#90-0011

Exception Handling for C++: An Opinion

Bob Fraley
Hewlett-Packard

March 7, 1990

The paper "Exception Handling in C++" by Koenig and Stroustrup caused some concern about the intended use of the facility and how to best achieve that purpose. This paper presents some ideas on exception handling, but falls short of proposing a complete exception facility. Note that the author has not tried to implement these ideas within a C++ environment.

The first question is: what is the purpose of an exception handling facility? The proposal follows along the lines which are popular these days: post a handler that will intercept an exception notification from any procedure level within the indicated computation. One must ask why this is being done. For example, a library routine might return an array out of bounds exception. The caller, who is totally unfamiliar with the implementation of the library routine, has no idea as to why this exception occurred, and generally doesn't know whether it occurred in a high-level procedure or 27 procedure calls deep within its implementation. Such a caller is unlikely to know how to respond differently to an array bounds violation, a division by zero, or a user-defined "symbol not found in table" exception in such a situation. The only purpose for distinguishing the type of exception is to allow the caller to serve as an error reporting facility for the library routine. It seems unreasonable to ask each calling program to create a library diagnostic facility.

Taking an even higher-level view of what is happening, the only reason that the calling program needs to intercept an exception generated 27 procedure calls deep within a library is that the author of the library routine was sloppy and didn't take care of the errors within the code. Rather than providing a facility that will help the caller deal with the library writer's mess, it might be better to provide the library writer with a facility that will help in cleaning up the mess.

Some situations that occur within a library routine need to be reported to the caller. These seem to fall into three categories:

1. Something happened, and the function which you asked for can't be completed.

2. A failure of some specific sort occurred.

3. Several actions are possible, and there is insufficient information to select one.

The first form can be used for any failure, and should be used when the other two forms are inappropriate.

The second form only makes sense to report to the caller if its meaning can be expressed in terms that the caller can understand. In other words, it must be explainable in terms of the abstraction that the called object is providing. An exception "Too many input values" can be understood by the user of a table handling package, while "Array reference out of bounds" could be caused by some programming error unrelated to the user's view of the function, perhaps in an array unknown to the caller.

The third form is characteristic of the situation where the library routine

is capable of resuming execution if some additional information can be provided. This is dismissed in the Koenig and Stroustrup paper, but will be considered here.

Why do we need an exception handling facility at all? It is used to simplify software. One could imagine a pointer dereference syntax that requires the programmer to specify a statement that will be executed if the pointer is null; one cannot imagine wanting to program in such a language or read programs written in it. The same can be said for needing to explicitly specify the error action on add, divide, and assignment operations. As the code becomes higher level, the same can be said for the use of library packages.

There are two objectives of the exception handling facility. The first is to make software more reliable through improved handling of error situations. The second is to do so conveniently. To assist in the first task, the exception facility should help the programmer realize what exceptions can occur and whether they have been handled.

Abstraction of Exceptions

For C++, it is particularly important to constrain the exceptions which can occur within a specific procedure. For example, a programmer may be writing software knowing only the abstract class, not the various implementations of that class that might be called, since there may be many and some might not be written yet. Without standardization of potential exceptions provided by the abstract class definition, the program writer won't know what exceptions need to be considered. Furthermore, similar situations could arise in different implementations but be given different names, so that a program that is capable of responding to an exception won't be able to do so.

One might ask why the caller should even be concerned about the possible exceptions. They can just be allowed to pass upward, and not be handled at this level. Someone else can take them. Many times, however, the program close to the exception can deal with the situation in a better manner. At the specific point that the exception occurred, it might be irrelevant. The function could not be done; too bad. The main objective of the current function can still be achieved.

Exceptions vs Debugging

For many, the notion that many different exception situations create a single exception condition, as would be necessary when an abstract class dictates the available exception conditions, is undesirable. If these error situations are all lumped into one, then how would an expert find out what is wrong? For me, this is not the purpose of exception conditions. The exception is raised because the called routine was not able to carry out the requested operation, and the caller is being notified so that it can recover in the best manner possible.

For those who want debugging information, the language system should provide a debugging log. This would be used to record debugging information. By making this a part of the language, there would be a single source of information available to diagnose failures. It might be important to tie the debugging log to the exception handling facility if procedure frames would be thrown away, recording those frames for later analysis. However, the diagnostic facility will not be discussed further in this document.

Exception as a Message

The referenced paper asks whether an exception is a class or an object. Another possibility is that an exception is a function invocation, and that the message

handler is a function. What, then, is being declared as an exception? It is a virtual function. One might suggest that it is a pointer to a function, but as with virtual functions the pointer mechanism is hidden from the user. We shall regard an exception as an invocation of a function.

The notion of exception as a function invocation raises an interesting issue: what about the parameters to the exception? These are dismissed by the referenced paper, with some suggestions as to how to deal with their omission; here they are a natural part of the exception handling process. Parameters can be passed to communicate to the handler the specific values which raised the exception (as in a floating point overflow). Likewise, a return value can be specified when the exception returns to the point that it was raised, as in returning a specific NaN value as the result of a floating point overflow.

Structuring the Language Features

Let's look at the three types of exceptions, and discuss how they could be handled.

3. Several actions are possible, and there is insufficient information to select one.

An example of this type of exception is the floating point underflow. In performing a floating point add, the exponent could become too small to represent a normalized answer in the machine's representation. The choices are:

> Represent the result as a denormalized number and continue the computation.
>
> Represent the result as 0, and continue the computation.
>
> Record the fact that the error occurred (say in a variable) and continue the computation in one of the ways indicated above.
>
> Terminate the computation with an exception condition.

This is best handled as an exception having parameters, which might be used to compute the denormalized number, or might be recorded in the error log. The result must be passed to the point of invocation in the first 3 cases. The role of the exception handler should be to allow the caller to specify the handler without needing to code it for each floating point operation.

An important design question for the exception facility is whether the handler should be chosen by a lexical specification (where one can look at the software near the floating operation and determine whether which handler will be invoked), or a dynamic specification (where different invocations of the containing procedure might have different handlers). Unfortunately, the floating point community is divided on this issue, but quite often the algorithm will be dependent on the behavior of the handler, so a lexical technique seems to be more appropriate.

The try/catch construct is rather wordy for this type of exception, since it must be specified in each procedure body. More appropriate would be a declaration at the compilation unit or class level. A single declaration could then be used within many procedure bodies. There could be a performance issue if several exception routine pointers need to be set for each procedure call, so the language design should allow the caller to understand the global settings needed by the called program, so that only those that differ need to be set.

Another of the exception types is:

2. A failure of some specific sort occurred.

In this case the function has terminated, and cannot resume. The exception handler needs to divert execution from code which follows the function invocation, take appropriate actions, and then resume execution at an appropriate point or raise the exception to a higher level if appropriate. The exception must be handled in he function from which the invocation occurred, since only it can be aware of the calling context. A lexically scoped handler is again appropriate, since the existence of this exception situation should not be known outside of the procedure. Parameters could be included with the exception message, but no return (and therefore result) is possible.

The final case of exception handling:

1. Something happened, and the function which you asked for can't be completed.

This type of exception happens when there is no handler for an exception, or when the program cannot handle a situation and there is no specific exception available in its abstraction that is appropriate. The compiler ought to be able to detect that no handler is available, and give a warning, but execution might take place anyway. It would be appropriate for this type of failure to be handled dynamically, with the exception being passed up multiple levels in the call chain until a handler is found. This exception shall be called the "Uncaught" exception. Since the exception could strip many stack frames, it needs to be tied with the debug log facility to save the frames for later diagnostic investigations.

Declaration of Exceptions

In order to maintain the strong typing of C++, exception handler parameter structures must be declared. In addition, the exceptions that may be raised by a procedure must be declared. The syntax is left as an exercise for the reader.

Constructors and Destructors

Constructors and destructors are functions. They operate like other functions. Destructors can never declare exception returns, so they may only raise the Uncaught exception. This could occur while processing another exception, in which case the original exception processing is abandoned.

Constructors are more interesting. Constructors are not called directly, but are (effectively) called by the function "new" to which they are passed as a function parameter. "new", of course, accepts any function of any parameter structure and (we'll presume) any exception returns, so it is a bit different from other functions. The invocation of "new" leads to the invocation of the constructor, which can fail. This exception may be passed to the point of call of the "new" function. A question: can any procedure that has a function parameter raise to its caller the exceptions stated for the function parameter?

Debuggers

The debugger should indicate its interest in stopping if a particular exception (or any exception) occurs. This will reduce the difficulty of determining whether a handler already exists for the exception, and will allow the debugger to intervene even if a handler does exist.

Nested Exception Handlers

Each handler applies to all handlers contained within its scope. This is

true of the try--catch statement, where a later catch applies to all previous catch statements, and to globally declared handlers, where the first declared applies to all later ones. No handler loops are possible.

A raise statement always terminates the current function execution, even if it is contained within a try statement having a catch for the same condition.

Grouping of Exceptions

Grouping of exceptions does not seem necessary. In the conventional exception handling situation, the file system might indicate one of 57 conditions that has made it impossible to read a record, and the program is only interested in 2: end of file, and file error. In this case grouping would be essential. Once a failure logging facility is provided, the need for reporting such detail upwards disappears and only a few exception types should remain.

Of course, a single handler can handle multiple exception types (if they have the same parameter structure). There should be some way to identify the actual exception which took place. (Should there be a "Catch anything and ignore the parameters"?)

Exception Naming

Naming conventions must be put in place. A probable method would be to use standard C++ naming, where exceptions declared within a class public area are called Class::exception.

Multiple Processes

The exception handlers for one process should be distinct from those for another process. This is even true if an Unhandled exception leaves the procedure which contains the fork which created it.

Summary

This paper proposes in principle a mechanism for handling exceptions in C++. It differs from Koenig and Stroustrup in using lexical scoping for the declaration of many handlers, and it allows parameters to be passed to handlers and results to be returned to the point of exception. It preserves the notion of abstraction which class declarations provide by limiting the implementor to raising only those exceptions which are permitted by the class declaration. While some exception situations would become more wordy, the lexical scoping can actually reduce the wordiness of some constructs.