

第五部分 结构化异常处理

第23章 结束处理程序

你可以闭上眼睛，想象有这样一个编程环境，在这个环境里，你编写的代码永远不会出错。总有足够的内存，没有人会传递给你一个无效的指针，你需要的文件总是存在。如果按照这种假想，编写程序不是很愉快的事情吗？那样的话，程序会非常容易编写、阅读和理解。我们不会再让每个程序中通篇的if语句和goto语句搞得昏头昏脑，只需从头到尾书写代码就是了。

如果说这种直接的编程环境只是一个美妙的梦想的话，那么结构化异常处理（SEH）就会给你一个现实的惊喜。使用SEH的好处就是当你编写程序时，只需要关注程序要完成的任务。如果在运行时发生什么错误，系统会发现并将发生的问题通知你。

利用SEH，你可以完全不用考虑代码里是不是有错误，这样就把主要的工作同错误处理分离开来。这样的分离，可以使你集中精力处理眼前的工作，而将可能发生的错误放在后面处理。

微软在Windows中引入SEH的主要动机是为了便于操作系统本身的开发。操作系统的开发人员使用SEH，使得系统更加强壮。我们也可以使用SEH，使我们的自己的程序更加强壮。

使用SEH所造成的负担主要由编译程序来承担，而不是由操作系统承担。当异常块（exception block）出现时，编译程序要生成特殊的代码。编译程序必须产生一些表（table）来支持处理SEH的数据结构。编译程序还必须提供回调（callback）函数，操作系统可以调用这些函数，保证异常块被处理。编译程序还要负责准备栈结构和其他内部信息，供操作系统使用和参考。在编译程序中增加SEH支持不是一件容易的事。不同的编译程序厂商会以不同的方式实现SEH，这一点并不让人感到奇怪。幸亏我们可以不必考虑编译程序的实现细节，而只使用编译程序的SEH功能。

由于各编译程序的实现上存在着差别，这样以特定的方式用特定的代码例子讨论SEH的优点就很困难。但大多数编译程序厂商都采用微软建议的文法。本书中的例子使用的文法和关键字可能与其他一些公司编译程序所使用的不同，但主要的SEH概念是一样的。本章使用Microsoft Visual C++编译程序的文法。

注意 不要将结构化异常处理同C++的异常处理相混淆。C++异常处理是一种不同形式的异常处理，其形式是使用C++关键字catch和throw。微软的Visual C++也支持C++的异常处理，并且在内部实现上利用了已经引入到编译程序和Windows操作系统的结构化异常处理的功能。

SEH实际包含两个主要功能：结束处理（termination handling）和异常处理（exception handling）。本章讨论结束处理，下一章讨论异常处理。

一个结束处理程序能够确保去调用和执行一个代码块（结束处理程序，termination handler），而不管另外一段代码（保护体，guarded body）是如何退出的。结束处理程序的文法结构（使用微软的Visual C++编译程序）如下：

```
__try {
    // Guarded body
    :
}
__finally {
    // Termination handler
    :
}
```

--try和--finally关键字用来标出结束处理程序两段代码的轮廓。在上面的代码段中，操作系统和编译程序共同来确保结束处理程序中的 --finally代码块能够被执行，不管保护体（try块）是如何退出的。不论你在保护体中使用 return，还是 goto，或者是 longjump，结束处理程序（finally块）都将被调用。下面将通过几个例子来说明这一点。

23.1 通过例子理解结束处理程序

由于在使用 SEH 时，编译程序和操作系统直接参与了程序代码的执行，为了解释 SEH 如何工作，最好的办法就是考察源代码例子，讨论例子中语句执行的次序。

因此，在下面几节给出不同的源代码片段，对每一个片段解释编译程序和操作系统如何改变代码的执行次序。

23.2 Funcenstein1

为了甄别使用结束处理程序的各种情况，我们来考察更具体的代码例子。

```
DWORD Funcenstein1() {
    DWORD dwTemp;

    // 1. Do any processing here.
    :

    __try {
        // 2. Request permission to access
        //    protected data, and then use it.
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;
    }
    __finally {
        // 3. Allow others to use protected data.
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // 4. Continue processing.
    return(dwTemp);
}
```

上面程序中加了标号的注释指出了代码执行的次序。在 Funcenstein1 中，使用 try-finally 块并没有带来很多好处。代码要等待信标（semaphore），改变保护数据的内容，保存局部变量 dwTemp 的新值，释放信标，将新值返回给调用程序。

23.3 Funcenstein2

现在我们把这个程序稍微改动一下，看会发生什么事情。

```
DWORD Funcenstein2() {
```

```
DWORD dwTemp;

// 1. Do any processing here.
:

__try {
    // 2. Request permission to access
    //    protected data, and then use it.
    WaitForSingleObject(g_hSem, INFINITE);

    g_dwProtectedData = 5;
    dwTemp = g_dwProtectedData;

    // Return the new value.
    return(dwTemp);
}
__finally {
    // 3. Allow others to use protected data.
    ReleaseSemaphore(g_hSem, 1, NULL);
}

// Continue processing--this code
// will never execute in this version.
dwTemp = 9;
return(dwTemp);
}
```

在Funcenstein2中，try块的末尾增加了一个return语句。这个return语句告诉编译程序在这里要退出这个函数并返回 dwTemp变量的内容，现在这个变量的值是5。但是，如果这个return语句被执行，该线程将不会释放信标，其他线程也就不能再获得对信标的控制。可以想象，这样的执行次序会产生很大的问题，那些等待信标的线程可能永远不会恢复执行。

通过使用结束处理程序，可以避免 return语句的过早执行。当return语句试图退出try块时，编译程序要确保finally块中的代码首先被执行。要保证finally块中的代码在try块中的return语句退出之前执行。Funcenstein2中，将对ReleaseSemaphore的调用放在结束处理程序块中，保证信标总会被释放。这样就不会造成一个线程一直占有信标，否则将意味着所有其他等待信标的线程永远不会被分配CPU时间。

在finally块中的代码执行之后，函数实际上就返回。任何出现在 finally块之下的代码将不再执行，因为函数已在try块中返回。所以这个函数的返回值是5，而不是9。

读者可能要问编译程序是如何保证在try块可以退出之前执行finally块的。当编译程序检查源代码时，它看到在try块中有return语句。这样，编译程序就生成代码将返回值（本例中是5）保存在一个编译程序建立的临时变量中。编译程序然后再生成代码来执行finally块中包含的指令，这称为局部展开。更特殊的情况是，由于try块中存在过早退出的代码，从而产生局部展开，导致系统执行finally块中的内容。在finally块中的指令执行之后，编译程序临时变量的值被取出并从函数中返回。

可以看到，要完成这些事情，编译程序必须生成附加的代码，系统要执行额外的工作。在不同的CPU上，结束处理所需要的步骤也不同。例如，在Alpha处理器上，必须执行几百个甚至几千个CPU指令来捕捉try块中的过早返回并调用finally块。在编写代码时，就应该避免引起结束处理程序的try块中的过早退出，因为程序的性能会受到影响。本章后面，将讨论 __leave关键字，它有助于避免编写引起局部展开的代码。

设计异常处理的目的是用来捕捉异常的——不常发生的语法规则的异常情况（在我们的例子中，就是过早返回）。如果情况是正常的，明确地检查这些情况，比起依赖操作系统和编译

程序的 SEH 功能来捕捉常见的事情要更有效。

注意当控制流自然地离开 try 块并进入 finally 块（就像在 Funcenstein1 中）时，进入 finally 块的系统开销是最小的。在 x86 CPU 上使用微软的编译程序，当执行离开 try 块进入 finally 块时，只有一个机器指令被执行，读者可以在自己的程序中注意到这种系统开销。当编译程序要生成额外的代码，系统要执行额外的工作时（如同在 Funcenstein2 中），系统开销就很值得注意了。

23.4 Funcenstein3

现在我们对函数再做修改，看会出现什么情况：

```
DWORD Funcenstein3() {
    DWORD dwTemp;

    // 1. Do any processing here.
    :

    __try {
        // 2. Request permission to access
        //    protected data, and then use it.
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;

        // Try to jump over the finally block.
        goto ReturnValue;
    }

    __finally {
        // 3. Allow others to use protected data.
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    dwTemp = 9;
    // 4. Continue processing.
    ReturnValue:
    return(dwTemp);
}
```

在 Funcenstein3 中，当编译程序看到 try 块中的 goto 语句，它首先生成一个局部展开来执行 finally 块中的内容。这一次，在 finally 块中的代码执行之后，在 ReturnValue 标号之后的代码将执行，因为在 try 块和 finally 块中都没有返回发生。这里的代码使函数返回 5。而且，由于中断了从 try 块到 finally 块的自然流程，可能要蒙受很大的性能损失（取决于运行程序的 CPU）。

23.5 Funcfurter1

现在我们来考察另外的情况，这里可以真正显示结束处理的价值。看下面的函数：

```
DWORD Funcfurter1() {
    DWORD dwTemp;

    // 1. Do any processing here.
    :
    :
```

```

__try {
    // 2. Request permission to access
    //    protected data, and then use it.
    WaitForSingleObject(g_hSem, INFINITE);

    dwTemp = Funcinator(g_dwProtectedData);
}
__finally {
    // 3. Allow others to use protected data.
    ReleaseSemaphore(g_hSem, 1, NULL);
}

// 4. Continue processing.
return(dwTemp);
}

```

现在假想一下，try块中的Funcinator函数调用包含一个错误，会引起一个无效内存访问。如果没有SEH，在这种情况下，将会给用户显示一个很常见的Application Error对话框。当用户忽略这个错误对话框，该进程就结束了。当这个进程结束（由于一个无效内存访问），信标仍将被占用并且永远不会被释放，这时候，任何等待信标的其他进程中的线程将不会被分配CPU时间。但若将对ReleaseSemaphore的调用放在finally块中，就可以保证信标获得释放，即使某些其他函数会引起内存访问错误。

如果结束处理程序足够强，能够捕捉由于无效内存访问而结束的进程，我们就可以相信它也能够捕捉setjump和longjump的结合，还有那些简单语句如break和continue。

23.6 突击测验：FuncaDoodleDoo

现在做一个测试，读者判断一下下面的函数返回什么值？

```

DWORD FuncaDoodleDoo() {
    DWORD dwTemp = 0;

    while (dwTemp < 10) {

        __try {
            if (dwTemp == 2)
                continue;

            if (dwTemp == 3)
                break;
        }
        __finally {
            dwTemp++;
        }

        dwTemp++;
    }

    dwTemp += 10;
    return(dwTemp);
}

```

我们一步一步地分析函数做了什么。首先dwTemp被设置成0。try块中的代码执行，但两个if语句的值都不为TRUE。执行自然移到finally块中的代码，在这里dwTemp增加到1。然后finally块之后的指令又增加dwTemp，使它的值成为2。

当循环继续，dwTemp为2，try块中的continue语句将被执行。如果没有结束处理程序在从try块中退出之前强制执行finally块，执行就立即跳回while测试，dwTemp不会被改变，将出现无限（死）循环。利用一个结束处理程序，系统知道continue语句要引起控制流过早退出try块，而将执行移到finally块。在finally块中，dwTemp被增加到3。但finally块之后的代码不执行，因为控制流又返回到continue，再到循环的开头。

现在我们处理循环的第三次重复。这一次，第一个if语句的值是FALSE，但第二个语句的值是TRUE。系统又能够捕捉要跳出try块的企图，并先执行finally块中的代码。现在dwTemp增加到4。由于break语句被执行，循环之后程序部分的控制恢复。这样，finally块之后的循环中的代码没有执行。循环下面的代码对dwTemp增加10，这时dwTemp的值是14，这就是调用这个函数的结果。当然，实际上我们不会写出FuncuDoodleDoo这样的代码。这里将continue和break语句放在代码的中间，是为了说明结束处理程序的操作。

尽管结束处理程序可以捕捉try块过早退出的大多数情况，但当线程或进程被结束时，它不能引起finally块中的代码执行。当调用ExitThread或ExitProcess时，将立即结束线程或进程，而不会执行finally块中的任何代码。另外，如果由于某个程序调用TerminateThread或TerminateProcess，线程或进程将死掉，finally块中的代码也不执行。某些C运行期函数（例如abort）要调用ExitProcess，也使finally块中的代码不能执行。虽然没有办法阻止其他程序结束你的一个线程或进程，但你可以阻止你自己过早调用ExitThread和ExitProcess。

23.7 Funcenstein4

我们再看一种异常处理的情况。

```
DWORD Funcenstein4() {
    DWORD dwTemp;
    // 1. Do any processing here.
    :
    :
    __try {
        // 2. Request permission to access
        //    protected data, and then use it.
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;

        // Return the new value.
        return(dwTemp);
    }
    __finally {
        // 3. Allow others to use protected data.
        ReleaseSemaphore(g_hSem, 1, NULL);
        return(103);
    }

    // Continue processing--this code will never execute.
    dwTemp = 9;
    return(dwTemp);
}
```

在Funcenstein4中，try块将要执行，并试图将dwTemp的值（5）返回给Funcenstein4的调用者。如同对Funcenstein2的讨论中提到的，试图从try块中过早的返回将导致产生代码，把返回

值置于由编译程序建立的临时变量中。然后，`finally`块中的代码被执行。在这里，与`Funcenstein2`不同的是在`finally`块中增加了一个`return`语句。`Funcenstein4`会向调用程序返回5还是103？这里的答案是103，因`finally`块中的`return`语句引起值103存储在值5所存储的临时变量中，覆盖了值5。当`finally`块完成执行，现在临时变量中的值（103）从`Funcenstein4`返回给调用程序。

我们已经看到结束处理程序在补救`try`块中的过早退出的执行方面很有效，但也看到结束处理程序由于要阻止`try`块的过早退出而产生了我们不希望有的结果。更好的办法是在结束处理程序的`try`块中避免任何会引起过早退出的语句。实际上，最好将`return`、`continue`、`break`、`goto`等语句从结束处理程序的`try`块和`finally`块中移出去，放在结束处理程序的外面。这样做会使编译程序产生较小的代码，因为不需要再捕捉`try`块中的过早退出，也使编译程序产生更快的代码（因为执行局部展开的指令也少）。另外，代码也更容易阅读和维护。

23.8 Funcarama1

我们已经谈过了结束处理程序的基本语法和语意。现在看一看如何用结束处理程序来简化一个更复杂的编程问题。先看一个完全没有利用结束处理程序的函数：

```
BOOL Funcarama1() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        return(FALSE);
    }
    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf == NULL) {
        CloseHandle(hFile);
        return(FALSE);
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
        CloseHandle(hFile);
        return(FALSE);
    }

    // Do some calculation on the data.
    :

    // Clean up all the resources.
    VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    CloseHandle(hFile);
    return(TRUE);
}
```

`Funcarama1`中的各种错误检查使这个函数非常难以阅读，也使这个函数难以理解、维护和修改。

23.9 Funcarama2

当然，可以重新编写Funcarama1，使它更清晰一些，也更容易理解。

```

BOOL Funcarama2() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL fOk, fSuccess = FALSE;

    hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {

        pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);

        if (pvBuf != NULL) {

            fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
            if (fOk && (dwNumBytesRead != 0)) {
                // Do some calculation on the data.
                :
                :
                fSuccess = TRUE;
            }

        }

        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    }

    CloseHandle(hFile);
    return(fSuccess);
}

```

Funcarama2尽管比Funcarama1容易理解一些，但还是不好修改和维护。而且，当增加更多的条件语句时，这里的缩排格式就会走向极端，很快就到屏幕的最右边。

23.10 Funcarama3

我们使用一个SEH结束处理程序来重新编写Funcarama1。

```

DWORD Funcarama3() {

    // IMPORTANT: Initialize all variables to assume failure.
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;

    __try {
        DWORD dwNumBytesRead;
        BOOL fOk;

        hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
        if (hFile == INVALID_HANDLE_VALUE) {
            return(FALSE);
        }
    }
}

```



```

pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
if (pvBuf == NULL) {
    return(FALSE);
}

fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
if (!fOk || (dwNumBytesRead != 1024)) {
    return(FALSE);
}

// Do some calculation on the data.
:
}

__finally {
    // Clean up all the resources.
    if (pvBuf != NULL)
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    if (hFile != INVALID_HANDLE_VALUE)
        CloseHandle(hFile);
}
// Continue processing.
return(TRUE);
}

```

Funcarama3版的真正好处是函数的所有清理（cleanup）代码都局部化在一个地方且只在一个地方：finally块。如果需要在这个函数中再增加条件代码，只需在 finally块中简单地增加一个清理行，不需要回到每个可能失败的地方添加清理代码。

23.11 Funcarama4：最终的边界

Funcarama3版本的问题是系统开销。就像在 Funcenstein4中讨论的，应该尽可能避免在 try块中使用return语句。

为了帮助避免在 try块中使用return语句，微软在其C/C++ 编译程序中增加了另一个关键字--leave。这里是 Funcarma4版，它使用了--leave关键字：

```

DWORD Funcarama4() {

    // IMPORTANT: Initialize all variables to assume failure.
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;

    // Assume that the function will not execute successfully.
    BOOL fFunctionOk = FALSE;

    __try {
        DWORD dwNumBytesRead;
        BOOL fOk;
        hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
        if (hFile == INVALID_HANDLE_VALUE) {
            __leave;
        }

        pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    }
}

```

```
    if (pvBuf == NULL) {
        __leave;
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
        __leave;
    }

    // Do some calculation on the data.
    :
    // Indicate that the entire function executed successfully.
    fFunctionOk = TRUE;
}
__finally {
    // Clean up all the resources.
    if (pvBuf != NULL)
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    if (hFile != INVALID_HANDLE_VALUE)
        CloseHandle(hFile);
}
// Continue processing.
return(fFunctionOk);
}
```

在try块中使用__leave关键字会引起跳转到try块的结尾。可以认为是跳转到try块的右大括号。由于控制流自然地由try块中退出并进入finally块，所以不产生系统开销。当然，需要引入一个新的Boolean型变量fFunctionOk，用来指示函数是成功或失败。这是比较小的代价。

当按照这种方式利用结束处理程序来设计函数时，要记住在进入try块之前，要将所有资源句柄初始化为无效的值。然后，在finally块中，查看哪些资源被成功的分配，就可以知道哪些要释放。另外一种确定需要释放资源的办法是对成功分配的资源设置一个标志。然后，finally块中的代码可以检查标志的状态，来确定资源是否要释放。

23.12 关于finally块的说明

我们已经明确区分了强制执行finally块两种情况：

- 从try块进入finally块的正常控制流。
- 局部展开：从try块的过早退出（goto、longjump、continue、break、return等）强制控制转移到finally块。

第三种情况，全局展开（global unwind），在发生的时候没有明显的标识，我们在本章前面Funcfurther1函数中已经见到。在Funcfurther1的try块中，有一个对Funcinator函数的调用。如果Funcinator函数引起一个内存访问违规（memory access violation），一个全局展开会使Funcfurther1的finally块执行。下一章将详细讨论全局展开。

由于以上三种情况中某一种的结果而导致finally块中的代码开始执行。为了确定是哪一种情况引起finally块执行，可以调用内部函数（或内蕴函数，intrinsic function）Abnormal Termination：

```
BOOL AbnormalTermination();
```

这个内部函数只在finally块中调用，返回一个Boolean值。指出与finally块相结合的try块是否过早退出。换句话说，如果控制流离开try块并自然进入finally块，AbnormalTermination将返

回FALSE。如果控制流非正常退出 try块——通常由于 goto、return、break或continue语句引起的局部展开，或由于内存访问违规或其他异常引起的全局展开——对AbnormalTermination的调用将返回TRUE。没有办法区别finally块的执行是由于全局展开还是由于局部展开。但这通常不会成为问题，因为可以避免编写执行局部展开的代码。

注意 内部函数是编译程序识别的一种特殊函数。编译程序为内部函数产生内联（inline）代码而不是生成调用函数的代码。例如，memcpy是一个内部函数（如果指定/Oi编译程序开关）。当编译程序看到一个对memcpy的调用，它直接将memcpy的代码插入调用memcpy的函数中，而不是生成一个对memcpy函数的调用。其作用是代码的长度增加了，但执行速度加快了。

23.13 Funcfurter2

这里是Funcfurter2，说明AbnormalTermination内部函数的使用。

```
DWORD Funcfurter2() {
    DWORD dwTemp;

    // 1. Do any processing here.
    :

    __try {
        // 2. Request permission to access
        //    protected data, and then use it.
        WaitForSingleObject(g_hSem, INFINITE);

        dwTemp = Funcinator(g_dwProtectedData);
    }
    __finally {
        // 3. Allow others to use protected data.
        ReleaseSemaphore(g_hSem, 1, NULL);

        if (!AbnormalTermination()) {
            // No errors occurred in the try block, and
            // control flowed naturally from try into finally.
            :
        }
        else {
            // Something caused an exception, and
            // because there is no code in the try block
            // that would cause a premature exit, we must
            // be executing in the finally block
            // because of a global unwind.

            // If there were a goto in the try block,
            // we wouldn't know how we got here.
            :
        }
    }

    // 4. Continue processing.
    return(dwTemp);
}
```

我们已经知道如何编写结束处理程序了，下一章读者会看到异常过滤程序和异常处理程序

更有用,更重要。在继续之前,回顾一下使用结束处理程序的理由:

- 简化错误处理,因所有的清理工作都在一个位置并且保证被执行。
- 提高程序的可读性。
- 使代码更容易维护。
- 如果使用得当,具有最小的系统开销。

23.14 SEH结束处理示例程序

SEHTerm程序,“23 SEHTerm.exe”(见清单23-1),说明了结束处理程序如何工作。这个程序的源代码和资源文件在本书附带的CD-ROM的23-SEHTerm目录下。

当运行这个程序时,主线程进入一个try块。在这个try块中显示图23-1所示的消息框。

这个消息框询问是否要程序存取一个无效的内存字节(许多程序的编写不这样考虑,只是不加询问地存取无效内存)。我们来看一看当选择Yes按钮时会发生什么。在这种情况下,线程试图向内存地址NULL写一个5。向地址NULL的写操作总要引起存取异常。当该线程引发一个存取异常时,系统显示图23-2所示的消息框(在Windows98中)。

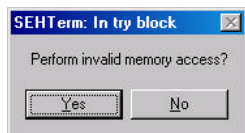


图23-1 运行SEHTerm时显示的消息框

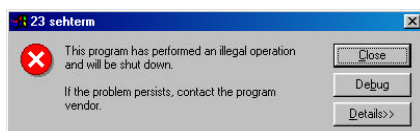


图23-2 在Windows 98中线程引发存取异常时显示的消息框

在Windows2000中,消息框如图23-3所示。

如果现在选择Close按钮(在windows98中)或OK按钮(在Windows 2000中),进程将结束。但在源代码中有一个finally块,所以在进程结束之前,finally块要先执行。这个finally块显示图23-4所示的消息框。



图23-3 在Windows 2000中,线程引发存取异常时显示的消息框



图23-4 SEH Term: In finally block 消息框

finally块执行是因为相应的try块非正常退出。当忽略这个消息框,进程就真的结束了。

我们再运行这个程序。这一次,我们选择No按钮,不去试图存取无效的内存。当点击No按钮,线程从try块中自然流进finally块。然后,finally块显示图23-5所示的消息框。

注意,这次消息框指出try块正常退出。当忽略这个消息框时,线程离开finally块,并显示图23-6所示的消息框。

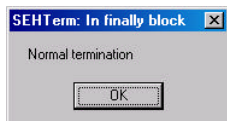


图23-5 选择No按钮时出现的SEH Term: In finally block 消息框

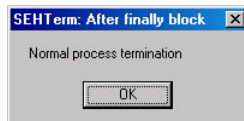


图23-6 忽略图23-5所示的消息框时,显示的SEHTerm: After finally block 消息框

当忽略这个消息框时，因 WinMain 返回，进程自然结束时。注意当进程结束，由于有一个存取异常，你实际看不到最后一个消息框。

清单23-1 SEHTerm 示例程序



SEHTerm.cpp

```

/*****
Module: SEHTerm.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h"    /* See Appendix A. */
#include <tchar.h>

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    __try {
        int n = MessageBox(NULL, TEXT("Perform invalid memory access?"),
            TEXT("SEHTerm: In try block"), MB_YESNO);

        if (n == IDYES) {
            * (PBYTE) NULL = 5; // This causes an access violation.
        }
    }
    __finally {
        PCTSTR psz = AbnormalTermination()
            ? TEXT("Abnormal termination") : TEXT("Normal termination");
        MessageBox(NULL, psz, TEXT("SEHTerm: In finally block"), MB_OK);
    }

    MessageBox(NULL, TEXT("Normal process termination"),
        TEXT("SEHTerm: After finally block"), MB_OK);

    return(0);
}

//////////////////////////////////// End of File //////////////////////////////////

```

