CHAPTER

# 4 Exception Handling

Windows Structured Exception Handling (SEH) is the principal focus of this chapter, which also describes console control handlers and vectored exception handling.

SEH provides a robust mechanism that allows applications to respond to unexpected asynchronous events, such as addressing exceptions, arithmetic faults, and system errors. SEH also allows a program to exit from anywhere in a code block and automatically perform programmer-specified processing and error recovery. SEH ensures that the program will be able to free resources and perform other cleanup processing before the block, thread, or process terminates either under program control or because of an unexpected exception. Furthermore, SEH can be added easily to existing code, often simplifying program logic.

SEH will prove to be useful in the examples and also will allow extension of the `ReportError` error-processing function from Chapter 2. SEH is usually confined to C programs. C++, C#, and other languages have very similar mechanisms, however, and these mechanisms build on the SEH facilities presented here.

Console control handlers, also described in this chapter, allow a program to detect external signals such as a `Ctrl-C` from the console or the user logging off or shutting down the system. These signals also provide a limited form of process-to-process signaling.

The final topic is vectored exception handling. This feature allows the user to specify functions to be executed directly when an exception occurs, and the functions are executed before SEH is invoked.

## Exceptions and Their Handlers

Without some form of exception handling, an unintended program exception, such as dereferencing a `NULL` pointer or division by zero, will terminate a program immediately without performing normal termination processing, such as deleting temporary files. SEH allows specification of a code block, or *exception handler*, which can delete the temporary files, perform other termination operations, and analyze and log the exception. In general, exception handlers can perform any required cleanup operations before leaving the code block.

Normally, an exception indicates a fatal error with no recovery, and the thread (Chapter 7), or even the entire process, should terminate after the handler reports the exception. Do not expect to be able to resume execution from the point where the exception occurs. Nonetheless, we will show an example (Program 4–2) where a program can continue execution.

SEH is supported through a combination of Windows functions, compiler-supported language extensions, and run-time support. The exact language support may vary; the examples here were all developed for Microsoft C.

## Try and Except Blocks

The first step in using SEH is to determine which code blocks to monitor and provide them with exception handlers, as described next. It is possible to monitor an entire function or to have separate exception handlers for different code blocks and functions.

A code block is a good candidate for an exception handler in situations that include the following, and catching these exceptions allows you to detect bugs and avoid potentially serious problems.

- Detectable errors, including system call errors, might occur, and you need to recover from the error rather than terminate the program.
- There is a possibility of dereferencing pointers that have not been properly initialized or computed.
- There is array manipulation, and it is possible for array indices to go out of bounds.
- The code performs floating-point arithmetic, and there is concern with zero divides, imprecise results, and overflows.
- The code calls a function that might generate an exception intentionally, because the function arguments are not correct, or for some other occurrence.

SEH uses "try" and "except" blocks. In the examples in this chapter and throughout the book, once you have decided to monitor a block, create the try and except blocks as follows:

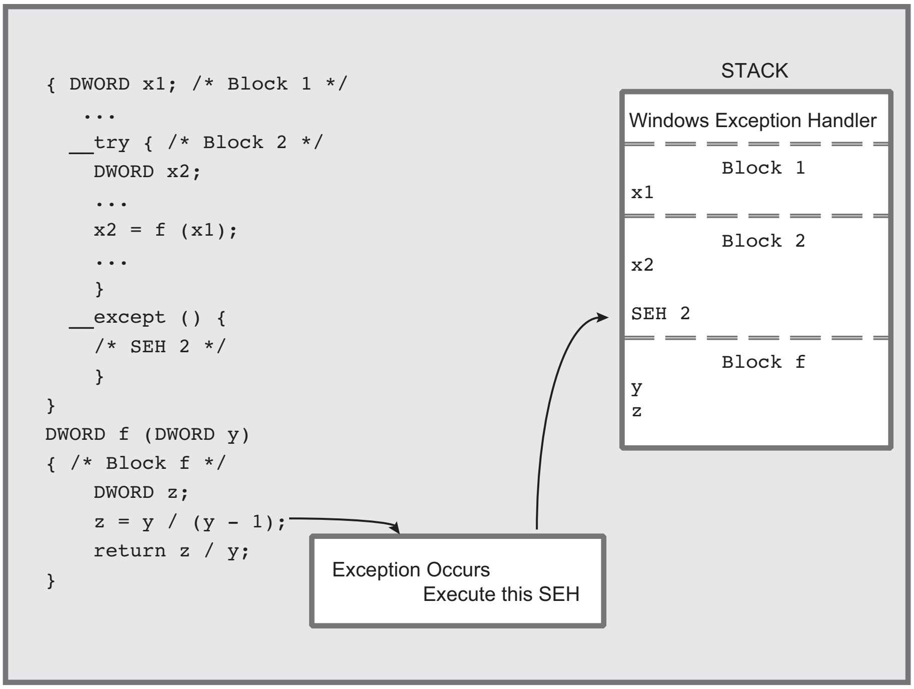
```
 __try {
      /* Block of monitored code */
}
 __except (filter_expression) {
   /* Exception handling block */
}
```

Note that `__try` and `__except` are keywords that the C compiler recognizes; however, they are not part of standard C.

The try block is part of normal application code. If an exception occurs in the block, the OS transfers control to the exception handler, which is the code in the block associated with the `__except` clause. The value of the *`filter_expression`* determines the actions that follow.

The exception could occur within a block embedded in the try block, in which case the run-time support "unwinds" the stack to find the exception handler and then gives control to the handler. The same thing happens when an exception occurs within a function called within a try block if the function does not have an appropriate exception handler.

For the x86 architecture, Figure 4–1 shows how an exception handler is located on the stack when an exception occurs. Once the exception handler block completes, control passes to the next statement after the exception block unless there is some other control flow statement in the handler. Note that SEH on some other architectures uses a more efficient static registration process (out of scope for this discussion) to achieve a similar result.



```
{ DWORD x1; /* Block 1 */
   ...
  __try { /* Block 2 */
    DWORD x2;
    ...
    x2 = f (x1);
    ...
    }
  __except () {
    /* SEH 2 */
    }
}
DWORD f (DWORD y)
{ /* Block f */
    DWORD z;
    z = y / (y - 1);
    return z / y;
}
```

**Figure 4–1** SEH, Blocks, and Functions

## Filter Expressions and Their Values

The *`filter_expression`* in the `__except` clause is evaluated immediately after the exception occurs. The expression is usually a literal constant, a call to a *filter function*, or a conditional expression. In all cases, the expression should return one of three values.

1. `EXCEPTION_EXECUTE_HANDLER`—Windows executes the except block as shown in Figure 4–1 (also see Program 4–1).
2. `EXCEPTION_CONTINUE_SEARCH`—Windows ignores the exception handler and searches for an exception handler in the enclosing block, continuing until it finds a handler.
3. `EXCEPTION_CONTINUE_EXECUTION`—Windows immediately returns control to the point at which the exception occurred. It is not possible to continue after some exceptions, and inadvisable in most other cases, and another exception is generated immediately if the program attempts to do so.

Here is a simple example using an exception handler to delete a temporary file if an exception occurs in the loop body. Notice you can apply the `__try` clause to any block, including the block associated with a `while`, `if`, or other flow control statement. In this example, if there is any exception, the exception handler closes the file handle and deletes the temporary file. The loop iteration continues.

The exception handler executes unconditionally. In many realistic situations, the exception code is tested first to determine if the exception handler should execute; the next sections show how to test the exception code.

```
hFile = INVALID_HANDLE_VALUE;
while (...) __try {
   GetTempFileName(tempFile, ...);
   hFile = CreateFile(tempFile, ..., OPEN_ALWAYS, ...);
   SetFilePointerEx(hFile, 0, NULL, FILE_END);
   ...
   WriteFile(hFile, ...);
   i = *p; /* An addressing exception could occur. */
   ...
   CloseHandle(hFile);
   hFile = INVALID_HANDLE_VALUE;
}
__except (EXCEPTION_EXECUTE_HANDLER) {
   /* Print error information; this is probably a bug */
   if (INVALID_HANDLE_VALUE != hFile) CloseHandle(hFile);
```

```
      hFile = INVALID_HANDLE_VALUE;
      DeleteFile(tempFile);
      /* The loop will now execute the next iteration .*/
   }
   /* Control passes here after normal loop termination.
      The file handle is always closed and the temp file
      will not exist if there was an exception. */
```

The logic of this code fragment is as follows.

- Each loop iteration writes data to a temporary file associated with the iteration. An enhancement would append an identifier to the temporary file name.
- If an exception occurs in any loop iteration, all data accumulated in the temporary file is deleted, and the next iteration, if any, starts to accumulate data in a new temporary file with a new name. You need to create a new name so that another process does not get the temporary name after the deletion.
- The example shows just one location where an exception could occur, although the exception could occur anywhere within the loop body.
- The file handle is assured of being closed when exiting the loop or starting a new loop iteration.
- If an exception occurs, there is almost certainly a program bug. Program 4–4 shows how to analyze an address exception. Nonetheless, this code fragment allows the loop to continue, although it might be better to consider this a fatal error and terminate the program.

## Exception Codes

The `__except` block or the filter expression can determine the exact exception using this function:

```
DWORD GetExceptionCode (VOID)
```

You must get the exception code immediately after an exception. Therefore, the filter function itself cannot call `GetExceptionCode` (the compiler enforces this restriction). A common usage is to invoke it in the filter expression, as in the following example, where the exception code is the argument to a user-supplied filter function.

```
__except (MyFilter(GetExceptionCode())) {
}
```

In this situation, the filter function determines and returns the filter expression value, which must be one of the three values enumerated earlier. The function can use the exception code to determine the function value; for example, the filter may decide to pass floating-point exceptions to an outer handler (by returning `EXCEPTION_CONTINUE_SEARCH`) and to handle a memory access violation in the current handler (by returning `EXCEPTION_EXECUTE_HANDLER`).

`GetExceptionCode` can return a large number of possible exception code values, and the codes are in several categories.

- Program violations such as the following:
  - `EXCEPTION_ACCESS_VIOLATION`—An attempt to read, write, or execute a virtual address for which the process does not have access.
  - `EXCEPTION_DATATYPE_MISALIGNMENT`—Many processors insist, for example, that `DWORD`s be aligned on 4-byte boundaries.
  - `EXCEPTION_NONCONTINUABLE_EXECUTION`—The filter expression was `EXCEPTION_CONTINUE_EXECUTION`, but it is not possible to continue after the exception that occurred.
- Exceptions raised by the memory allocation functions—`HeapAlloc` and `HeapCreate`—if they use the `HEAP_GENERATE_EXCEPTIONS` flag (see Chapter 5). The value will be either `STATUS_NO_MEMORY` or `EXCEPTION_ACCESS_VIOLATION`.
- A user-defined exception code generated by the `RaiseException` function; see the User-Generated Exceptions subsection.
- A large variety of arithmetic (especially floating-point) codes such as `EXCEPTION_INT_DIVIDE_BY_ZERO` and `EXCEPTION_FLT_OVERFLOW`.
- Exceptions used by debuggers, such as `EXCEPTION_BREAKPOINT` and `EXCEPTION_SINGLE_STEP`.

`GetExceptionInformation` is an alternative function, callable only from within the filter expression, which returns additional information, some of which is processor-specific. Program 4–3 uses `GetExceptionInformation`.

```
LPEXCEPTION_POINTERS GetExceptionInformation (VOID)
```

The EXCEPTION_POINTERS structure contains both processor-specific and processor-independent information organized into two other structures, an exception record and a context record.

```
typedef struct _EXCEPTION_POINTERS {
   PEXCEPTION_RECORD ExceptionRecord;
   PCONTEXT ContextRecord;
} EXCEPTION_POINTERS;
```

EXCEPTION_RECORD contains a member for the ExceptionCode with the same set of values as returned by GetExceptionCode. The ExceptionFlags member of the EXCEPTION_RECORD is either 0 or EXCEPTION_NONCONTINUABLE, which allows the filter function to determine that it should not attempt to continue execution. Other data members include a virtual memory address, ExceptionAddress, and a parameter array, ExceptionInformation.

In the case of EXCEPTION_ACCESS_VIOLATION or EXCEPTION_IN_PAGE-_VIOLATION, the first element indicates whether the violation was a memory write (1), read (0), or execute (8). The second element is the virtual memory address. The third array element specifies the NSTATUS code that caused the exception.

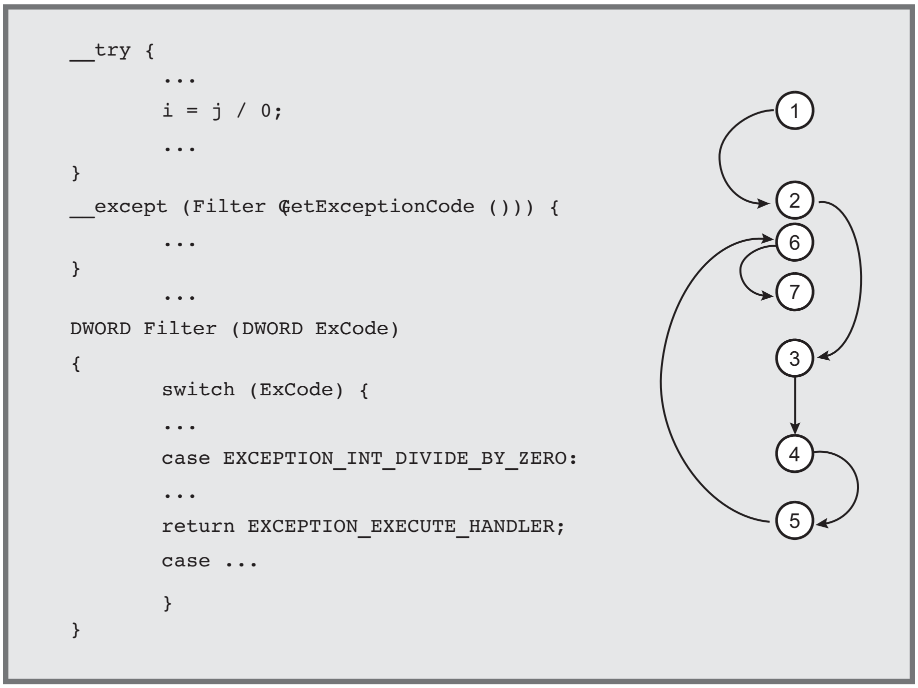
The execute error (code 8) is a Data Execution Prevention (DEP) error, which indicates an attempt to execute data that is not intended to be code, such as data on the heap. This feature is supported as of XP SP2; see MSDN for more information.

ContextRecord, the second EXCEPTION_POINTERS member, contains processor-specific information, including the address where the exception occurred. There are different structures for each type of processor, and the structure can be found in <winnt.h>.

## Summary: Exception Handling Sequence

Figure 4–2 shows the sequence of events that takes place when an exception occurs. The code is on the left side, and the circled numbers on the right show the steps carried out by the language run-time support. The steps are as follows.

1. The exception occurs, in this case a division by zero.
2. Control transfers to the exception handler, where the filter expression is evaluated. GetExceptionCode is called first, and its return value is the argument to the function Filter.



```
__try {
        ...
        i = j / 0;
        ...
}
__except (Filter (GetExceptionCode ())) {
        ...
}
        ...
DWORD Filter (DWORD ExCode)
{
        switch (ExCode) {
        ...
        case EXCEPTION_INT_DIVIDE_BY_ZERO:
        ...
        return EXCEPTION_EXECUTE_HANDLER;
        case ...

        }
}
```

**Figure 4–2** Exception Handling Sequence

3. The filter function bases its actions on the exception code value.
4. The exception code is `EXCEPTION_INT_DIVIDE_BY_ZERO` in this case.
5. The filter function determines that the exception handler should be executed, so the return value is `EXCEPTION_EXECUTE_HANDLER`.
6. The exception handler, which is the code associated with the `__except` clause, executes.
7. Control passes out of the try-except block.

## Floating-Point Exceptions

Readers not interested in floating-point arithmetic may wish to skip this section.

There are seven distinct floating-point exception codes. These exceptions are disabled initially and will not occur without first setting the processor-independent floating-point mask with the `_controlfp` function. Alternatively, enable floating-point exceptions with the `/fp:except` compiler flag (you can also specify this from Visual Studio).

There are specific exceptions for underflow, overflow, division by zero, inexact results, and so on, as shown in a later code fragment. Turn the mask bit *off* to enable the particular exception.

```
DWORD _controlfp (DWORD new, DWORD mask)
```

The new value of the floating-point mask is determined by its current value (`current_mask`) and the two arguments as follows:

```
(current_mask & ~mask) | (new & mask)
```

The function sets the bits specified by `new` that are enabled by `mask`. All bits *not* in `mask` are unaltered. The floating-point mask also controls processor precision, rounding, and infinity values, which should not be modified (these topics are out-of-scope).

The return value is the updated setting. Thus, if both argument values are `0`, the value is unchanged, and the return value is the current `mask` setting, which can be used later to restore the mask. On the other hand, if `mask` is `0xFFFFFFFF`, then the register is set to `new`, so that, for example, an old value can be restored.

Normally, to enable the floating-point exceptions, use the floating-point exception `mask` value, `MCW_EM`, as shown in the following example. Notice that when a floating-point exception is processed, the exception must be cleared using the `_clearfp` function.

```
#include <float.h>
DWORD fpOld, fpNew; /* Old and new mask values. */
   ...
fpOld = _controlfp(0, 0); /* Saved old mask. */
/* Specify six exceptions to be enabled. */
   fpNew = fpOld & ~(EM_OVERFLOW | EM_UNDERFLOW
   | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL | EM_INVALID);
/* Set new control mask. MCW_EM combines the six
   exceptions in the previous statement. */
_controlfp(fpNew, MCW_EM);
while (...) __try { /* Perform FP calculations. */
   ... /* An FP exception could occur here. */
}
```

```
__except (EXCEPTION_EXECUTE_HANDLER) {
   ... /* Analyze and log the FP exception. */
   _clearfp(); /* Clear the exception. */
   _controlfp(fpOld, 0xFFFFFFFF); /* Restore mask. */
   /* Don't continue execution. */
}
```

This example enables all possible floating-point exceptions except for the floating-point stack overflow, `EXCEPTION_FLT_STACK_CHECK`. Alternatively, enable specific exceptions by using only selected exception masks, such as `EM_OVERFLOW`. Program 4–3 uses similar code in the context of a larger example.

## Errors and Exceptions

An error can be thought of as a situation that could occur occasionally and synchronously at known locations. System call errors, for example, should be detected and reported immediately by logic in the code. Thus, programmers normally include an explicit test to see, for instance, whether a file read operation has failed. Chapter 2's `ReportError` function can diagnose and respond to errors.

An exception, on the other hand, could occur nearly anywhere, and it is not possible or practical to test for an exception. Division by zero and memory access violations are examples. Exceptions are asynchronous.

Nonetheless, the distinction is sometimes blurred. Windows will, optionally, generate exceptions during memory allocation using the `HeapAlloc` and `HeapCreate` functions if memory is insufficient (see Chapter 5). Programs can also raise their own exceptions with programmer-defined exception codes using the `RaiseException` function, as described next.

Exception handlers provide a convenient mechanism for exiting from inner blocks or functions under program control without resorting to a `goto`, `longjmp`, or some other control logic to transfer control; Program 4–2 illustrates this. This capability is particularly important if the block has accessed resources, such as open files, memory, or synchronization objects, because the handler can release them.

User-generated exceptions provide one of the few cases where it is possible or desirable to continue execution at the exception point rather than terminate the program, thread, or the block or function. However, use caution when continuing execution from the exception point.

Finally, a program can restore system state, such as the floating-point mask, on exiting from a block. Some examples use handlers in this way.

## User-Generated Exceptions

You can raise an exception at any point during program execution using the RaiseException function. In this way, your program can detect an error and treat it as an exception.

```
VOID RaiseException (
   DWORD dwExceptionCode,
   DWORD dwExceptionFlags,
   DWORD nNumberOfArguments,
   CONST DWORD *lpArguments)
```

### *Parameters*

dwExceptionCode is the user-defined code. Do not use bit 28, which is reserved and Windows clears. The error code is encoded in bits 27–0 (that is, all except the most significant hex digit). Set bit 29 to indicate a "customer" (not Microsoft) exception. Bits 31–30 encode the severity as follows, where the resulting lead exception code hex digit is shown with bit 29 set.

- 0—Success (lead exception code hex digit is 2).
- 1—Informational (lead exception code hex digit is 6).
- 2—Warning (lead exception code hex digit is A).
- 3—Error (lead exception code hex digit is E).

dwExceptionFlags is normally 0, but setting the value to EXCEPTION_NONCONTINUABLE indicates that the filter expression should not generate EXCEPTION_CONTINUE_EXECUTION; doing so will cause an immediate EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

lpArguments, if not NULL, points to an array of size nNumberOfArguments (the third parameter) containing values to be passed to the filter expression. The values can be interpreted as pointers and are 32 (Win32) or 64 (Win64) bits long, EXCEPTION_MAXIMUM_PARAMETERS (15) is the maximum number of parameters that can be passed. Use GetExceptionInformation to access this structure.

Note that it is not possible to raise an exception in another process or even another thread in your process. Under very limited circumstances, however, console control handlers, described at the end of this chapter and in Chapter 6, can raise exceptions in a different process.

## Example: Treating Errors as Exceptions

Previous examples use `ReportError` to process system call and other errors. The function terminates the process when the programmer indicates that the error is fatal. This approach, however, prevents an orderly shutdown, and it also prevents program continuation after recovering from an error. For example, the program may have created temporary files that should be deleted, or the program may simply proceed to do other work after abandoning the failed task. `ReportError` has other limitations, including the following.

- A fatal error shuts down the entire process when only a single thread (Chapter 7) should terminate.
- You may wish to continue program execution rather than terminate the process.
- Synchronization resources (Chapter 8), such as events or semaphores, will not be released in many circumstances.

Open handles will be closed by a terminating process, but not by a terminating thread. It is necessary to address this and other deficiencies.

The solution is to write a new function that invokes `ReportError` (Chapter 2) with a nonfatal code in order to generate the error message. Next, on a fatal error, it will raise an exception. Windows will use an exception handler from the calling try block, so the exception may not actually be fatal if the handler allows the program to recover and resume. Essentially, `ReportException` augments normal defensive programming techniques, previously limited to `ReportError`. Once a problem is detected, the exception handler allows the program to recover and continue after the error. Program 4–2 illustrates this capability.

Program 4–1 shows the function. It is in the same source module as `ReportError`, so the definitions and include files are omitted.

**Program 4–1** `ReportException`: Exception Reporting Function

```
/* ReportError extension to generate a nonfatal user-exception code. */

VOID ReportException(LPCTSTR userMessage, DWORD exceptionCode)
{
   ReportError(userMessage, 0, TRUE);
   if (exceptionCode != 0) /* If fatal, raise an exception. */
      RaiseException(
         (0x0FFFFFFF & exceptionCode) | 0xE0000000, 0, 0, NULL);
   return;
}
```

`ReportException` is used in Program 4–2 and elsewhere.

---

The UNIX signal model is significantly different from SEH. Signals can be missed or ignored, and the flow is different. Nonetheless, there are points of comparison.

UNIX signal handling is largely supported through the C library, which is also available in a limited implementation under Windows. In many cases, Windows programs can use console control handlers, which are described near the end of this chapter, in place of signals.

Some signals correspond to Windows exceptions.

Here is the limited signal-to-exception correspondence:

- `SIGILL`—`EXCEPTION_PRIV_INSTRUCTION` or `EXCEPTION_ILLEGAL_INSTRUCTION`
- `SIGSEGV`—`EXCEPTION_ACCESS_VIOLATION`
- `SIGFPE`—Seven distinct floating-point exceptions, such as `EXCEPTION_FLT_DIVIDE_BY_ZERO`
- `SIGUSR1` and `SIGUSR2`—User-defined exceptions

The C library `raise` function corresponds to `RaiseException`.

Windows will not generate `SIGILL`, `SIGSEGV`, or `SIGTERM`, although `raise` can generate one of them. Windows does not support `SIGINT`.

The UNIX `kill` function (`kill` is not in the Standard C library), which can send a signal to another process, is comparable to the Windows function `GenerateConsoleCtrlEvent` (Chapter 6). In the limited case of `SIGKILL`, there is no corresponding exception, but Windows has `TerminateProcess` and `TerminateThread`, allowing one process (or thread) to "kill" another, although these functions should be used with care (see Chapters 6 and 7).

## Termination Handlers

A termination handler serves much the same purpose as an exception handler, but it is executed when a thread leaves a block as a result of normal program flow as well as when an exception occurs. On the other hand, a termination handler cannot diagnose an exception.

Construct a termination handler using the `__finally` keyword in a try-finally statement. The structure is the same as for a try-except statement, but there is no filter expression. Termination handlers, like exception handlers, are a convenient way to close handles, release resources, restore masks, and otherwise restore the process to a known state when leaving a block. For example, a program may execute `return` statements in the middle of a block, and the termination handler can perform the cleanup work. In this way, there is no need to include the cleanup code in the code block itself, nor is there a need for `goto` or other control flow statements to reach the cleanup code.

Here is the try-finally form, and Program 4–2 illustrates the usage.

```
__try {
   /* Code block. */
}
__finally {
   /* Termination handler (finally block). */
}
```

## Leaving the Try Block

The termination handler is executed whenever the control flow leaves the try block for any of the following reasons:

- Reaching the end of the try block and "falling through" to the termination handler
- Execution of one of the following statements in such a way as to leave the block:

  `return`
  `break`
  `goto`[1]
  `longjmp`
  `continue`
  `__leave`[2]
- An exception

## Abnormal Termination

Termination for any reason other than reaching the end of the try block and falling through or performing a `__leave` statement is considered an abnormal termi-

[1] It may be a matter of taste, either individual or organizational, but many programmers never use the `goto` statement and try to avoid `break`, except with the `switch` statement and sometimes in loops, and with `continue`. Reasonable people continue to differ on this subject. The termination and exception handlers can perform many of the tasks that you might want to perform with a `goto` to a labeled statement.

[2] This statement is specific to the Microsoft C compiler and is an efficient way to leave a try-finally block without an abnormal termination.

nation. The effect of `__leave` is to transfer to the end of the `__try` block and fall through. Within the termination handler, use this function to determine how the try block terminated.

```
BOOL AbnormalTermination (VOID)
```

The return value will be `TRUE` for an abnormal termination or `FALSE` for a normal termination.

*Note:* The termination would be abnormal even if, for example, a `return` statement were the last statement in the try block.

## Executing and Leaving the Termination Handler

The termination handler, or `__finally` block, is executed in the context of the block or function that it monitors. Control can pass from the end of the termination handler to the next statement. Alternatively, the termination handler can execute a flow control statement (`return`, `break`, `continue`, `goto`, `longjmp`, or `__leave`). Leaving the handler because of an exception is another possibility.

## Combining Finally and Except Blocks

A single try block must have a single finally or except block; it cannot have both, even though it might be convenient. Therefore, the following code would cause a compile error.

```
__try {
   /* Block of monitored code. */
}
__except (filter_expression) {
   /* Except block. */
}
__finally {
   /* Do not do this! It will not compile. */
}
```

It is possible, however, to embed one block within another, a technique that is frequently useful. The following code is valid and ensures that the temporary file is deleted if the loop exits under program control or because of an exception. This technique is also useful to ensure that file locks are released. There is also an inner try-except block with some floating-point processing.

```
__try { /* Outer try-except block. */
   while (...) __try { /* Inner try-finally block. */
      hFile = CreateFile(tempFile, ...);
      if (...) __try { /* Inner try-except block. */
         /* Enable FP exceptions. Perform computations. */
         ...
      }
      __except (fp-filter-expression) {
         ... /* Process FP exception. */ _clearfp();
      }
      ... /* Non-FP processing. /*
   }
   __finally { /* End of while loop. */
   /* Executed on EVERY loop iteration. */
      CloseHandle(hFile); DeleteFile(tempFile);
   }
}
__except (filter-expression) {
   /* Exception handler. */
}
```

## Global and Local Unwinds

Exceptions and abnormal terminations will cause a *global stack unwind* to search for a handler, as in Figure 4–1. For example, suppose an exception occurs in the monitored block of the example at the end of the preceding section before the floating-point exceptions are enabled. The termination handler will be executed first, followed by the exception handler at the end. There might be numerous termination handlers on the stack before the exception handler is located.

Recall that the stack structure is dynamic, as shown in Figure 4–1, and that it contains, among other things, the exception and termination handlers. The contents at any time depend on:

- The *static* structure of the program's blocks
- The *dynamic* structure of the program as reflected in the sequence of open function calls

### Termination Handlers: Process and Thread Termination

Termination handlers do not execute if a process or thread terminates, whether the process or thread terminates itself by using `ExitProcess` or `ExitThread`, or whether the termination is external, caused by a call to `TerminateProcess` or `TerminateThread` from elsewhere. Therefore, a process or thread should not execute one of these functions inside a try-except or try-finally block.

Notice also that the C library `exit` function or a return from a `main` function will exit the process.

### SEH and C++ Exception Handling

C++ exception handling uses the keywords `catch` and `throw` and is implemented using SEH. Nonetheless, C++ exception handling and SEH are distinct. They should be mixed with care, or not at all, because the user-written and C++-generated exception handlers may interfere with expected operation. For example, an `__except` handler may be on the stack and catch a C++ exception so that the C++ handler will never receive the exception. The converse is also possible, with a C++ handler catching, for example, an SEH exception generated with `RaiseException`. The Microsoft documentation recommends that Windows exception handlers not be used in C++ programs at all but instead that C++ exception handling be used exclusively.

Normally, a Windows exception or termination handler will not call destructors to destroy C++ object instances. However, the `/EHa` compiler flag (settable from Visual Studio) allows C++ exception handling to include asynchronous exceptions and "unwind" (destroy) C++ objects.

## Example: Using Termination Handlers to Improve Program Quality

Termination and exception handlers allow you to make your program more robust by both simplifying recovery from errors and exceptions and helping to ensure that resources and file locks are freed at critical junctures.

Program 4–2, `toupper`, illustrates these points, using ideas from the preceding code fragments. `toupper` processes multiple files, as specified on the command line, rewriting them so that all letters are in uppercase. Converted files are named by prefixing `UC_` to the original file name, and the program "specification" states that an existing file should not be overridden. File conversion is performed in memory, so there is a large buffer (sufficient for the entire file) allocated for each file. There are multiple possible failure points for each processed file, but the program must defend against all such errors and then recover and attempt to process all the remaining files named on the command line. Program 4–2 achieves this without resorting to the elaborate control flow methods that would be necessary without SEH.

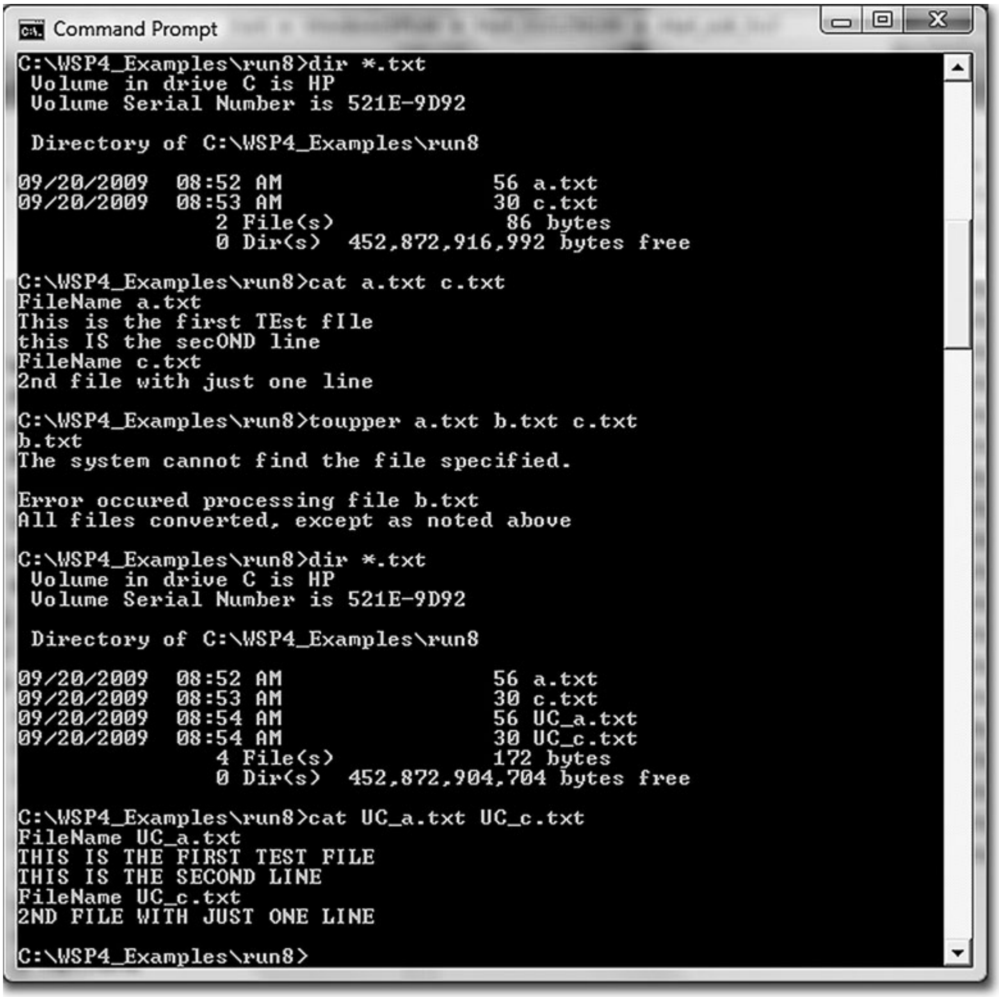
Note that this program depends on file sizes, so it will not work on objects for which `GetFileSizeEx` fails, such as a named pipe (Chapter 11). Furthermore, it fails for large text files longer than 4GB.

The code in the *Examples* file has more extensive comments.

**Program 4–2** `toupper`: File Processing with Error and Exception Recovery

```
/* Chapter 4. toupper command. */
/* Convert one or more files, changing all letters to uppercase.
   The output file will be the same name as the input file, except
   a UC_ prefix will be attached to the file name. */

#include "Everything.h"

int _tmain(DWORD argc, LPTSTR argv[])
{
   HANDLE hIn = INVALID_HANDLE_VALUE, hOut = INVALID_HANDLE_VALUE;
   DWORD nXfer, iFile, j;
   CHAR outFileName[256] = "", *pBuffer = NULL;
   OVERLAPPED ov = { 0, 0, 0, 0, NULL};
   LARGE_INTEGER fSize;

   /* Process all files on the command line. */
   for (iFile = 1; iFile < argc; iFile++) __try { /* Exceptn block */
      /* All file handles are invalid, pBuffer == NULL, and
         outFileName is empty. This is assured by the handlers */
      if (_tcslen(argv[iFile]) > 250)
         ReportException(_T("The file name is too long."), 1);
      _stprintf(outFileName, "UC_%s", argv[iFile]);

      __try { /* Inner try-finally block */
         hIn  = CreateFile(argv[iFile], GENERIC_READ,
            0, NULL, OPEN_EXISTING, 0, NULL);
         if (hIn == INVALID_HANDLE_VALUE)
            ReportException(argv[iFile], 1);

         if (!GetFileSizeEx(hIn, &fSize) || fSize.HighPart > 0)
            ReportException(_T("This file is too large."), 1);

         hOut = CreateFile(outFileName,
            GENERIC_READ | GENERIC_WRITE,
            0, NULL, CREATE_NEW, 0, NULL);
         if (hOut == INVALID_HANDLE_VALUE)
            ReportException(outFileName, 1);

         /* Allocate memory for the file contents */
         pBuffer = malloc(fSize.LowPart);
         if (pBuffer == NULL)
            ReportException(_T("Memory allocation error"), 1);

         /* Read the data, convert it, and write to the output file */
         /* Free all resources on completion; process next file */

         if (!ReadFile(hIn, pBuffer, fSize.LowPart, &nXfer, NULL)
                  || (nXfer != fSize.LowPart))
               ReportException(_T("ReadFile error"), 1);

         for (j = 0; j < fSize.LowPart; j++) /* Convert data */
            if (isalpha(pBuffer[j])) pBuffer[j] =
               toupper(pBuffer[j]);

         if (!WriteFile(hOut, pBuffer, fSize.LowPart, &nXfer, NULL)
                  || (nXfer != fSize.LowPart))
               ReportException(_T("WriteFile error"), 1);

      } __finally { /* File handles are always closed */
         /* memory freed, and handles and pointer reinitialized. */
         if (pBuffer != NULL) free(pBuffer); pBuffer = NULL;
         if (hIn  != INVALID_HANDLE_VALUE) {
            CloseHandle(hIn);
            hIn  = INVALID_HANDLE_VALUE;
         }
         if (hOut != INVALID_HANDLE_VALUE) {
            CloseHandle(hOut);
            hOut = INVALID_HANDLE_VALUE;
         }
         _tcscpy(outFileName, _T(""));
      }
   } /* End of main file processing loop and try block. */
   /* This exception handler applies to the loop body */

   __except (EXCEPTION_EXECUTE_HANDLER) {
      _tprintf(_T("Error processing file %s\n"), argv[iFile]);
      DeleteFile(outFileName);
   }
   _tprintf(_T("All files converted, except as noted above\n"));
   return 0;
}
```

Run 4–2 shows `toupper` operation. Originally, there are two text files, `a.txt` and `b.txt`. The `cat` program (Program 2–2) displays the contents of these two files; you could also use the Windows `type` command. `toupper` converts these two files, continuing after failing to find `b.txt`. Finally, cat displays the two converted files, `UC_a.txt` and `UC_c.txt`.



```
C:\WSP4_Examples\run8>dir *.txt
 Volume in drive C is HP
 Volume Serial Number is 521E-9D92

 Directory of C:\WSP4_Examples\run8

09/20/2009  08:52 AM                56 a.txt
09/20/2009  08:53 AM                30 c.txt
               2 File(s)             86 bytes
               0 Dir(s)  452,872,916,992 bytes free

C:\WSP4_Examples\run8>cat a.txt c.txt
FileName a.txt
This is the first TEst fIle
this IS the secOND line
FileName c.txt
2nd file with just one line

C:\WSP4_Examples\run8>toupper a.txt b.txt c.txt
b.txt
The system cannot find the file specified.

Error occured processing file b.txt
All files converted, except as noted above

C:\WSP4_Examples\run8>dir *.txt
 Volume in drive C is HP
 Volume Serial Number is 521E-9D92

 Directory of C:\WSP4_Examples\run8

09/20/2009  08:52 AM                56 a.txt
09/20/2009  08:53 AM                30 c.txt
09/20/2009  08:54 AM                56 UC_a.txt
09/20/2009  08:54 AM                30 UC_c.txt
               4 File(s)            172 bytes
               0 Dir(s)  452,872,904,704 bytes free

C:\WSP4_Examples\run8>cat UC_a.txt UC_c.txt
FileName UC_a.txt
THIS IS THE FIRST TEST FILE
THIS IS THE SECOND LINE
FileName UC_c.txt
2ND FILE WITH JUST ONE LINE

C:\WSP4_Examples\run8>
```

**Run 4–2** `toupper`**:** Converting Text Files to Uppercase

## Example: Using a Filter Function

Program 4–3 is a skeleton program that illustrates exception and termination handling with a filter function. This example prompts the user to specify the exception type and then proceeds to generate an exception. The filter function disposes of the different exception types in various ways; the selections here are arbitrary and intended simply to illustrate the possibilities. In particular, the program diagnoses memory access violations, giving the virtual address of the reference.

The `__finally` block restores the state of the floating-point mask. Restoring state, as done here, is not important when the process is about to terminate, but it is important later when a thread is terminated. In general, a process should still restore system resources by, for example, deleting temporary files and releasing synchronization resources (Chapter 8) and file locks (Chapters 3 and 6). Program 4–4 shows the filter function.

This example does not illustrate memory allocation exceptions; they will be used starting in Chapter 5.

Run 4–4, after the filter function (Program 4–4) shows the program operation.

**Program 4–3** `Exception`: Processing Exceptions and Termination

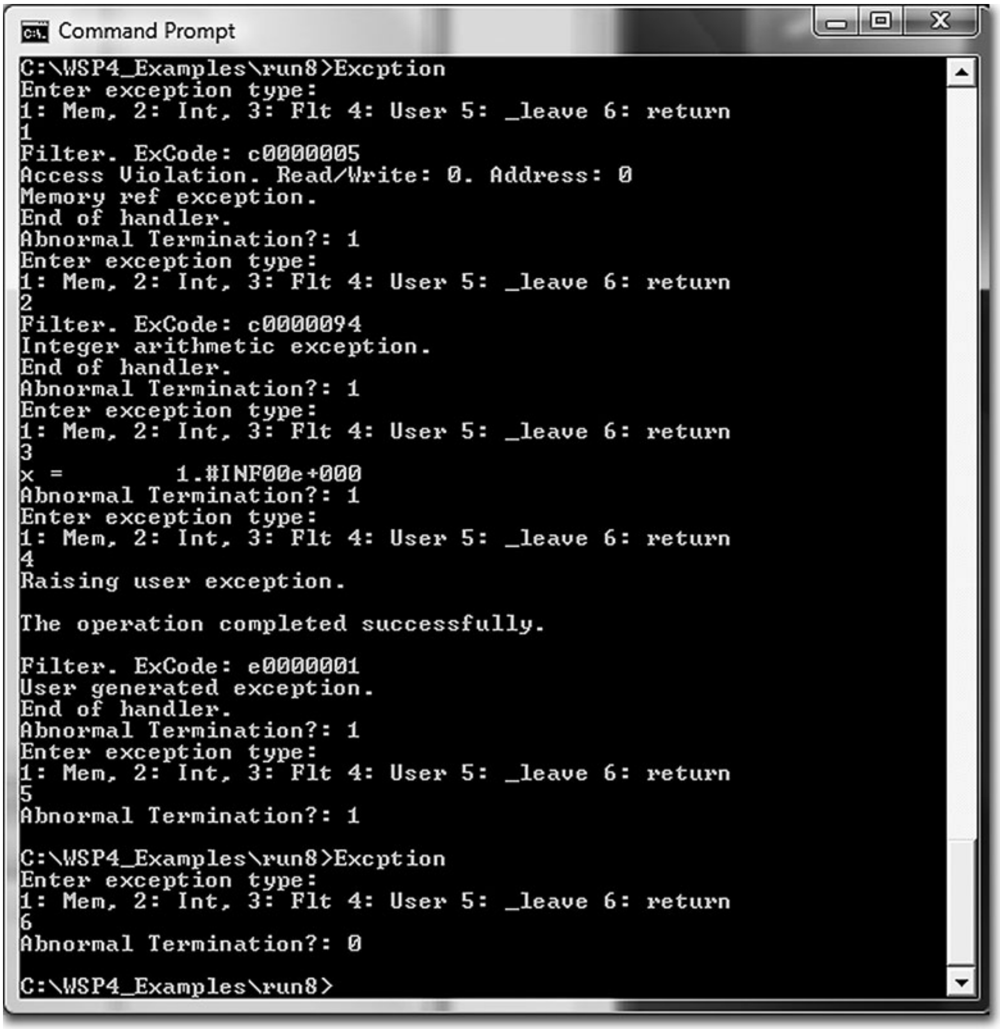
```
#include "Everything.h"
#include <float.h>

DWORD Filter(LPEXCEPTION_POINTERS, LPDWORD);
double x = 1.0, y = 0.0;

int _tmain(int argc, LPTSTR argv[])
{
   DWORD eCategory, i = 0, ix, iy = 0;
   LPDWORD pNull = NULL;
   BOOL done = FALSE;
   DWORD fpOld, fpNew;
   fpOld = _controlfp(0, 0); /* Save old control mask. */
                   /* Enable floating-point exceptions. */
   fpNew = fpOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT
         | EM_ZERODIVIDE | EM_DENORMAL | EM_INVALID);
   _controlfp(fpNew, MCW_EM);

   while (!done) __try { /* Try-finally. */
      _tprintf(_T("Enter exception type: "));
      _tprintf(_T
            (" 1: Mem, 2: Int, 3: Flt 4: User 5: __leave "));
      _tscanf(_T("%d"), &i);
      __try { /* Try-except block. */
         switch (i) {
         case 1: /* Memory reference. */
            ix = *pNull; *pNull = 5; break;
         case 2: /* Integer arithmetic. */
            ix = ix / iy; break;
         case 3: /* Floating-point exception. */
            x = x / y;
            _tprintf(_T("x = %20e\n"), x); break;
         case 4: /* User-generated exception. */
            ReportException(_T("User exception"), 1); break;
         case 5: /* Use the _leave statement to terminate. */
```

```
                __leave;
            default: done = TRUE;
            }
        } /* End of inner __try. */

        __except (Filter(GetExceptionInformation(), &eCategory))
        {
            switch (eCategory) {
                case 0:
                    _tprintf(_T("Unknown Exception\n")); break;
                case 1:
                    _tprintf(_T("Memory Ref Exception\n")); continue;
                case 2:
                    _tprintf(_T("Integer Exception\n")); break;
                case 3:
                    _tprintf(_T("Floating-Point Exception\n"));
                    _clearfp(); break;
                case 10:
                    _tprintf(_T("User Exception\n")); break;
                default:
                    _tprintf( _T("Unknown Exception\n")); break;
            } /* End of switch statement. */

            _tprintf(_T("End of handler\n"));
        } /* End of try-except block. */
    } /* End of While loop -- the termination handler is below. */

    __finally { /* This is part of the while loop. */
        _tprintf(_T("Abnormal Termination?: %d\n"),
                AbnormalTermination());
    }
    _controlfp(fpOld, 0xFFFFFFFF); /* Restore old FP mask.*/
    return 0;
}
```

Program 4–4 shows the filter function used in Program 4–3. This function simply checks and categorizes the various possible exception code values. The code in the *Examples* file checks every possible value; here the function tests only for a few that are relevant to the test program.

**Program 4–4** `Filter`: Exception Filtering

```
static DWORD Filter(LPEXCEPTION_POINTERS pExP, LPDWORD eCategory)
/* Categorize the exception and decide action. */
{
   DWORD exCode, readWrite, virtAddr;
   exCode = pExP->ExceptionRecord->ExceptionCode;
   _tprintf(_T("Filter. exCode: %x\n"), exCode);
   if ((0x20000000 & exCode) != 0) { /* User exception. */
      *eCategory = 10;
      return EXCEPTION_EXECUTE_HANDLER;
   }

   switch (exCode) {
      case EXCEPTION_ACCESS_VIOLATION:
         readWrite = /* Was it a read, write, execute? */
               pExP->ExceptionRecord->ExceptionInformation[0];
         virtAddr = /* Virtual address of the violation. */
               pExP->ExceptionRecord->ExceptionInformation[1];
         _tprintf(
         _T("Access Violation. Read/Write/Exec: %d. Address: %x\n"),
               readWrite, virtAddr);
         *eCategory = 1;
         return EXCEPTION_EXECUTE_HANDLER;
      case EXCEPTION_INT_DIVIDE_BY_ZERO:
      case EXCEPTION_INT_OVERFLOW:
         *eCategory = 2;
         return EXCEPTION_EXECUTE_HANDLER;
      case EXCEPTION_FLT_DIVIDE_BY_ZERO:
      case EXCEPTION_FLT_OVERFLOW:
         _tprintf(_T("Flt Exception - large result.\n"));
         *eCategory = 3;
         _clearfp();
         return EXCEPTION_EXECUTE_HANDLER;

      default:
         *eCategory = 0;
         return EXCEPTION_CONTINUE_SEARCH;
   }
}
```



```
C:\WSP4_Examples\run8>Excption
Enter exception type:
1: Mem, 2: Int, 3: Flt 4: User 5: _leave 6: return
1
Filter. ExCode: c0000005
Access Violation. Read/Write: 0. Address: 0
Memory ref exception.
End of handler.
Abnormal Termination?: 1
Enter exception type:
1: Mem, 2: Int, 3: Flt 4: User 5: _leave 6: return
2
Filter. ExCode: c0000094
Integer arithmetic exception.
End of handler.
Abnormal Termination?: 1
Enter exception type:
1: Mem, 2: Int, 3: Flt 4: User 5: _leave 6: return
3
x =        1.#INF00e+000
Abnormal Termination?: 1
Enter exception type:
1: Mem, 2: Int, 3: Flt 4: User 5: _leave 6: return
4
Raising user exception.

The operation completed successfully.

Filter. ExCode: e0000001
User generated exception.
End of handler.
Abnormal Termination?: 1
Enter exception type:
1: Mem, 2: Int, 3: Flt 4: User 5: _leave 6: return
5
Abnormal Termination?: 1

C:\WSP4_Examples\run8>Excption
Enter exception type:
1: Mem, 2: Int, 3: Flt 4: User 5: _leave 6: return
6
Abnormal Termination?: 0

C:\WSP4_Examples\run8>
```

**Run 4–4** `Filter`: Exception Filtering

## Console Control Handlers

Exception handlers can respond to a variety of asynchronous events, but they do not detect situations such as the user logging off or entering a `Ctrl-C` from the keyboard to stop a program. Use console control handlers to detect such events.

The function `SetConsoleCtrlHandler` allows one or more specified functions to be executed on receipt of a `Ctrl-C`, `Ctrl-break`, or one of three other console-related signals. The `GenerateConsoleCtrlEvent` function, described in

Chapter 6, also generates these signals, and the signals can be sent to other processes that are sharing the same console. The handlers are user-specified Boolean functions that take a DWORD argument identifying the signal.

Multiple handlers can be associated with a signal, and handlers can be removed as well as added. Here is the function to add or delete a handler.

```
BOOL SetConsoleCtrlHandler (
   PHANDLER_ROUTINE HandlerRoutine,
   BOOL Add)
```

The handler routine is added if the Add flag is TRUE; otherwise, it is deleted from the list of console control routines. Notice that the signal is not specified. The handler must test to see which signal was received.

The handler routine returns a Boolean value and takes a single DWORD parameter that identifies the signal. The *HandlerRoutine* in the definition is a placeholder; the programmer specifies the name.

Here are some other considerations when using console control handlers.

- If the *HandlerRoutine* parameter is NULL and Add is TRUE, Ctrl-C signals will be ignored.
- The ENABLE_PROCESSED_INPUT flag on SetConsoleMode (Chapter 2) will cause Ctrl-C to be treated as keyboard input rather than as a signal.
- The handler routine actually executes as an *independent thread* (see Chapter 7) within the process. The normal program will continue to operate, as shown in the next example.
- Raising an exception in the handler *will not* cause an exception in the thread that was interrupted because exceptions apply to threads, not to an entire process. If you wish to communicate with the interrupted thread, use a variable, as in the next example, or a synchronization method (Chapter 8).

There is one other important distinction between exceptions and signals. A signal applies to the entire process, whereas an exception applies only to the thread executing the code where the exception occurs.

```
BOOL HandlerRoutine (DWORD dwCtrlType)
```

`dwCtrlType` identifies the signal (or *event*) and can take on one of the following five values.

1. `CTRL_C_EVENT` indicates that the `Ctrl-C` sequence was entered from the keyboard.
2. `CTRL_CLOSE_EVENT` indicates that the console window is being closed.
3. `CTRL_BREAK_EVENT` indicates the `Ctrl-break` signal.
4. `CTRL_LOGOFF_EVENT` indicates that the user is logging off.
5. `CTRL_SHUTDOWN_EVENT` indicates that Windows is shutting down.

The signal handler can perform cleanup operations just as an exception or termination handler would. The signal handler can return `TRUE` to indicate that the function handled the signal. If the signal handler returns `FALSE`, the next handler function in the list is executed. The signal handlers are executed in the reverse order from the way they were set, so that the most recently set handler is executed first and the system handler is executed last.

## Example: A Console Control Handler

Program 4–5 loops forever, calling the self-explanatory `Beep` function every 5 seconds. The user can terminate the program with a `Ctrl-C` or by closing the console. The handler routine will put out a message, wait 10 seconds, and, it would appear, return `TRUE`, terminating the program. The main program, however, detects the `exitFlag` flag and stops the process. This illustrates the concurrent operation of the handler routine; note that the timing of the signal determines the extent of the signal handler's output. Examples in later chapters also use console control handlers.

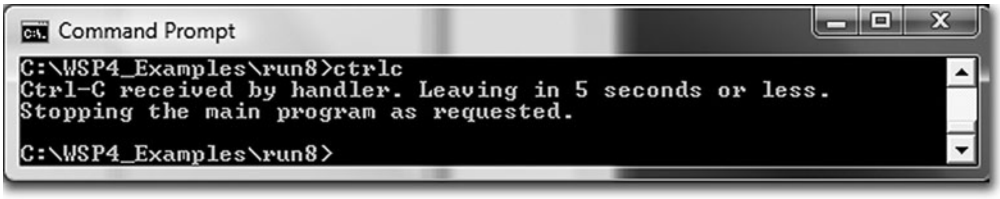
Note the use of `WINAPI`; this macro is for user functions passed as arguments to Windows functions to assure the proper calling conventions. It is defined in the Platform SDK header file `windef.h`.

**Program 4–5** `Ctrlc`: Signal Handling Program

```
/* Chapter 4. Ctrlc.c */
/* Catch console events. */

#include "Everything.h"

static BOOL WINAPI Handler(DWORD cntrlEvent);
static BOOL exitFlag = FALSE;
```

```
int _tmain(int argc, LPTSTR argv[])

/* Beep periodically until signaled to stop. */
{
   /* Add an event handler. */
   SetConsoleCtrlHandler(Handler, TRUE);

   while (!exitFlag) {
      Sleep(5000); /* Beep every 5 seconds. */
      Beep(1000 /* Frequency. */, 250 /* Duration. */);
   }
   _tprintf(_T("Stopping the program as requested.\n"));
   return 0;
}

BOOL WINAPI Handler(DWORD cntrlEvent)
{
   exitFlag = TRUE;

   switch (cntrlEvent) {
      /* Timing determines if you see the second handler message. */
      case CTRL_C_EVENT:
         _tprintf(_T("Ctrl-C. Leaving in <= 5 seconds.\n"));
         exitFlag = TRUE;
         Sleep(4000); /* Decrease to get a different effect */
         _tprintf(_T("Leaving handler in 1 second or less.\n"));
         return TRUE; /* TRUE indicates the signal was handled. */
      case CTRL_CLOSE_EVENT:
         _tprintf(_T("Close event. Leaving in <= 5 seconds.\n"));
         exitFlag = TRUE;
         Sleep(4000); /* Decrease to get a different effect */
         _tprintf(_T("Leaving handler in <= 1 second.\n"));
         return TRUE; /* Try returning FALSE. Any difference? */
      default:
         _tprintf(_T("Event: %d. Leaving in <= 5 seconds.\n"),
            cntrlEvent);
         exitFlag = TRUE;
         Sleep(4000); /* Decrease to get a different effect */
         _tprintf(_T("Leaving handler in <= 1 second.\n"));
         return TRUE; /* TRUE indicates the signal was handled. */
   }
}
```

There's very little to show with this program, as we can't show the sound effects. Nonetheless, Run 4–5 shows the command window where I typed Ctrl-C after about 11 seconds.



```
C:\WSP4_Examples\run8>ctrlc
Ctrl-C received by handler. Leaving in 5 seconds or less.
Stopping the main program as requested.

C:\WSP4_Examples\run8>
```

**Run 4–5** `Ctrlc`: Interrupting Program Execution from the Console

## Vectored Exception Handling

Exception handling functions can be directly associated with exceptions, just as console control handlers can be associated with console control events. When an exception occurs, the *vectored exception handlers* are called first, before the system unwinds the stack to look for structured exception handlers. No keywords, such as `__try` and `__catch`, are required.

Vectored exception handling (VEH) management is similar to console control handler management, although the details are different. Add, or "register," a handler using `AddVectoredExceptionHandler`.

```
PVOID WINAPI AddVectoredExceptionHandler (
   ULONG FirstHandler,
   PVECTORED_EXCEPTION_HANDLER VectoredHandler)
```

Handlers can be chained, so the `FirstHandler` parameter specifies that the handler should either be the first one called when the exception occurs (nonzero value) or the last one called (zero value). Subsequent `AddVectoredException-Handler` calls can update the order. For example, if two handlers are added, both with a zero `FirstHandler` value, the handlers will be called in the order in which they were added.

The return value is a handler to the exception handler (`NULL` indicates failure). This handle is the sole parameter to `RemoveVectoredExceptionHandler`, which returns a non-`NULL` value if it succeeds.

The successful return value is a pointer to the exception handler, that is, *`VectoredHandler`*. A `NULL` return value indicates failure.

*`VectoredHandler`* is a pointer to the handler function of the form:

```
LONG WINAPI VectoredHandler (PEXCEPTION_POINTERS
   ExceptionInfo)
```

PEXCEPTION_POINTERS is the address of an EXCEPTION_POINTERS structure with processor-specific and general information. This is the same structure returned by GetExceptionInformation and used in Program 4–4.

A VEH handler function should be fast so that the exception handler will be reached quickly. In particular, the handler should never access a synchronization object that might block the thread, such as a mutex (see Chapter 8). In most cases, the VEH simply accesses the exception structure, performs some minimal processing (such as setting a flag), and returns. There are two possible return values, both of which are familiar from the SEH discussion.

1. EXCEPTION_CONTINUE_EXECUTION—No more handlers are executed, SEH is not performed, and control is returned to the point where the exception occurred. As with SEH, this may not always be possible or advisable.
2. EXCEPTION_CONTINUE_SEARCH—The next VEH handler, if any, is executed. If there are no additional handlers, the stack is unwound to search for SEH handlers.

Exercise 4–9 asks you to add VEH to Programs 4–3 and 4–4.

## Summary

Windows SEH provides a robust mechanism for C programs to respond to and recover from exceptions and errors. Exception handling is efficient and can result in more understandable, maintainable, and safer code, making it an essential aid to defensive programming and higher-quality programs. Similar concepts are implemented in most languages and OSs, although the Windows solution allows you to analyze the exact cause of an exception.

Console control handlers can respond to external events that do not generate exceptions. VEH is a newer feature that allows functions to be executed before SEH processing occurs. VEH is similar to conventional interrupt handling.

## Looking Ahead

`ReportException` and exception and termination handlers are used as convenient in subsequent examples. Chapter 5 covers memory management, and in the process, SEH is used to detect memory allocation errors.

## Exercises

4–1. Extend Program 4–2 so that every call to `ReportException` contains sufficient information so that the exception handler can report precisely what error occurred and also the output file. Further enhance the program so that it can work with `$CONIN` and pipes (Chapter 11).

4–2. Extend Program 4–3 by generating memory access violations, such as array index out of bounds and arithmetic faults and other types of floating-point exceptions not illustrated in Program 4–3.

4–3. Augment Program 4–3 so as to print the value of the floating-point mask after enabling the exceptions. Are all the exceptions actually enabled? Explain the results.

4–4. What values do you get after a floating-point exception, such as division by zero? Can you set the result in the filter function as Program 4–3 attempts to do?

4–5. What happens in Program 4–3 if you do not clear the floating-point exception? Explain the results. *Hint:* Request an additional exception after the floating-point exception.

4–6. Extend Program 4–5 so that the handler routine raises an exception rather than returning. Explain the results.

4–7. Extend Program 4–5 so that it can handle shutdown and log-off signals.

4–8. Confirm through experiment that Program 4–5's handler routine executes concurrently with the main program.

4–9. Enhance Programs 4–3 and 4–4. Specifically, handle floating-point and arithmetic exceptions before invoking SEH.