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 `Generate-ConsoleCtrlEvent` (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.

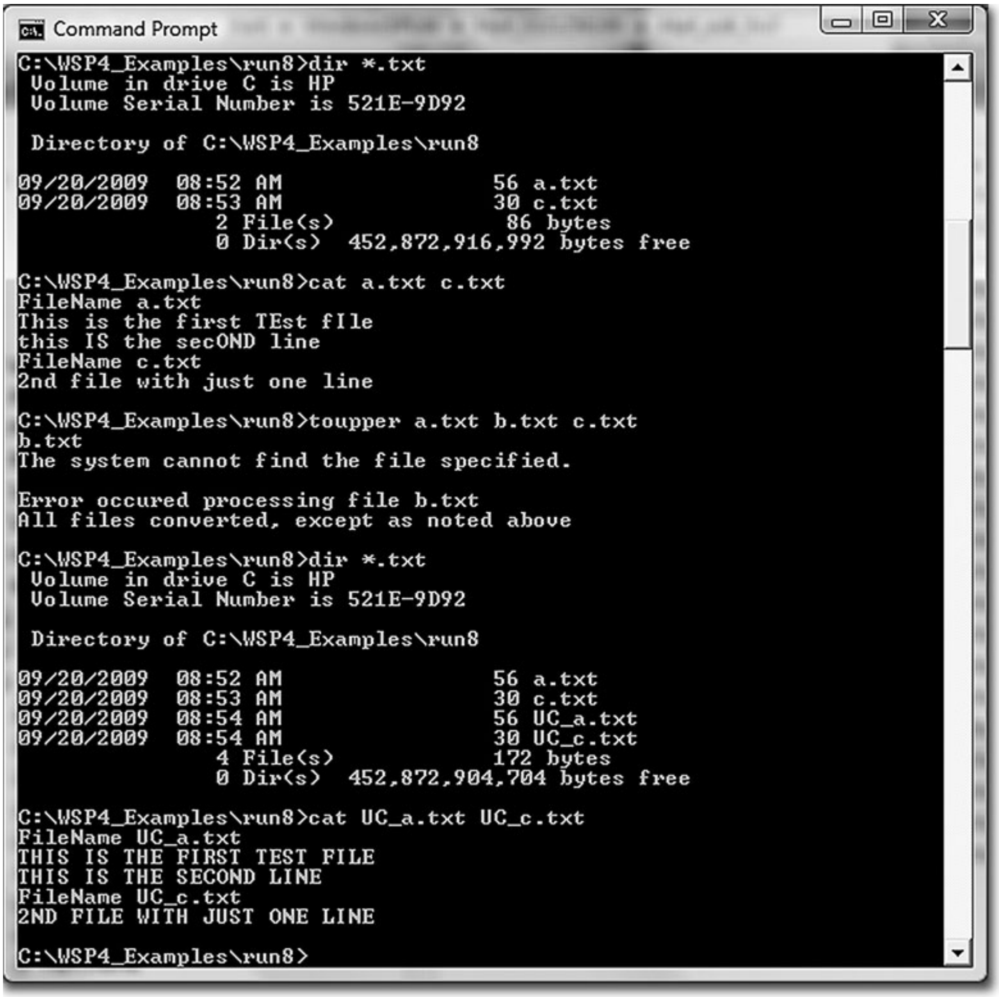
Note that this program depends on file sizes, so it will not work on objects for which `GetFileSizeEx` fails, such as a named pipe (Chapter 11). Furthermore, it fails for large text files longer than 4GB.

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.