A preprint of 978-1-4673-1766-5/12 © 2012 IEEE
WEH 2012, Zurich, Switzerland
The final publication is available at IEEE Xplore
http://ieeexplore.ieee.org/document/6226598/

Concrete Error Handling Mechanisms Should Be Configurable

Tero Hasu
Bergen Language Design Laboratory
Department of Informatics, University of Bergen
Bergen, Norway
tero.hasu@ii.uib.no

***Abstract*—We argue that programmers should not need to decide on a specific error handling mechanism when implementing a C or C++ library. Rather, it should be possible to make that decision at configuration time in order to achieve better portability and more convenient use of libraries.**

***Keywords*-C, C++, error handling, exceptions, portability, programming languages, software engineering, source-to-source translation**

I. Motivation and Position

When creating a software library one would ideally like to make it generally usable in any application that requires some of the provided functionality. Given an application and a library, one has to ask: (1) can the library be built for the platform on which the application runs; and (2) is the library convenient to use given the idioms and conventions used in the application code?

By implementing in C or C++ one automatically achieves a degree of portability to a multitude of platforms since C and C++ compilers are so ubiquitous, but the programming language is only a part of the picture.

It is true that there typically are vendor-supported C and C++ compilers even for niche platforms (e.g., Bada, Symbian, z/OS). However, these platforms also come with their own custom libraries and idioms used in native applications, and the small size of their respective communities means that the availability of even the most widely used C and C++ APIs (e.g., ANSI C, POSIX, STL, zlib) cannot be taken for granted. There may also be restrictions to the language features available. These factors often mean that a library (along with its dependencies) cannot easily be built for a given target platform, or even if it can, it may not fit in well with the existing application codebase.

In some cases one can switch dependencies to available ones. For example, consider a mail client using the GnuTLS library to implement secure sending of emails. For targets on which GnuTLS is not available, one might be able to switch to using OpenSSL with reasonable effort, as the GnuTLS use is likely confined to a small part of the application. There are, however, two cross-cutting concerns affecting perhaps a majority of applications, and these are *event handling* and *error handling*.[1] Both involve libraries and customs not so easily replaced as they permeate the codebase.

Here we only consider error handling, as we believe it worthwhile to focus on a specific, common portability problem before trying to generalize. (We use the word *error* as a general term for any exceptional condition, no matter how reported and handled. By the word *exception* we only mean errors handled using the standard C++ language exception mechanism.) Indeed we feel that some cross-cutting concerns are so commonplace as to warrant first-class language support. *Aspect-oriented programming* (AOP) may offer a general way to address such concerns, but generality tends to come with limitations. Full-blown *model-driven engineering*, on the other hand, comes at a relatively large infrastructure cost [1], and is unlikely to be adopted for addressing a single concern.

Although the ideas discussed here may apply to other languages, we focus on C and C++ because they have no single de facto standard error handling mechanism that everyone uses. The standard C `errno` facility sees limited use beyond the standard library. C++ is often used together with C, and language interoperability requirements then constrain the use of C++ exceptions. Some C++ based toolchains do not even support exceptions (e.g., Arduino, CUDA), and even when supported, it may be a platform convention not to use them (e.g. Bada) or not to use them directly (e.g. Symbian). Hard real time requirements may also preclude their use.

We posit that programmers should not need to fix an error handling mechanism when implementing a library. Rather, there should be tools to specialize a suitably abstract error declaration and handling code at the time when software is configured or built for a given platform. Having such tooling would mean that library implementors would not need to stress over which mechanism to choose, and library users would still not require any obscure tool to build an application against the library; rather, the library could be specialized once for a given platform, and then be distributed as standard C or C++ code.

[1] Interestingly, the two concepts are similar in that both involve reporting and handling (of errors or events). The differences are in flow of control, as error handling typically involves some form of a (possibly non-local) return, while event handling involves a move of control, typically via an event loop, sometimes across threads.

A preprint of 978-1-4673-1766-5/12 © 2012 IEEE
WEH 2012, Zurich, Switzerland
The final publication is available at IEEE Xplore
http://ieeexplore.ieee.org/document/6226598/

II. Requirements and Choice of Abstraction

In specifying requirements for an implementation of a retargetable error handling mechanism, we might accept that when *using* a non-retargetable "legacy" C or C++ library internally within a codebase one would have to explicitly use whatever mechanism is exposed by the library API. But when *creating* a new library, one should have the option of using an abstraction involving enough information to allow for adapting both the implementation and the API for different mechanisms.

In the following discussion we sketch code samples in an imaginary language to illustrate a potentially suitable abstraction, using custom keywords as appropriate to distinguish from C++.

In choosing the abstraction we might downplay the importance of declaring error behavior, as such declarations are little used in C or C++ code. Error conditions for functions are commonly documented in comments, but actual declarations are only supported for C++ exceptions, without being required or statically checked. It seems necessary to know which functions may cause *some* error, and this might have to be declared in some manner (unless static analysis can determine it). We might not go as far as listing all possible errors for functions, let alone having *checked exceptions*.

```
Db* dbOpen() raising { /* implementation */ }
raising dbExec; // for all overloads
raising dbCreateTable(Db*);
```

Individual errors might be declared (once) for the purpose of specifying how to map abstract errors to concrete ones. The latter might name either values or types, as actions conditional on exception object type are common in C++.[2]

```
#if defined(POSIX_ERRORS)
  raisable int ENOMEM NoMemory;
#elif defined(CXX_ERRORS)
  raisable typename std::bad_alloc NoMemory;
#elif defined(SYMBIAN_ERRORS)
  raisable TInt KErrNoMemory NoMemory;
#else
  #error unsupported error mechanism
#endif
```

The error *handling* abstraction should be chosen to map well to the various target mechanisms. A reasonable choice might be the widely known abstraction of **`throw`** and **`try/catch`** like statements. To retain the familiar **`throw`** semantics, many-level propagation of errors would need to be supported, and can be simulated through explicit propagation for concrete mechanisms not involving non-local returns.

[2]Where subtyping is involved, we might require concrete error handling code such that it matches handlers to errors in decreasing type specificity order, as described by Entwisle et al [1].

`throw` and **`try/catch`** are likely to map well to most non-local return based error handling mechanisms. Mechanisms not triggering a break in normal execution are less straightforward targets, as with them it is necessary to insert checks of return values or `errno` or similar reporting "channels". Here is where it becomes useful to know which functions might fail, unless we prefer to annotate every potentially failing function *call* expression.

```
Db* db = dbOpen();
try { again:
  try { dbExec(db, sql); }
  rescue (DbNoTable _) {
    dbCreateTable(db);
    goto again;
  }
} rescue (_ e) {
  dbClose(db); raise e;
}
dbClose(db);
```

As one of the most important error recovery tasks in languages like C and C++ (with manual or semi-manual memory management) is resource cleanup, we might want a cleanup abstraction as well, something mapping reasonably well to both implicit (C++ RAII) and explicit (e.g., Symbian `CleanupStack`) stack based cleanup mechanisms. The abstraction might resemble the **`scope`** statements of the D language, probably supporting at least the usual lexical block scope (as in D).[3] There are other scoping options, however, as e.g. cleanup stacks are separate from the C stack and independent of lexical scope. The cleanup statements would have to be restricted to something possible to express using C++11 lambda functions or similar.

```
Db* db = dbOpen();
scope(exit) dbClose(db);
again:
try { dbExec(db, sql); }
rescue (DbNoTable _) {
  dbCreateTable(db);
  goto again;
}
```

III. Implementation and Use Scenarios

Implementing a mapping to a concrete mechanism should be possible at least through source-to-source translation, requiring a language that compiles down to C or C++. The Java-resembling Vala language, for instance, demonstrates how to target C and GLib's `GError` as its (sole) concrete error handling mechanism. In the `GError` case error information is passed via a function argument; one would get similar code structure using the `errno` "side channel".

[3]For a GCC-specific C implementation one might use GCC's `cleanup __attribute__` to have a cleanup function run on an automatic variable when it goes out of scope.

A preprint of 978-1-4673-1766-5/12 © 2012 IEEE
WEH 2012, Zurich, Switzerland
The final publication is available at IEEE Xplore
http://ieeexplore.ieee.org/document/6226598/

```
errno = 0;
Db* db = dbOpen();
if (errno) return;
again:
{
  errno = 0;
  dbExec(db, sql);
  if (errno == DbNoTable) {
    errno = 0;
    dbCreateTable(db);
    if (errno) goto __cleanup;
    goto again;
  }
  if (errno) goto __cleanup;
}
int __errsave;
__cleanup:
__errsave = errno;
dbClose(db);
errno = __errsave;
```

Challenges in error handling code translation include catering for different mechanisms that can be used for error propagation and resource cleanup. These include: conditionals, **goto**s, **return**s, exceptions, `setjmp`/`longjmp`, and libunwind for propagation (but probably not *continuation-passing style* in the case of C and C++); and direct operations and declared cleanup actions (e.g., RAII, cleanup stack) for cleanup.

Possible use scenarios for retargetable error mechanisms are: generating a C++ library implementation (and API) that does explicit error propagation via extra function arguments (instead of using **throw** statements) to work around a toolchain not supporting C++ exceptions; generating an API (and implementation) that reports errors via Symbian's *leave* mechanism to make the API convenient to use in a Symbian native application; etc.

IV. Related Work

Alerts [2] are a powerful solution as the implementor and user of a library can independently choose the error handling mechanism against which to program. Existing "legacy" libraries can also be retrofitted to support alerts by merely declaring their alert reporting behavior. The solution also includes additional abstractions for alert handling convenience.

The approach advocated in this paper has more modest goals, perhaps making it easier to realize. It also allows for specializing the same library for multiple concrete error reporting mechanisms, giving library users choice. Alert reporting libraries are implemented using a specific concrete error reporting mechanism, and alert-enabled tooling is required throughout the development of an application using a different mechanism as call sites have to be adapted.

The Lua language includes its own error handling abstractions, and its implementation [3] supports C preprocessor based compile-time configuration of the stack unwinding mechanism to use for error propagation. Basic resource cleanup (both after normal and exceptional processing) happens as Lua known but unreachable objects are subject to garbage collection. When programming against Lua's C API one could regard Lua's state object (and the associated virtual stack) as something akin to a cleanup stack.

As Lua is highly portable and has a rich C API, one could claim that it in a sense already provides a way to do portable error handling in C or C++. Our suggested approach has the advantage of not requiring any form of non-local return operation for the error handling implementation. Another notable difference is that our error handling mechanism configuration affects the API as well as the implementation.

AOP has been used for separating error handling concerns into aspects [4], and could hence, in theory, be used to achieve late binding of mechanism-specific error handling code. However, while "aspectization" may work for general error handling such as reporting and terminating execution, realistic software systems have more context-dependent error recovery requirements, with resource cleanup being a prime example. This makes it uncertain whether applying AOP to application-specific error handling is beneficial [5].

Acknowledgments

Thanks to Eva Burrows, Anya Helene Bagge, Magne Haveraaen, and the anonymous referees for advice and improvement suggestions. This work has been funded by the Research Council of Norway.

References

[1] S. Entwisle, H. Schmidt, I. Peake, and E. Kendall, "A model driven exception management framework for developing reliable software systems," in *Proceedings of 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, Hong Kong, Oct. 2006, pp. 307–318.

[2] A. H. Bagge, V. David, M. Haveraaen, and K. T. Kalleberg, "Stayin' alert: Moulding failure and exceptions to your needs," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, Portland, Oregon, Oct. 2006.

[3] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The implementation of Lua 5.0," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, Jul. 2005.

[4] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, Jun. 2000, pp. 418–427.

[5] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira, "Exceptions and aspects: The devil is in the details," in *Proceedings of the 14th International Symposium on the Foundations of Software Engineering (FSE)*, Portland, Oregon, Nov. 2006, pp. 152–162.