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;
}
```

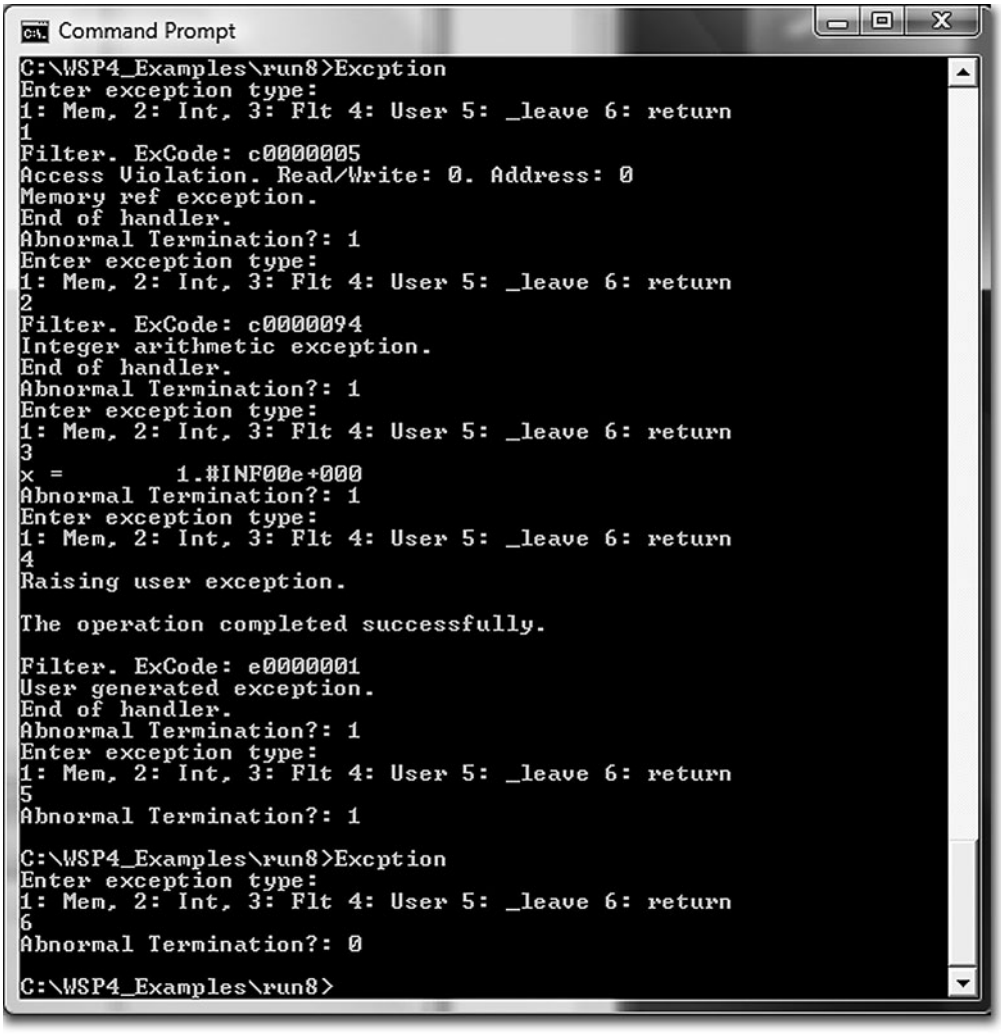
Program 4–4 shows the filter function used in Program 4–3. This function simply checks and categorizes the various possible exception code values. The code in the *Examples* file checks every possible value; here the function tests only for a few that are relevant to the test program.

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;
   }
}
```



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.

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 `AddVectoredExceptionHandler` 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.

CHAPTER

5 Memory Management, Memory-Mapped Files, and DLLs

Most programs require some form of dynamic memory management. This need arises whenever there is a need to create data structures whose size or number is not known at program build time. Search trees, symbol tables, and linked lists are common examples of dynamic data structures where the program creates new instances at run time.

Windows provides flexible mechanisms for managing a program's dynamic memory. Windows also provides memory-mapped files to associate a process's address space directly with a file, allowing the OS to manage all data movement between the file and memory so that the programmer never needs to deal with `ReadFile`, `WriteFile`, `SetFilePointer`, or the other file I/O functions. With memory-mapped files, the program can maintain dynamic data structures conveniently in permanent files, and memory-based algorithms can process file data. What is more, memory mapping can significantly speed up file processing, and it provides a mechanism for memory sharing between processes.

Dynamic link libraries (DLLs) are an essential special case of file mapping and shared memory in which files (primarily read-only code files) are mapped into the process address space for execution.

This chapter describes the Windows memory management and file mapping functions, illustrates their use and performance advantages with several examples, and describes both implicitly and explicitly linked DLLs.

Windows Memory Management Architecture

Win32 (the distinction between Win32 and Win64 is important here) is an API for the Windows 32-bit OS family. The "32-bitness" manifests itself in memory addresses, and pointers (`LPCTSTR`, `LPDWORD`, and so on) are 4-byte (32-bit) objects. The Win64 API provides a much larger virtual address space with 8-byte, 64-bit pointers and is a natural evolution of Win32. Nonetheless, use care to ensure that your applications can be targeted at both platforms; the examples have all been tested on both 64-bit and 32-bit systems, and 32-bit and 64-bit executables are available in the *Examples* file. With the example programs, there are comments about changes that were required to support Win64.

Every Windows process, then, has its own private virtual address space of either 4GB (2^{32} bytes) or 16EB (16 exabytes or 2^{64} bytes[1]). Win32 makes at least half of this (2–3GB; 3GB must be enabled at boot time) available to a process. The remainder of the virtual address space is allocated to shared data and code, system code, drivers, and so on.

The details of these memory allocations, although interesting, are not important here. From the programmer's perspective, the OS provides a large address space for code, data, and other resources. This chapter concentrates on exploiting Windows memory management without being concerned with OS implementation. Nonetheless, a very short overview follows.

Memory Management Overview

The OS manages all the details of mapping virtual to physical memory and the mechanics of page swapping, demand paging, and the like. This subject is discussed thoroughly in OS texts. Here's a brief summary.

- The computer has a relatively small amount of physical memory; 1GB is the practical minimum for 32-bit Windows XP, and much larger physical memories are typical.[2]
- Every process—and there may be several user and system processes—has its own virtual address space, which may be much larger than the physical memory available. For example, the virtual address space of a 4GB process is two

[1] Current systems cannot provide the full 2^{64}-byte virtual address space. 2^{44} bytes (16 terabytes) is a common processor limit at this time. This limit is certain to increase over time.

[2] Memory prices continue to decline, and "typical" memory sizes keep increasing, so it is difficult to define typical memory size. At the time of publication, even the most inexpensive systems contain 2GB, which is sufficient for Windows XP, Vista, and 7. Windows Server systems generally contain much more memory.