WG14 N1841 - Alternate Exception Handling Syntax for C

Notes for TS 18661 Part 5 Supplementary attributes

What syntax should be used to express IEEE 754-2008 alternate exception handling in C?

The C Floating Point group working of TS 18661 has developed two principal approaches and seeks advice from the C committee as to which of these two, or some other, would be more productive to develop in detail. What follows is the background, necessary semantics, and discussion of try/catch and #pragma approaches.

BACKGROUND - How fast you can drive a car depends on how fast you can stop

Alternate Exception Handling is specified in Chapter 8 of IEEE 754-2008. It's a series of recommendations to programming environments about what kinds of facilities to provide in languages for portable numeric programming. Some of these recommendations have to do with changes of control flow, and the best syntax to propose for these is the subject of this note.

The overall idea is that the normal case should go as fast as possible, so testing for exceptional conditions should not slow it down. Likewise the normal case should terminate as quickly as possible when exceptional conditions arise, lest (particularly in a loop terminated by a floating-point conditional test) it get into an expensive or infinite loop.

The previous IEEE 754-1985 recommended traps on exceptions. These are implemented in most IEEE 754 hardware systems, but they are so low level that they are not used much except for debugging, where they can be easily used to stop the program altogether. It is not possible to write portable code to handle a hardware trap and continue in any useful way. But if free testing for exceptions is built into the hardware without slowing down the normal case, how can we make it available to the application programmer?

Thus 754-2008 recommends alternate exception handling at a level that is meaningful to the application programmer, leaving the details of hardware trapping to the compiler, run time library, and operating system. It is also possible to implement 754-2008 alternate exception handling without hardware traps, but that might entail significant performance penalties for either the normal case or exceptional case. Exceptions can be detected inline for each operation with tests on operands and results and flags - slowing down the normal case - or at the end of the try clause with flags - slowing the detection of the exceptional case (but testing flags alone does not detect exact underflow exceptions). Instead of traps, some existing hardware such as PowerPC can branch conditionally on floating-point exceptions.

SEMANTICS

So from the higher point of view of an applications programmer, rather than the lower point of view of a system implementer, the common programming exception-handling paradigm is naturally expressed in many languages in this fashion:

```
try {
 normal case code...
}
catch (floating-point-exceptions1) {
 exceptional case 1 code...
}
catch (floating-point-exceptions2) {
 exceptional case 2 code...
}
 following code...
```

The normal case code is executed; if any of the exceptions specified in catch clauses arises, the normal case code is interrupted as quickly as possible (and thus variables written and other state modified as side effects may be indeterminate) and the corresponding exceptional case code is executed. In either case, after the normal case terminates normally or the exceptional case terminates, the "following code" is executed. Note that there might be more than one catch clause to catch more than one group of exceptions to be handled, though typically there is only one catch clause; the examples that follow show only one but there could be more. Note that exceptions can be lists: in some applications FE_UNDERFLOW|FE_OVERFLOW are handled together to invoke a scaling version of the normal case code; in other applications FE_INVALID|FE_DIVISIONBYZERO|FE_OVERFLOW are handled together as errors, perhaps by breaking to a higher level or perhaps by aborting altogether.

Sometimes the possibility of interrupting computation and creating an indeterminate state is worth avoiding; one can imagine having an additional kind of catch keyword such as "catchafter" or "patchup" to be used in those cases to insure that the try clause completed before exception testing.

IEEE 754 specifies default exception handling, but if the default were always satisfactory then it wouldn't be exceptional. By definition, an exception can't be handled the same way in every situation in which it might arise. So alternate exception handling must be specified from time to time, more often in code that is intended to be as fast as possible in the normal case, but as robust as possible when exceptions arise which are not normal, but are not so rare that they can be ignored. So some desirable attributes of exception handling include

* The normal case should be as fast as possible.
* The exceptional case should be detected as quickly as possible.
* The application programmer can understand the exception and its handling in terms that do not depend on the specific hardware and operating system, so the syntax and semantics are portable.
* The compiler/operating system/hardware figure out the most efficient way to implement the intended semantics on a particular platform, rather than leaving that burden on the application programmer.

Like other floating-point environment, the exception environment is inherited from the parent thread and then thread-local (C Standard 7.6).

SYNTAX

There are many different ways to express the syntax. They all have in common a need to specify

* the normal case code that the exception handling applies to
* the exceptions to be handled specially
* the exception case code to be executed when those exceptions arise

There are versions of try/catch in standard C++, java, C#, perl, php, python, visual basic, and matlab. These are all very similar but each slightly different. There are also try/catch extensions in at least two Fortran compilers targeting .NET applications.

NetRexx has a somewhat more different version; any do/end compound statement may have one or more catch clauses; there is no explicit try keyword. Ruby has a similar structure but the catch keyword is "rescue". Avoiding an explicit try clause reduces the number of gratuitous nesting levels.

Naturally, for C, the closest analog is C++. C++ catch clauses have as arguments a parameter and its type. The parameter is set by the system for standard exceptions and by the programmer for programmer-defined exceptions generated with the throw() function; the catch clause is matched to the throw by the type of the thrown object. There is also a catch(...) syntax to catch all unspecified exceptions.

Unlike most of the exceptions for which try/catch was designed, some floating-point exceptions arise frequently - inexact especially and underflow in some programs - and the default treatment of continuing execution with a prescribed result and raising a flag should be in force if no alternate treatment is specified in a catch clause.

Thus several aspects of C++ exception handling are not needed to support 754-2008 floating-point alternate exception handling:

* no need to throw()
* no need to catch(...) unspecified exceptions
* catch clause is matched by exception rather than type -
  in standard C, exceptions receive names in <fenv.h> like FE_INVALID

Furthermore there is another difference important for performance: whereas in C++ the code in the try clause is compiled without reference to subsequent catch clauses, for floating-point purposes it is usually desirable or necessary to know which exceptions are to be trapped when generating the code for the try clause. If the implementation is to be by traps, a trap handler has to be set up before the try clause and torn down on exit; if the implementation is to be by conditional branches on operands and results, that code has to be generated inline in the try clause. As a consequence, a strictly one-pass C compiler would have to exploit hardware traps in order to implement alternate exception handling fully and efficiently. Otherwise by only testing exception flags at the end of the try clause, such a compiler would not be able to promptly terminate an exceptional loop until it had perhaps run for a long time or infinitely. Nor could it properly test for the underflow exception: a little-appreciated but important requirement of 754-1985 and 754-2008 is that exact underflow conditions must be detected by trapping (1985) or alternate exception handling (2008) if enabled, even though the underflow flag is not raised (and thus can't be used to detect exact underflow).

Thus syntactically, a compiler might prefer that the catch clause for floating-point exceptions be seen before the try clause rather than afterward, as it is in most languages. However the conventional try/catch paradigm properly puts the normal case first syntactically, for the benefit of the programmer who writes and the programmer who reads the code and must understand the normal case before delving into the exceptional case. One can imagine having the catch before, in which case the keyword should perhaps be different:

```
 catch_fe( exception ) {
  exceptional case code...
 }
 try {
  normal case code...
 }
```

or having it within the try clause:

```
 try {
 catch_fe( exception ) {
  exceptional case code...
 }
  normal case code...
 }
```

One result of preserving some form of try/catch syntax is that floating-point exception handling could be added to C++ with less syntactic effort than other possible approaches, and most of C++ exception handling could be added to C with less syntactic effort than other possible approaches. If that compatibility were deemed to cost more than its benefit, then there are other alternatives.

Since the explicit try clause is not really needed, one could define catch_fe clauses for any compound statement delimited by curly brackets, and instead of adding a catch_fe keyword, it could be a variety of pragma:

```
{
 normal case code...
{
#pragma STDC CATCH_FE exception
 exceptional case code...
}
}
```

It's exactly the same try/catch semantics that we started with, but in syntax not particularly close to C++ or anybody else's try/catch. However it does match other #pragma's having scope in compound statements or in files, that are already in C, and are already in previous floating-point extension reports going through the approval process, and that are being considered to control other aspects of floating-point code generation.

Other syntaxes that have been used in the past include:

PL/I:

```
 ON exception BEGIN; ... END;
```

Basic:

```
 ON ERROR GOTO label
 ON event GOSUB label
```

Rexx:

```
 SIGNAL ON exception;
```

but these don't seem to offer any advantage.

While the C Floating Point group working of TS 18661 could pursue any of these directions and report the results to the C committee, we'd appreciate early feedback and direction from the C committee as to which seems the most acceptable, or whether there's some better C-like way to express the semantics. The semantic content is pretty much the same with all, but the syntax possibilities vary widely.

Note: Can adequate floating-point exception handling be handled without any syntactic support in C? Most of what is desired can be expressed with the existing floating-point flag functions or by enabling SIGFPE handlers for hardware floating-point traps. However the flags functions are very verbose and tend to obscure the underlying logic of the applications, and even a SIGFPE handler that does nothing but abort or longjump has a lot of implementation-dependent handling of the trap details; one that does anything more complicated might end up decoding op codes, once again obscuring the application logic. Worse, the C standard 7.14.1.1 specifies that the signal function behavior is undefined in the presence of multiple threads. Worst, the application programmer must decide whether coding with flags or traps is best, even though that system-dependent decision might better be left to the compiler and runtime.

So even though existing standard C has support for detecting exception flags and handling SIGFPE, that's sort of like saying that Fortran-77 supports a heap. It is true that any competent Fortran-77 programmer could program blank common to be used like a heap. It is also true that the result is difficult to write correctly and even harder to read and debug. C provides malloc and free instead. Fortran-90 eventually caught up to C in this respect.