Method for extended C++ exception features to handle OS signal exceptions

Disclosed is a method for extended C++ exception features to handle operating system (OS) signal exceptions. Benefits include improved functionality, improved reliability, and improved portability.

Background

The C++ programming language is standardized by a specification titled "`Programming languages - C++`", 14882:1998, dated September 1998 by American National Standards Institute, Inc.

Conventionally, the C++ programming language provides intrinsic exception handling features. However, the features are somewhat limited in capability. The only exceptions that can be processed (caught) are those explicitly issued (thrown) by the program. No intrinsic support is provided for unexpected errors that are signaled by hardware or the operating system.

Each `catch` block handles a particular type of exception that can be raised by a C++ `throw` operator code in the `try` block. For example, line 13 of the `try` block checks a divisor. If its value is zero, an exception of type `div_BY_zero` is issued by the `throw` operator in the ":" clause of the conditional assignment statement. The exception thrown on line 13 is processed in the `catch` block at line 15, which delivers the appropriate message (see Figure 1).

When first introduced to the intrinsic structured exception handling of C++, many programmers hope they can use the `try` and `catch` features to handle errors encountered by the hardware or operating system. The hope is that the operating system and C++ runtime library will `throw` special types of exceptions in response to hardware and operating system exceptions encountered inside `try` blocks and that these special exceptions can be handled in special types of `catch` blocks. However, this capability is beyond standard C++ exception handling features.

C++ exception handling can only be used by C++ code. For example, exceptions cannot be thrown by C routines called from the same `try` block. Additionally, the capability to `throw` exceptions does not pass through non-C++ routines to downstream C++ routines. For example, if a C routine is called from within a `try` block and subsequently calls a C++ routine, the called C++ routine cannot `throw` exceptions.

General description

The disclosed method extends the functionality of C++ exception handling to include limited capabilities to handle six types of exceptions that are signaled by hardware and the operating system. The six exceptions are defined by the ANSI C++ programming language as the following signals:

- SIGABRT - Abort
- SIGFPE - Hardware detected arithmetic exceptions
- SIGILL - Hardware detected illegal instruction exceptions
- SIGINT - Interactive attention signal
- SIGSEGV - Hardware detected exceptions caused by invalid memory accesses
- SIGTERM - Signals delivered by the operating system to terminate executing programs

Advantages

The disclosed method provides advantages, including:

- Improved functionality due to providing extended C++ exception features to handle six OS signal exceptions
- Improved reliability due to increasing the types of errors detected and processed
- Improved portability due to being implemented in ANSI C++ and using similar semantics

Detailed description

The disclosed method includes extended C++ exception features to handle OS signal exceptions.

SIGABRT signals are delivered in response to calls to the abort() C++ runtime system service. This system service is used to deliberately and immediately terminate a program.

SIGFPE signals are delivered in response to hardware detected arithmetic exceptions. SIGILL signals are delivered in response to hardware detected illegal instruction exceptions. These exceptions may also be caused by attempts to use privileged instructions from non-privileged modes.

SIGINT signals are delivered in response to an interactive attention signal. It is typically created by holding the Ctrl key down while pressing the C key.

SIGSEGV signals are delivered in response to hardware detected exceptions caused by invalid memory accesses.

SIGTERM signals are delivered by the operating system to terminate executing programs.

Any of the six ANSI signals can be delivered by the C/C++ run-time library raise routine or by other hardware or software events regardless of the computer language a routine is written in. If any one of the six ANSI signals is delivered inside the scope of a __Try block, the signal is caught by the corresponding __Catch block, regardless of the languages used or the method of signal delivery.

The __Try/__Catch mechanism is semantically similar to the intrinsic C++ exception handling. When an exception condition is realized and the exception is thrown, no method enables the program to return to the site of the error and fix it. Both the conventional and disclosed mechanisms are designed to report exceptions, not to correct and retry them.

Sample implementation

A sample implementation illustrates the disclosed method (see Figure 2).

On line 11, the `__Try` operator establishes mechanisms to receive the six ANSI-specified signals described above and to `throw` them as C++ executions of type `signal_exeception`. On line 13 of the `__Try` block, a floating point division operation is performed. If the divisor is zero, and floating point exception hardware is enabled, a `SIGFPE` signal is raised. Mechanisms of the `__Try` operator on line 11 receive the `SIGFPE` signal and `throw` a C++ exception of type `signal_execption`. The exception thrown on line 11 is caught in the `__Catch` block at line 16, and an appropriate message is produced.

The exception condition is unknown until hardware discovers it in the `__Try` block. In response to this discovery, a signal is raised and delivered as a `signal_exception` type exception using a `throw` operator inside the `__Try` block. The exception thrown in the `__Try` block is handled in the `__Catch` block.

The `__Try`, `__EndTry`, `__Catch`, and `__EndCatch` operators are implemented as C++ preprocessor macros (see Figure 3).

Lines 1 through 10 are exactly as in Figure 2. Lines 11 through 11.10 extend the `__Try` macro. Lines 12 through 14 are unchanged. Lines 15 and 15.0 extend the `__EndTry` macro. Lines 16 and 16.0 extend the `__Catch` macro. Lines 17 through 19 are unchanged. Lines 20 through 20.4 extend the `__EndCatch` macro.

The opening brace (‘{‘) character on line 11.0 is closed by the closing brace character (‘}’) found at the end of line 20.4. They define a single scope of reference that includes the `__Try` and `__EndCatch` operators. The following variables defined at lines 11.1 through 11.4 are local to that scope:

- `new_try_jmp_context`
- `old_try_jmp_context`
- `new_try_sigaction`
- `old_try_sigaction`

The C++ `try` block extends from the line opening brace character (‘{‘) on line 11.6 to line the closing brace character (‘}’) on line 15.0. The C++ `catch block` of type `signal_exception` extends from the opening brace character (‘{‘) on line 16.0 to the first closing brace character (‘}’) on line 20.0.

The variable `new_try_jmp_context` (defined at line 11.1) is a pointer set to the `setjmp/longjmp` context buffer for use by the current `__Try` block. The variable `old_try_jmp_context` (defined at line 11.2) saves a pointer to the currently active context buffer. A global context block pointer is shared by all `__Try blocks`, so it must be saved by each `__Try` macro and restored by each `__EndCatch` macro.

The variable new_try_sigaction (defined at line 11.3) is pointer set to a global table of function pointers. Each entry defines a handler for a signal number that corresponds to the table entry. The variable old_try_sigaction (defined at line 11.4) saves and restores the old signal handler function pointers. The SetSignalHandlers function (called at line 11.5) saves function pointers to the old signal handlers (stored in old_try_sigaction) and establishes the functions addressed in new_try_sigaction as the new signal handlers.

The C++ try block is entered at line 11.6. At line 11.7, the global jmp_buf pointer (try_jmp_context) is set to point at new_try_jmp_context. At line 11.8, setjmp initializes the global jmp_buf variable (try_jmp_context) for use by a longjmp context buffer from one of the new signal handlers. The setjmp routine sets the status variable to zero.

Because the status variable is zero, control passes into the code block opened by the opening brace character ('{') at line 12. The divide operation at line 13 is executed. If it succeeds without error, control passes to line 20.1 just after the closing brace character ('}') that terminates the catch block opened at line 16.0. Otherwise, a SIGFPE signal is delivered and processed by the signal handler established earlier in the SetSignalHandlers routine (at line 11.5). The signal handler issues a longjmp, using try_jm_context and SIGFPE as arguments. The longjmp context buffer returns control to line 11.8, setting the status variable to SIGFPE.

Control passes through line 11.9 to line 11.10, where the throw operator delivers a C++ exception of type signal_exception. Control passes to the catch operator block at line 16.0. At line 18, a message is printed that displays the numerical value of the SIGFPE signal number.

Regardless of whether an exception is thrown or not, ResetSignalHandlers (at line 20.1) executes and restores the original signal handlers. The global jmp_buf pointer is restored from the old_try_jmp_context variable (at line 20.2). The new_try_jmp_context variable is deleted at line 20.4. Control leaves the code block opened at lines 11.0 at line 20.4.

Implementations of the SetSignalHandlers (at line 11.5), the ResetSignalHandlers (at line 20.1), and the SignalHandler appear below.

The SetSignalHandlers routine saves the old handler and installs a new handler for each of the six ANSI signals (see Figure 4).

At lines 5 through 8, the C++ run-time library routine signal obtains its arguments, which are corresponding signal numbers from the global table of signal numbers (try_signals) and new handlers from the global table of signal handlers (new_sigaction). The routine returns corresponding old signal handlers to the old_sigaction handlers table.

The ResetSignalHandlers routine is similar to the SetSignalHandlers routine (see Figure 5).

The ResetSignalHandlers routine restores the signal handlers that were installed when the __Try block was entered. Lines 4 through 7 invoke the C++ signal run-time library routine. Its arguments are corresponding signal numbers from the global table of signal numbers (`try_signals`) and old signal handlers from the table of signal handlers (`old_sigaction`). Unlike the `SetSignalHandler` routine, `ResetSignalHandler` ignores the value returned by `signal`.

The `SignalHandler` routine handles the six ANSI signals (see Figure 6).

All six signals are handled the same way. The switch statement at line 3 accepts any signal that arrives. If the arriving signal is one of the six ANSI signals, control passes to line 11, where the `longjmp` operation transfers processing to the statement in the most recent `__Try` block where the `setjmp` was issued (line 11.8 in Figure 3). The `longjmp` operation uses the global `jmp_buf` structure as its first argument and the `signal_type` value as its second argument. The `longjmp` operation sets the status variable (in line 11.8 of Figure 3) to the value of `signal_type`. Control passes to line 11.9 of Figure 3.

```
double i,j,k;
class div_BY_zero
   {
   public:
   double divisor;
   double dividend;
   div_BY_zero( double i, double j ) { dividend = i; divisor = j; };
   virtual ~divide_by_zero() {};
   };
...
try
   {
   k = ( j != 0.0 )? i/j : throw divide_by_zero(i,j);
   }
catch( div_BY_zero exception )
   {
   printf( "Divide by zero exception: dividend = %d : divisor = %d \n",
           exception.dividend,
           exception.divisor   );
   };
```

Fig. 1

```
double i,j,k;
class signal_exception
   {
   public:
   int type;
   signal_exception( int i ) { type = i; };
   virtual ~signal_exception() {};
   };
...
__Try
   {
   k = i/j;
   }
_EndTry
__Catch( signal )
   {
   printf( “An exception of type %d was signaled.\n”, signal.type );
   };
__EndCatch;
```

Fig. 2

```
    double i,j,k;
    class signal_exception
       {
       public:
       int type;
       signal_exception( int i ) { type = i; };
       virtual ~signal_exception() {};
       };
    ...
  //__Try
       {
       jmp_buf *new_try_jmp_context  = new jmp_buf[sizeof(jmp_buf)];
       jmp_buf *old_try_jmp_context  = try_jmp_context;
       sigaction_fp *new_try_sigaction = try_sigactions;
       sigaction_fp old_try_sigaction[ TRY_NSIG ];
       SetSignalHandlers( new_try_sigaction, old_try_sigaction );
       try {
          try_jmp_context = new_try_jmp_context;
          int status = setjmp( try_jmp_context );
          if ( status )
                throw signal_exception( status );
       {
       k = i/j;
       }
    //_EndTry
       }
  //__Catch( signal )
       catch( signal_exception signal ) {
       {
       printf( "An exception of type %d was signaled.\n", signal.type );
       };
  //__EndCatch;
       }
       ResetSignalHandlers( old_try_sigaction );
       try_jmp_context = old_try_jmp_context;
       delete [] new_try_jmp_context;
}
```

Fig. 3

```
void SetSignalHandlers( sigaction_fp *new_sigaction,
                        sigaction_fp *old_sigaction )
   {
   int i;
   for ( i = 0; i < TRY_NSIG; i++ )
   {
   old_sigaction[i] = signal( try_signals[ i ],new_sigaction[i] );
   }
   return;
   }
```

Fig. 4

```
void ResetSignalHandlers( sigaction_fp *old_sigaction )
   {
   int i;
   for ( i = 0; i < TRY_NSIG; i++ )
      {
      signal( try_signals[ i ],old_sigaction[ i ] );
      }
   return;
   }
```

Fig. 5

```
void SignalHandler( int signal_type )
   {
   switch( signal_type )
      {
      case SIGILL:
      case SIGSEGV:
      case SIGTERM:
      case SIGABRT:
      case SIGFPE:
      case SIGINT:
         longjmp( try_block_context, signal_type );
         break;
      }
      return;
   }
```

Fig. 6

Disclosed anonymously