writing, 7.1 (2003)) parses exception specifications but does not actually enforce them, reducing the exception specifications to glorified comments. But wait, there's more: At the same time, there are legal optimizations a compiler can perform outside a function, and which the Microsoft 7.x compiler does perform, that rely on the exception specification enforcement's being done inside each function; the idea is that if the function did try to throw something it shouldn't, then the internal handler would stop the program and control would never return to the caller, so because control did return to the caller the code generated for the call site can assume nothing was thrown and do such things as eliminate external try/catch blocks.

On such a compiler that fails to enforce the exception specification but still relies on its being enforced, the meaning of throw() changes from the standard "check me on this, stop me if I inadvertently throw" to a "trust me on this, assume I'll never throw and optimize away." So beware: If you do choose to use even an empty exception specification, read your compiler's documentation and check to see what it will really do with it. You might just be surprised. Be aware, drive with care.

**Summary**

In brief, don't bother with exception specifications. Even experts don't bother.

Slightly less briefly, the major issues are:

- Exception specifications can cause surprising performance hits, for example if the compiler turns off inlining for functions with exception specifications.

- A run-time unexpected error is not always what you want to have happen for the kinds of mistakes that exception specifications are meant to catch.

- You generally can't write useful exception specifications for function templates anyway because you generally can't tell what the types they operate on might throw.

While presenting this material as part of a broader talk at a conference not long ago, I asked how many of the about 100 people in the room each time had used exception specifications. About half put up their hands. Then a wag at the back said (quite correctly) that I should also ask how many of those people later took the exception specifications back out again, so I asked; about the same number of hands went up. This is telling. The world-class library designers at Boost went through the same experience, and that's why their coding policy on writing exception specifications pretty much boils down to "don't do that" [BoostES].

True, many well-intentioned people wanted exception specifications in the language, and that's why we have them. This reminds me of a cute poem that I first encountered about 15 years ago as it circulated in midwinter holiday emails. Set to the cadence of "'Twas the Night Before Christmas," these days it's variously titled "'Twas the Night Before Implementation" or "'Twas the Night Before Crisis." It tells of a master programmer who slaves away late at night in the holiday season to meet user deadlines and performs multiple miracles to pull out a functioning system that perfectly implements the requirements… only to experience a final metaphorical kick in the teeth as the last four lines of the ditty report:

> The system was finished, the tests were concluded,
The users' last changes were even included.

And the users exclaimed, with a snarl and a taunt,

"It's just what we asked for, but not what we want!"

The thought resonates as we finish considering our current experience with exception specifications. The feature seemed like a good idea at the time… and it is just what some people had asked for.

Be careful what you wish for. You might get your wish.