Exception-Safety in Generic Components

Lessons Learned from Specifying Exception-Safety for the C++ Standard Library

David Abrahams

Dragon Systems
David_Abrahams@dragonsys.com

Abstract. This paper represents the knowledge accumulated in response to a real-world need: that the C++ Standard Template Library exhibit useful and well-defined interactions with exceptions, the error-handling mechanism built-in to the core C++ language. It explores the meaning of exception-safety, reveals surprising myths about exceptions and genericity, describes valuable tools for reasoning about program correctness, and outlines an automated testing procedure for verifying exception-safety.

Keywords: exception-safety, exceptions, STL, C++

1 What Is Exception-Safety?

Informally, exception-safety in a component means that it exhibits reasonable behavior when an exception is thrown during its execution. For most people, the term "reasonable" includes all the usual expectations for error-handling: that resources should not be leaked, and that the program should remain in a well-defined state so that execution can continue. For most components, it also includes the expectation that when an error is encountered, it is reported to the caller.

More formally, we can describe a component as minimally exception-safe if, when exceptions are thrown from within that component, its invariants are intact. Later on we'll see that at least three different levels of exception-safety can be usefully distinguished. These distinctions can help us to describe and reason about the behavior of large systems.

In a generic component, we usually have an additional expectation of *exception-neutrality*, which means that exceptions thrown by a component's type parameters should be propagated, unchanged, to the component's caller.

2 Myths and Superstitions

Exception-safety seems straightforward so far: it doesn't constitute anything more than we'd expect from code using more traditional error-handling techniques. It might be worthwhile, however, to examine the term from a psychological viewpoint. Nobody ever spoke of "error-safety" before C++ had exceptions.

It's almost as though exceptions are viewed as a *mysterious attack* on otherwise correct code, from which we must protect ourselves. Needless to say, this doesn't lead to a healthy relationship with error handling! During standardization, a democratic process which requires broad support for changes, I encountered many widely-held superstitions. In order to even begin the discussion of exception-safety in generic components, it may be worthwhile confronting a few of them.

"Interactions between templates and exceptions are not well-understood." This myth, often heard from those who consider these both *new* language features, is easily disposed of: there simply are no interactions. A template, once instantiated, works in all respects like an ordinary class or function. A simple way to reason about the behavior of a template with exceptions is to think of how a specific instantiation of that template works. Finally, the genericity of templates should not cause special concern. Although the component's client supplies part of the operation (which may, unless otherwise specified, throw arbitrary exceptions), the same is true of operations using familiar virtual functions or simple function pointers.

"It is well known to be impossible to write an exception-safe generic container." This claim is often heard with reference to an article by Tom Cargill [4] in which he explores the problem of exception-safety for a generic stack template. In his article, Cargill raises many useful questions, but unfortunately fails to present a solution to his problem.[1] He concludes by suggesting that a solution may not be possible. Unfortunately, his article was read by many as "proof" of that speculation. Since it was published there have been many examples of exception-safe generic components, among them the C++ standard library containers.

"Dealing with exceptions will slow code down, and templates are used specifically to get the best possible performance." A good implementation of C++ will not devote a single instruction cycle to dealing with exceptions until one is thrown, and then it can be handled at a speed comparable with that of calling a function [7]. That alone gives programs using exceptions performance equivalent to that of a program which ignores the possibility of errors. Using exceptions can actually result in faster programs than "traditional" error handling methods for other reasons. First, a catch block clearly indicates to the compiler which code is devoted to error-handling; it can then be separated from the usual execution-path, improving locality of reference. Second, code using "traditional" error handling must typically test a return value for errors after every single function call; using exceptions completely eliminates that overhead.

"Exceptions make it more difficult to reason about a program's behavior." Usually cited in support of this myth is the way "hidden" execution paths are followed during stack-unwinding. Hidden execution paths are nothing new to

[1] Probably the greatest impediment to a solution in Cargill's case was an unfortunate combination of choices on his part: the interface he chose for his container was incompatible with his particular demands for safety. By changing either one he might have solved the problem.