Looking Ahead

Chapter 4, Exception Handling, simplifies error and exception handling and extends the `ReportError` function to handle arbitrary exceptions.

Additional Reading

See Jerry Honeycutt's *Microsoft Windows Registry Guide* for information on registry programming and registry usage.

Exercises

3–1. Use the `GetDiskFreeSpaceEx` function to determine how the different Windows versions allocate file space sparsely. For instance, create a new file, set the file pointer to a large value, set the file size, and investigate the free space using `GetDiskFreeSpaceEx`. The same Windows function can also be used to determine how the disk is configured into sectors and clusters. Determine whether the newly allocated file space is initialized. `FreeSpace.c`, provided in the *Examples* file, is the solution. Compare the results for NT5 and NT6. It is also interesting to investigate how to make a file be sparse.

3–2. What happens if you attempt to set a file's length to a size larger than the disk? Does Windows fail gracefully?

3–3. Modify the `tail.c` program provided in the *Examples* file so that it does not use `SetFilePointer`; use overlapped structures. Also be sure that it works properly with files larger than 4GB.

3–4. Examine the "number of links" field obtained using the function `GetFileInformationByHandle`. Is its value always 1? Do the link counts appear to count hard links and links from parent directories and subdirectories? Does Windows open the directory as a file to get a handle before using this function? What about the shortcuts supported by the user interface?

3–5. Program 3–2 (`lsW`) checks for "`.`" and "`..`" to detect the names of the current and parent directories. What happens if there are actual files with these names? Can files have these names?

3–6. Does Program 3–2 list local times or UCT? If necessary, modify the program to give the results in local time.

3–7. Enhance Program 3–2 (`lsW`) so that it also lists the "`.`" and "`..`" (current and parent) directories (the complete program is in the *Examples* file). Also,

add options to display the file creation and last access times along with the last write time.

3–8. Further enhance Program 3–2 (`lsW`) to remove all uses of `SetCurrentDirectory`. This function is undesirable because an exception or other fault could leave you in an expected working directory.

3–9. Create a file deletion command, `rm`, by modifying the `ProcessItem` function in Program 3–2. A solution is in the *Examples* file.

3–10. Enhance the file copy command, `cpW`, from Chapter 1 so that it will copy files to a target directory. Further extensions allow for recursive copying (`-r` option) and for preserving the modification time of the copied files (`-p` option). Implementing the recursive copy option will require that you create new directories.

3–11. Write an `mv` command, similar to the UNIX command of the same name, which will move a complete directory. One significant consideration is whether the target is on a different drive from that of the source file or directory. If it is, copy the file(s); otherwise, use `MoveFileEx`.

3–12. Enhance Program 3–3 (`touch`) so that the new file time is specified on the command line. The UNIX command allows the time stamp to appear (optionally) after the normal options and before the file names. The format for the time is `MMddhhmm[yy]`, where the uppercase `MM` is the month and `mm` is for minutes.

3–13. Program 3–1 (`RecordAccess`) is written to work with large NTFS file systems. If you have sufficient free disk space, test this program with a huge file (length greater than 4GB, and considerably larger if possible); see Run 3–2. Verify that the 64-bit arithmetic is correct. It is not recommended that you perform this exercise on a network drive without permission from the network administrator. Don't forget to delete the test file on completion; disk space is cheap, but not so cheap that you want to leave orphaned huge files.

3–14. Write a program that locks a specified file and holds the lock for a long period of time (you may find the `Sleep` function useful). While the lock is held, try to access the file (use a text file) with an editor. What happens? Is the file properly locked? Alternatively, write a program that will prompt the user to specify a lock on a test file. Two instances of the program can be run in separate windows to verify that file locking works as described. `TestLock.c` in the *Examples* file is a solution to this exercise.

3–15. Investigate the Windows file time representation in `FILETIME`. It uses 64 bits to count the elapsed number of 100-nanosecond units from January 1,

1601. When will the time expire? When will the UNIX file time representation expire?

3–16. Write an interactive utility that will prompt the user for a registry key name and a value name. Display the current value and prompt the user for a new value. The utility could use command prompt `cd` and `dir` commands to illustrate the similarities (and differences) between the registry and file systems.

3–17. This chapter, along with most other chapters, describes the most important functions. There are often other functions that may be useful. The MSDN pages for each function provide links to related functions. Examine several, such as `FindFirstFileEx`, `FindFirstFileTransacted`, `ReplaceFile`, `SearchPath`, and `WriteFileGather`. Some of these functions are not available in all Windows versions.

This page intentionally left blank

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.

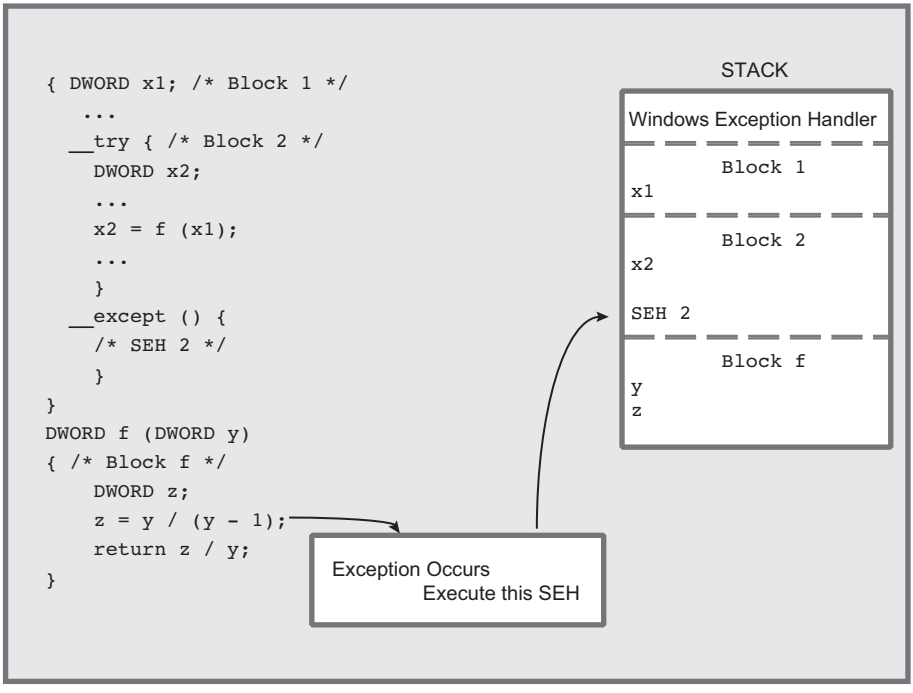

Figure 4–1 SEH, Blocks, and Functions

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 ...
        }
}
```

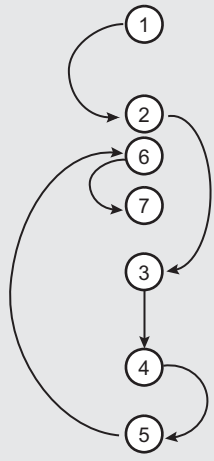

Figure 4–2 Exception Handling Sequence

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-