第25章　未处理异常和C++异常

前一章讨论了当一个异常过滤器返回EXCEPTION_CONTINUE_SEARCH时会发生什么事情。返回EXCEPTION_CONTINUE_SEARCH 是告诉系统继续上溯调用树，去寻找另外的异常过滤器。但是当每个过滤器都返回EXCEPTION_CONTINUE_SEARCH时会出现什么情况呢？在这种情况下，就出现了所谓的“未处理异常”（Unhandled exception）。

在第6章里我们已经知道，每个线程开始执行，实际上是利用 Kernel32.dll中的一个函数来调用BaseProcessStart或BaseThreadStart。这两个函数实际是一样的，区别在于一个函数用于进程的主线程（Primary thread）：

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
   __try {
      ExitThread((pfnStartAddr)());
   }
   __except (UnhandledExceptionFilter(GetExceptionInformation())) {
      ExitProcess(GetExceptionCode());
   }
   // NOTE: We never get here
}
```

另一个函数用于进程的所有辅助线程（Secondary thread）：

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
   __try {
      ExitThread((pfnStartAddr)(pvParam));
   }
   __except (UnhandledExceptionFilter(GetExceptionInformation())) {
      ExitProcess(GetExceptionCode());
   }
   // NOTE: We never get here
}
```

注意这两个函数都包含一个SEH框架。每个函数都有一个try块，并从这个try块里调用主线程或辅助线程的进入点函数。所以，当线程引发一个异常，所有过滤器都返回 EXCEPTION_CONTINUE_SEARCH时，将会自动调用一个由系统提供的特殊过滤器函数： UnhandledExceptionFilter。

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

这个函数负责显示一个消息框，指出有一个进程的线程存在未处理的异常，并且能让用户结束或调试这个进程。在Windows 98中，这个消息框如图25-1所示。

在Windows 2000中，消息框如图25-2所示。

在Windows 2000下的消息框中，第一段文字指出发生了哪一种异常，并给出在进程地址空间中产生异常的指令的地址。在这里，消息框中指出发生了一个内存访问异常，系统报告了要访问的无效内存地址，指出由于试图读这个内存而引发了异常。 UnhandledExceptionFilter函数从这个异常生成的EXCEPTION_RECORD结构的ExceptionInformation成员中获取这些附加的信息。

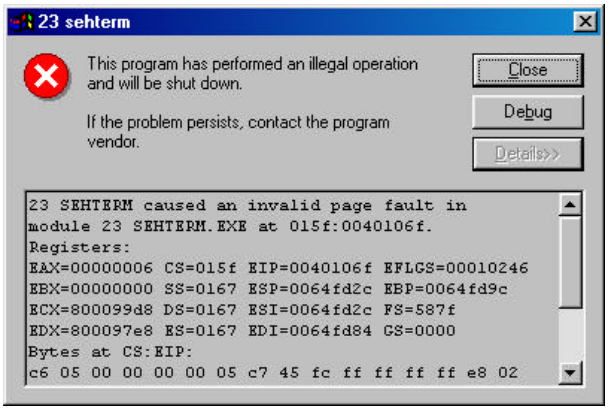

图25-1　Windows 98下显示进程的线程存在未处理的异常的消息框

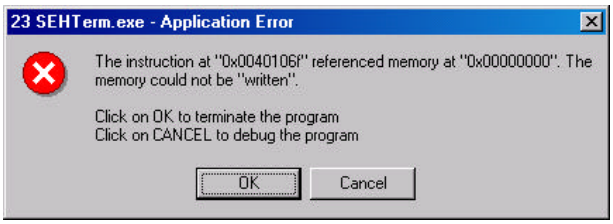

图25-2　Windows 2000下的消息框

在消息框中的异常描述之后，又提供给用户两种选择。第一种选择是点击 OK按钮，这将导致UnhandledExceptionFilter返回EXCEPTION_EXECUTE_HANDLER。这又引起全局展开的发生，所有的finally块都要执行，然后BaseProcessStart或BaseThreadStart中的处理程序执行。这两个处理程序都叫ExitProcess，意思是退出进程，这就是程序结束的原因。注意进程的退出代码就是异常代码。还要注意是进程的线程废（ kill）了进程本身，而不是操作系统。这也意味着程序员可以控制这种行为并可以改变它。

第二种选择是点击 Cancel按钮。在这里程序员的梦想成真。当点击 Cancel按钮时，UnhandledExceptionFilter试图加载一个调试程序，并将这个调试程序挂接在进程上。通过将调试程序附在进程上，可以检查全局、局部和静态变量的状态，设置断点，检查调用树，重新启动进程，以及调试一个进程时可以做的任何事情。

这里真正的好处是，你可以在程序运行当中错误发生时就处理错误。在其他操作系统中，必须在调试程序调用程序时才能对其进行调试。在那些操作系统中，如果一个进程中发生了一个异常，必须结束这个进程，启动调试程序，调用程序，再使用调试程序。这样，你必须重新产生这个错误，才有可能去修正它。但谁能记住问题最初发生时的各种条件和各变量的值？按这种方式解决程序错误问题非常困难。将调试程序动态挂接在运行中的进程上，是 Windows最好的特性之一。

Windows 2000　本书着重讨论用户方式（user-mode）的程序开发。但对于运行在内核方式（kernel-mode）的线程引发的未处理异常会造成什么情况，读者也许会感兴趣。内核方式中的异常同用户方式中的异常是按同样方式处理的。如果一个低级虚拟内存函数产生一个异常，系统查找是否有内核方式异常过滤器准备处理这个异常。如果系统找不到一个异常过滤器来处理这个异常，则异常就是未处理的。对于内核方式的异

常，未处理异常是在操作系统中或（更可能）在设备驱动程序中，而不是在应用程序中。这样一个未处理异常表示一个严重的程序错误（bug）！

如果一个未处理异常发生在内核方式，让系统继续运行是不安全的。所以系统在这种情况下不去调用UnhandledExceptionFilter函数，而是显示所谓的蓝屏死机（Blue Screen of Death）。显示画屏切换到只包含文本的蓝屏视频方式，并且计算机被停机（balt）。显示的文本告诉是哪个设备驱动程序被加载，并且该模块中包含有引发未处理异常的代码。用户应该记下这些信息并送交微软或设备驱动程序的厂商，以便修复这个错误。因为计算机被挂起，要想再做其他事情就必须重新启动计算机，所有未保存的工作都丢失了。

25.1 即时调试

随时将调试程序连接到任何进程的能力称为即时调试（Just-in-time Debugging）。这里我们对它如何工作稍加说明：当程序员点击Cancel按钮，就是告诉UnhandledExceptionFilter函数对进程进行调试。

在内部，UnhandledExceptionFilter调用调试程序，这需要查看下面的注册表子关键字：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
```

在这个子关键字里，有一个名为Debugger的数值，在安装Visual Studio时被设置成下面的值：

```
"C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe"
   -p %ld -e %ld
```

Windows 98 在Windows 98中，这些值不是存放在注册表中，而是存放在Win.ini文件中。

这一行代码是告诉系统要将哪一个程序（这里是MSDev. exe）作为调试程序运行。当然也可以选择其他调试程序。UnhandledExceptionFilter还在这个命令行中向调试程序传递两个参数。第一个参数是被调试进程的ID。第二个参数规定一个可继承的手工复位事件，这个事件是由UnhandledExceptionFilter按无信号状态建立的。厂商必须实现他们的调试程序，这样才能认识指定进程ID和事件句柄的-p和-e选项。

在进程ID和事件句柄都合并到这个串中之后，UnhandledExceptionFilter通过调用CreateProcess来执行调试程序。这时，调试程序进程开始运行并检查它的命令行参数。如果存在-p选项，调试程序取得进程ID，并通过调用DebugActiveProcess将自身挂接在该进程上。

```
BOOL DebugActiveProcess(DWORD dwProcessID);
```

一旦调试程序完成自身的挂接，操作系统将被调试者（debuggee）的状态通报给调试程序。例如，系统将告诉调试程序，在被调试的进程中有多少线程？哪些DDL加载到被调试进程的地址空间中？调试程序需要花时间来积累这些数据，以准备调试进程。在这些准备工作进行的时候，UnhandledExceptionFilter中的线程必须等待。为此，这要调用WaitForSingleObject函数并传递已经建立的手工复位事件的句柄作为参数。这个事件是按无信号状态建立起来的，所以被调试进程的线程要立即被挂起以等待事件。

在调试程序完全初始化之后，它要再检查它的命令行，找-e选项。如果该选项存在，调试程序取得相应的事件句柄并调用SetEvent。调试程序可以直接使用事件的句柄值，因为事件句柄具有创建的可继承性，并且被调试进程对UnhandledExceptionFilter函数的调用也使调试程序

进程成为一个子进程。

设定这个事件将唤醒被调试进程的线程。被唤醒的线程将有关未处理异常的信息传递给调试程序。调试程序接收这些通知并加载相应的源代码文件，再将自身放在引发异常的指令位置上。

还有，不必在调试进程之前等待异常的出现。可以随时将一个调试程序连接在任何进程上，只需运行“ MSDEV -p PID ”，其中PID是要调试的进程的ID。实际上，利用 Windows 2000 Task Manager，做这些事很容易。当观察Process标记栏时，可以选择一个进程，点击鼠标右键，并选择Debug菜单选项。这将引起 Task Manager去查看前面讨论过的注册表子关键字，调用CreateProcess，并传递所选定的进程的ID作为参数。在这里， Task Manager为事件句柄传送0值。

25.2　关闭异常消息框

有时候，在异常发生时，你可能不想在屏幕上显示异常消息框。例如，你可能不想让这些消息框出现在你产品的发售版本中。如果出现了消息框，很容易导致最终用户意外地启动调试程序来调试你的程序。最终用户只需点击一下消息框中的 Cancel按钮，就进入了不熟悉的、令人恐惶的区域 —— 调试程序。可以使用几种不同的方法来防止这种消息框的出现。

25.2.1　强制进程终止运行

为防止UnhandledExceptionFilter显示异常消息框，可以调用下面的 SetErrorModel函数，并向它传递一个SEM_NOGPFAULTERRORBOX标识符：

```
UINT SetErrorMode(UINT fuErrorMode);
```

然后，当调用UnhandledExceptionFilter函数来处理异常时，看到已经设置了这个标志，就会立即返回EXCEPTION_EXECUTE_HANDLER。这将导致全局展开并执行BaseProcessStart或BaseThreadStart中的处理程序。该处理程序结束进程。

笔者并不喜欢这种方式。在这种方式下，用户完全没有得到警告，程序只是自行消失。

25.2.2　包装一个线程函数

使用另外一种办法也可以避免出现这个消息框，就是针对主线程进入点函数（ main、wmain、WinMain或wWinMain）的整个内容安排一个try-except块。保证异常过滤器的结果值总是EXCEPTION_EXECUTE_HANDLER，这样就保证异常得到处理，防止了系统再调用UnhandledExceptionFilter函数。

在你的异常处理程序中，你可以显示一个对话框，在上面显示一些有关异常的诊断信息。用户可以记录下这些信息，并通报给你公司的客户服务部门，以便能够找到程序的问题根源。你应该建立这个对话框，这样用户只能结束程序而不能调用调试程序。

这种方法的缺点是它只能捕捉进程的主线程中发生的异常。如果其他线程在运行，并且其中有一个线程发生了一个未处理异常，系统就要调用内部的 UnhandledExceptionFilter函数。为了改正这一点，需要在所有的辅助线程进入点函数中包含 try-except块。

25.2.3　包装所有的线程函数

Windows还提供另外一个函数， SetUnhandledExceptionFilter，利用它可以按SEH格式包装所有的线程函数：

```
PTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
   PTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter);
```

在进程调用这些函数之后，进程的任何线程中若发生一个未处理的异常，就会导致调用程序自己的异常过滤器。需要将这个过滤器的地址作为参数传递给 SetUnhandledExceptionFilter。过滤器函数原型必须是下面的样子：

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

你可能会注意到这个函数同UnhandledExceptionFilter函数的形式是一样的。程序员可以在自己的异常过滤器中执行任何想做的处理，但要返回三个 EXCEPTION_*标识符中的一个。表25-1给出了当返回各标识符时所发生的事。

表25-1 返回标识符时的情况

标 识 符	出现的情况
EXCEPTION_EXECUTE_HANDLER	进程只是结束，因系统在其异常处理程序块中没有执行任何动作
EXCEPTION_CONTINUE_EXECUTION	从引起异常的指令处继续执行。可以参照PEXCEPTION_POINTERS参数修改异常信息
EXCEPTION_CONTINUE_SEARCH	执行正规的Windows UnhandledExceptionFilter函数

为了使UnhandledExceptionFilter函数再成为默认的过滤器，可以调用 SetUnhandled ExceptionFilter并传递 NULL给它。而且，每当设置一个新的未处理的异常过滤器时，SetUnhandled ExceptionFilter就返回以前安装的异常过滤器的地址。如果 Unhandled Exception Filter是当前所安装的过滤器，则这个返回的地址就是 NULL。如果你自己的过滤器要返回 EXCEPTION_CONTINUE_SEARCH，你就应该调用以前安装的过滤器，其地址通过 SetUnhandledExceptionFilter函数返回。

25.2.4 自动调用调试程序

现在再介绍关闭UnhandledExceptionFilter消息框的最后一种方法。在前面提到的同一个注册表子关键字里，还有另外一个数据值，名为 Auto。这个值用来规定UnhandledExceptionFilter是应该显示消息框，还是仅启动调试程序。如果 Auto设置成1，UnhandledExceptionFilter就不显示消息框向用户报告异常，而是立即调用调试程序。如果 Auto子关键设置成 0，UnhandledExceptionFilter就显示异常消息框，并按前面描述的那样操作。

25.3 程序员自己调用UnhandledExceptionFilter

UnhandledExceptionFilter函数是一个公开的、文档完备的 Windows函数，程序员可以直接在自己的代码中调用这个函数。这里是使用这个函数的一个例子：

```
void Funcadelic() {
   __try {
      .
      .
      .
   }
   __except (ExpFltr(GetExceptionInformation())) {
      .
      .
      .
   }
```

```
}

LONG ExpFltr(PEXCEPTION_POINTERS pEP) {
   DWORD dwExceptionCode = pEP->ExceptionRecord.ExceptionCode;

   if (dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) {
      // Do some work here....
      return(EXCEPTION_CONTINUE_EXECUTION);
   }

   return(UnhandledExceptionFilter(pEP));
}
```

在Funcadelic函数中，try块中的一个异常导致ExpFltr函数被调用。GetExceptionInformation的返回值作为参数传递给ExpFltr函数。在异常过滤器内，要确定异常代码并与EXCEPTION_ACCESS_VIOLATION相比较。如果发生一个存取违规，异常过滤器改正这个问题，并从过滤器返回EXCEPTION_CONTINUE_EXECUTION。这个返回值导致系统从最初引起异常的指令继续执行。

如果发生了其他异常，ExpFltr调用UnhandledExceptionFilter，将EXCEPTION_POINTERS结构的地址传递给它作为参数。UnhandledExceptionFilter显示消息框，可使程序员结束进程或开始调试进程。UnhandledExceptionFilter的返回值再由ExpFltr返回。

25.4　UnhandledExceptionFilter函数的一些细节

当笔者最初接触异常处理时，就认为如果能详细了解系统的UnhandledExceptionFilter函数所做的事情，就能得到许多有用的信息。为此，笔者仔细研究了这个函数。下面所列的步骤详细描述了UnhandledExceptionFilter函数的内部执行情况：

1) 如果发生一个存取违规并且是由于试图写内存（相对于读内存）引起的，系统要查看你是不是要修改一个.exe模块或DLL模块中的资源。默认时，资源是（而且应该是）只读的。试图修改资源会引起存取异常。然而16位Windows允许修改资源，从兼容性考虑，32位和64位Windows也应该允许修改资源。所以当想要修改资源的时候，UnhandledExceptionFilter调用VirtualProtect，将资源页上的保护改成PAGE_READWRITE，并返回EXCEPTION_CONTINUE_EXECUTION。

2) 如果你已经调用SetUnhandledExceptionFilter指定了你自己的过滤器，UnhandledExceptionFilter就调用你自己的过滤器函数。如果你自己的过滤器函数返回EXCEPTION_EXECUTE_HANDLER或EXCEPTION_CONTINUE_EXECUTION，UnhandledExceptionFilter就将这个值返回给系统。如果你没有设置你自己的未处理异常过滤器，或者你的未处理异常过滤器返回EXCEPTION_CONTINUE_SEARCH，转到第3步继续处理。

> Windows 98　Windows 98有这样的错误（bug）：如果进程不在调试中，系统只调用程序自己的未处理异常过滤器函数。这样我就不可能调试本章后面介绍的spreadsheet示例程序。

3) 如果你的进程是在调试程序下运行的，就返回EXCEPTION_CONTINUE_SEARCH。你可能会对此感到不解，因系统已经为线程执行了最高层的try或except框架，再往高层已经没有其他的异常过滤器可搜索。当系统看到最高层过滤器返回EXCEPTION_CONTINUE_SERCH时，系统知道要同调试程序联系并告诉调试程序，被调试程序只是有一个未处理异常。作为回答，

调试程序显示一个消息框并允许你调试进程（注意，IsDebuggerPresent函数用来确定一个进程是否正在被调试）。

4) 如果进程中的一个线程以SEM_NOGPFAULTERRORBOX标志为参数调用SetErrorMode，UnhandledExceptionFilter就返回EXCEPTION_EXECUTE_HANDLER。

5) 如果进程在一个作业（job）里并且作业的限制信息设定了JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION标志，则UnhandledExceptionFilter返回EXCEPTION_EXECUTE_HANDLER。

Windows 98 Windows 98不支持作业，所以这一步略去。

6) UnhandledExceptionFilter查阅注册表并取出Auto值。如果这个值是1，跳到第7步。如果这个值是0，向用户显示一个消息框。这个消息框指出引发了什么异常。如果注册表子关键字也包含这个Debugger值，消息框有OK按钮和Cancel按钮。如果注册表子关键字没有Debugger值，消息框只包含OK按钮。如果用户点击OK按钮，UnhandledExceptionFilter返回EXCEPTION_EXECUTE_HANDLER。如果Cancel按钮可用并且用户按了这个按钮，转到第7步继续处理。

Windows 98 在Windows 98中，这些值不是保存在注册表里，而是保存在Win.ini文件里。

7) UnhandledExceptionFilter现在要产生调试程序。它首先调用CreateEvent建立一个无信号的、手工复位的事件。这个事件的句柄是可继承的。然后它从注册表中取出Debugger值，调用sprintf把它粘贴到进程ID（通过调用GetCurrentProcessID函数得到）和事件句柄里。STARTUPINFO的lpDesktop成员也设置成“Winsta0\\Default”，这样调试程序就出现在交互式桌面上。调用CreateProcess，其中fInheritHandles参数设置成TRUE，这个函数再调用调试程序进程并允许它继承事件对象的句柄。UnhandledExceptionFilter通过以事件句柄为参数调用WaitForSingleObjectEx，等待调试程序初始化。注意这里是用WaitForSingleObjectEx函数而不是WaitForSingleObject函数，所以线程是在可报警状态等待。这样就可以处理线程的任何排队的同步过程调用（APC）。

8) 当调试程序完成初始化时，就设置事件句柄，这将唤醒UnhandledExceptionFilter中的线程。现在进程就在调试程序之下运行，UnhandledExceptionFilter返回EXCEPTION_CONTINUE_SEARCH。注意这就是第3步所发生的事。

25.5 异常与调试程序

Microsoft Visual C++调试程序支持调试异常。当一个进程的线程引起一个异常，操作系统立即通知调试程序（如果挂接了一个调试程序）。这个通知被称作最初机会通知（first-chance notificontion）。正常情况下，调试程序要响应最初机会通知，告诉线程去搜索异常过滤器。如果所有的异常过滤器都返回EXCEPTION_CONTINUE_SEARCH，操作系统再通知调试程序，称为最后机会通知(last-chance notification)。使用这两种通知是为了使软件开发人员对调试异常有更多的控制能力。

使用调试程序的Exceptions对话框（见图25-3）来告诉调试程序如何响应最初机会异常通知。

我们可以看到，对话框由系统定义的全部异常所组成。对话框中显示了异常的32位代码，后面跟着文本描述和调试程序的动作。在上面的窗口中，选择了存取违规(Access Violation)异

常，将它的动作改变成 Stop always。这样，当被调试程序中的一个线程引起一个存取违规时，调试程序会接到其最初机会通知，并显示类似图 25-4所示的消息框。

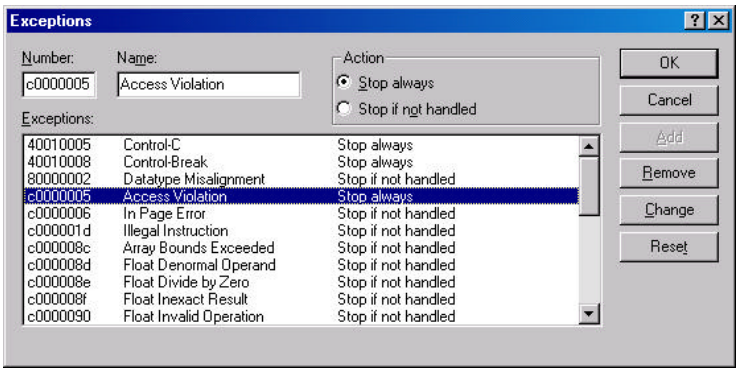

图25-3　Exceptions 对话框

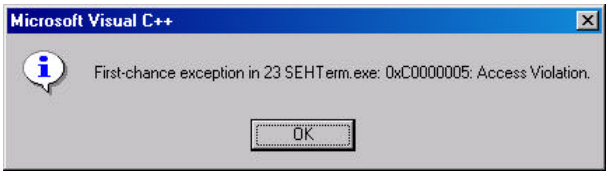

图25-4　“最初机会通知”消息框

在这一步，线程还没有机会去搜索异常过滤器。现在我们可以在代码中设置断点，检查变量，或检查线程的调用栈。现在还没有异常过滤器执行，异常只是发生而已。如果我们现在使用调试程序单步执行代码，会出现图 25-5所示的提示消息框。

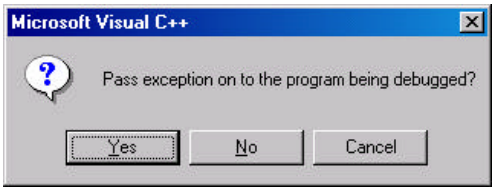

图25-5　提示消息框

点击Cancel按钮，返回到调试程序。点击NO按钮是告诉被调试的线程去重试失败的CPU指令。对大多数异常，重试失败的指令只会再一次引起异常，没有什么用处。但对于一个由函数 RaiseException引起的异常，重试是告诉线程继续执行，就好像异常从未发生。按这种方式继续执行，对于调试 C++程序特别有用：这就好像是一个从未执行过的C++ throw语句。C++异常处理在本章后面讨论。

点击 Yes按钮会使被调试程序的线程去搜索异常过滤器。如果找到一个返回EXCEPTION_EXECUTE_HANDLER或EXCEPTION_CONTINUE_EXECUTION的异常过滤器，则一切都好，线程可以继续执行其代码。如果所有的过滤器都返回 EXCEPTION_C ONTINUE_SEARCH，调试程序收到一个最后机会通知，显示类似图 25-6所示的消息框。

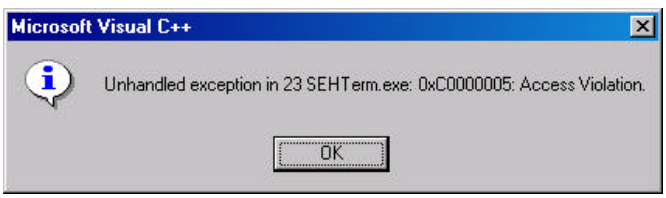

图25-6　调试程序接收到“最后机会通知”时显示的消息框

在这一步，必须要调试这个程序或结束这个程序。

我们讨论了当调试程序的动作设置成 Stop Always时所发生的各种可能情况。但对大多数异常，Stop If Not Handled是默认的动作。如果被调试程序中的一个线程引起一个异常，调试程序会收到最初机会通知。如果调试程序动作设置成 Stop If Not Handled，调试程序只是在调试程序的Output窗口显示一个字符串表示它收到了这个通知（见图 25-7）。

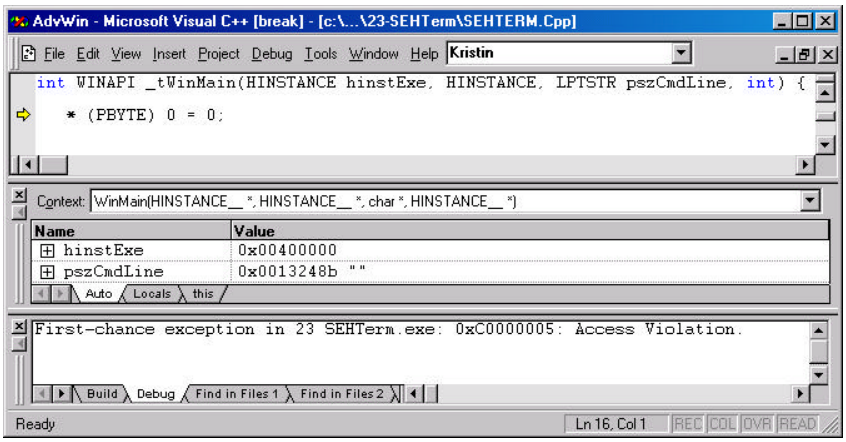

图25-7 调试程序的Output窗口

如果将对存取违规的动作设置成 Stop If Not Handled，调试程序会使线程去搜索异常过滤器。仅当异常没有被处理时，调试程序才会显示图 25-8所示的消息框。

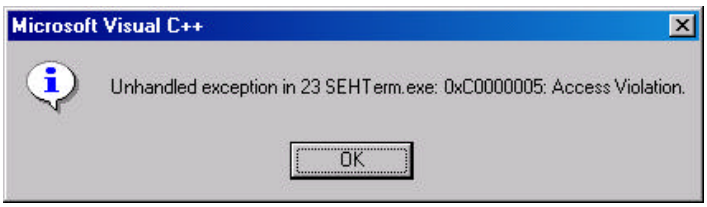

图25-8 异常没有被处理时显示的消息框

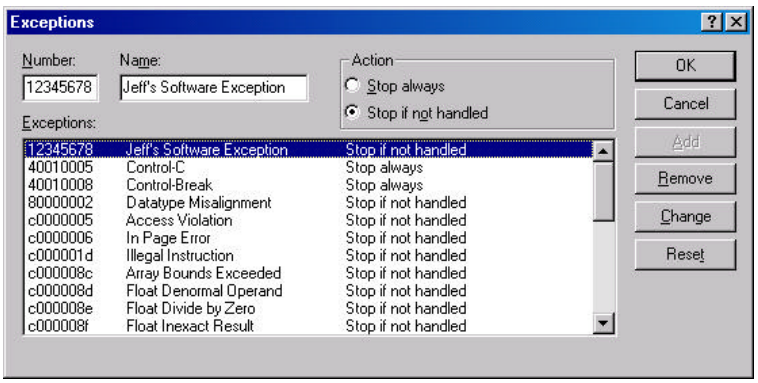

图25-9 Exceptions 窗口

注意 要记住的重要一点是最初机会通知并不指出程序中的问题或错误。实际上，这个通知只是在进程被调试时出现。调试程序只是报告引发了一个异常，但调试程序如

果不显示消息框，处理异常的过滤器和程序会继续运行。一个最后机会通知意味着程序中有问题或错误，必须要进行修改。

在结束这一节之前，需要再说一说调试程序的 Exception对话框。这个对话框完全支持程序员自己定义的任何软件异常。需要做的只是要输入你的软件异常代码编号，并保证唯一性，输入异常的字符串名，还有选择的动作。然后点击 Add按钮，将所定义的异常添加到异常表中。图25-9所示的窗口说明了如何使调试程序知道自己定义的软件异常。

Spreadsheet示例程序

本节后面清单25-1所列的Spreadsheet示例程序说明了如何使用结构化异常处理向一个预留的地址空间范围稀疏地提交存储区。程序的源代码和资源文件在本书所附光盘的“ 25-Spreadsheet ”目录下。当执行Spreadsheet程序时，会出现图25-10所示的对话框。

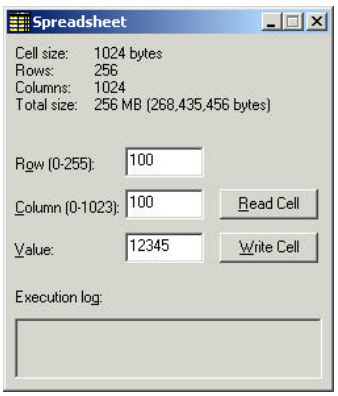

图25-10　Spreadsheet 对话框

在内部，程序为二维的空白表格保留一个范围。空白表格包含256行乘1024列，每个单元长度是1024字节。如果程序要提交内存覆盖整个空白表格，则需要 268 435 456字节，即256MB。为了保留宝贵的内存空间，程序保留一个 256MB的地址空间范围而不提交任何内存来覆盖这个范围。

假定用户试图在第100行第100列的单元中存放数值12345（见图25-10）。当用户点击Write Cell按钮时，程序代码试图对空白表格的这个位置写数据。当然，这个试图写操作要引起一个存取违规。但由于在程序中使用了 SEH，异常过滤器检测到这个试图写操作，在对话框的底部显示“ Violation: Attempting to Write ”消息，对该单元提存储区，并使CPU重新执行引起异常的指令。因存储区已经提交，数值被写入空白表格的单元（见图25-11）。

我们再做另一个实验。试图读第 5行第20列单元中的值。当你试图从这个单元读数据时又引起一个存取违规。对一个试图读操作，异常过滤器不提交存储区，但在对话框中显示“ Violation: Attempting to Read ”消息。程序通过删除对话框中 Value字段中的数值，很自然地从失败的读操作中恢复，见图25-12。

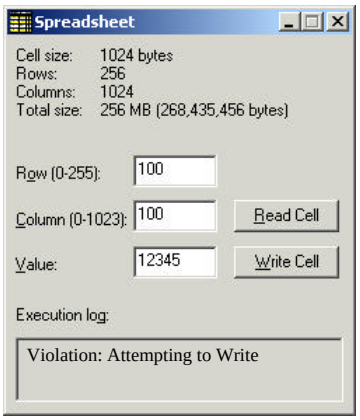

图25-11　Spreadsheet 对话框的下部显示信息

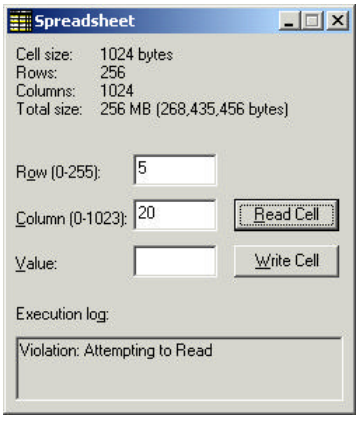

图25-12　删除Value 字段中的数值时的对话框

我们的第3个实验，是读第100行第100列单元的数值。因已经对这个单元提交了存储区，所以没有异常发生，也没有异常过滤器执行（改善了性能）。对话框如图25-13所示。

我们再看第4个实验：试图向第100行第101列单元写数值54321。当你试图这么做时，没有发生异常，这个单元在单元（100，100）相同的内存页上。我们可以看到对话框底部显示消息“No Violation Raised”（见图25-14）。

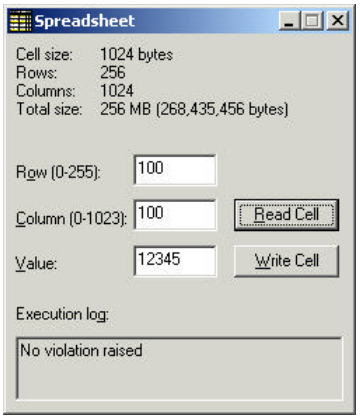

图25-13 第3个实验显示的对话框

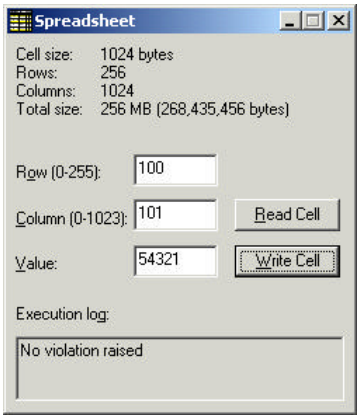

图25-14 第4个实验显示的对话框

笔者在自己的项目中倾向于使用虚拟内存和 SEH。后来决定建立一个模板化的 C++类，CVMArray，它封装了所有困难的工作。这个C++类的源代码在VMArray.h文件中（清单25-1所示的Spreadsheet示例程序的一部分）。可以按两种方式使用CVMArray类。第一，可以建立一个这个类的实例，将数组中元素的最大值传递给构造函数。这个类自动启动一个全进程范围内未处理异常过滤器，每当任何线程中的代码存取虚拟内存数组中的内存地址时，未处理异常过滤器调用VirtualAlloc为新元素提交存储区，并返回EXCEPTION_CONTINUE_EXECUTION。按这种方式使用CVMArray类，可以使你只用几行代码就实现稀疏存储区，并且也不必在源代码中到处使用SEH框架。这种方法的唯一不足是当由于某种原因存储区不能提交时，程序不能以自然的方式恢复。

使用CVMArray类的第二种方法是从中派生出自己的 C++类。使用派生类可以得到基类的优点，但同时也可以添加自己所需要的特性。例如，现在通过过载（ overload）虚拟函数OnAccessViolation更自然地解决了存储区不够的问题。 Spreadsheet示例程序展示了如何在CVMArray的派生类中添加这些特性的方法。

清单25-1 Spreadsheet示例程序

Spreadsheet.cpp

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


#include "..\CmnHdr.h"     /* See Appendix A. */
```

```
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "VMArray.h"


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


HWND g_hwnd;   // Global window handle used for SEH reporting

const int g_nNumRows = 256;
const int g_nNumCols = 1024;
// Declare the contents of a single cell in the spreadsheet
typedef struct {
   DWORD dwValue;
   BYTE  bDummy[1020];
} CELL, *PCELL;


// Declare the data type for an entire spreadsheet
typedef CELL SPREADSHEET[g_nNumRows][g_nNumCols];
typedef SPREADSHEET *PSPREADSHEET;


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


// A spreadsheet is a 2-dimensional array of CELLs
class CVMSpreadsheet : public CVMArray<CELL> {
public:
   CVMSpreadsheet() : CVMArray<CELL>(g_nNumRows * g_nNumCols) {}

private:
   LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
      PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);
};


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


LONG CVMSpreadsheet::OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
   PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

   TCHAR sz[200];
   wsprintf(sz, TEXT("Violation: Attempting to %s"),
      fAttemptedRead ? TEXT("Read") : TEXT("Write"));
   SetDlgItemText(g_hwnd, IDC_LOG, sz);

   LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;
   if (!fAttemptedRead) {

      // Return whatever the base class says to do
      lDisposition = CVMArray<CELL>::OnAccessViolation(pvAddrTouched,
```

```
            fAttemptedRead, pep, fRetryUntilSuccessful);
      }
      return(lDisposition);
   }

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


   // This is the global CVMSpreadsheet object
   static CVMSpreadsheet g_ssObject;

   // Create a global pointer that points to the entire spreadsheet region
   SPREADSHEET& g_ss = * (PSPREADSHEET) (PCELL) g_ssObject;


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


   BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

      chSETDLGICONS(hwnd, IDI_SPREADSHEET);

      g_hwnd = hwnd; // Save for SEH reporting

      // Put default values in the dialog box controls
      Edit_LimitText(GetDlgItem(hwnd, IDC_ROW),    3);
      Edit_LimitText(GetDlgItem(hwnd, IDC_COLUMN), 4);
      Edit_LimitText(GetDlgItem(hwnd, IDC_VALUE),  7);
      SetDlgItemInt(hwnd, IDC_ROW,    100,   FALSE);
      SetDlgItemInt(hwnd, IDC_COLUMN, 100,   FALSE);
      SetDlgItemInt(hwnd, IDC_VALUE,  12345, FALSE);
      return(TRUE);
   }


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


   void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

      int nRow, nCol;

      switch (id) {
         case IDCANCEL:
            EndDialog(hwnd, id);
            break;
         case IDC_ROW:
            // User modified the row, update the UI
            nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
            EnableWindow(GetDlgItem(hwnd, IDC_READCELL),
               chINRANGE(0, nRow, g_nNumRows - 1));
            EnableWindow(GetDlgItem(hwnd, IDC_WRITECELL),
               chINRANGE(0, nRow, g_nNumRows - 1));
            break;
```

```
         case IDC_COLUMN:
            // User modified the column, update the UI
            nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);
            EnableWindow(GetDlgItem(hwnd, IDC_READCELL),
               chINRANGE(0, nCol, g_nNumCols - 1));
            EnableWindow(GetDlgItem(hwnd, IDC_WRITECELL),
               chINRANGE(0, nCol, g_nNumCols - 1));
            break;

         case IDC_READCELL:
            // Try to read a value from the user's selected cell
            SetDlgItemText(g_hwnd, IDC_LOG, TEXT("No violation raised"));
            nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
            nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);
            __try {
               SetDlgItemInt(hwnd, IDC_VALUE, g_ss[nRow][nCol].dwValue, FALSE);
            }
            __except (
               g_ssObject.ExceptionFilter(GetExceptionInformation(), FALSE)) {

               // The cell is not backed by storage, the cell contains nothing.
               SetDlgItemText(hwnd, IDC_VALUE, TEXT(""));
            }
            break;

         case IDC_WRITECELL:
            // Try to read a value from the user's selected cell
            SetDlgItemText(g_hwnd, IDC_LOG, TEXT("No violation raised"));
            nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
            nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);

            // If the cell is not backed by storage, an access violation is
            // raised causing storage to automatically be committed.
            g_ss[nRow][nCol].dwValue =
               GetDlgItemInt(hwnd, IDC_VALUE, NULL, FALSE);
            break;
      }
   }

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

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////
```

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

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SPREADSHEET), NULL, Dlg_Proc);
   return(0);
}

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

Spreadsheet.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
```

```
    "\0"
END

#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_SPREADSHEET DIALOG DISCARDABLE  18, 18, 164, 165
STYLE DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Spreadsheet"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Cell size:\nRows:\nColumns:\nTotal size:",IDC_STATIC,4,
                    4,36,36
    LTEXT           "1024 bytes\n256\n1024\n256 MB (268,435,456 bytes)",
                    IDC_STATIC,44,4,104,36
    LTEXT           "R&ow (0-255):",IDC_STATIC,4,56,42,8
    EDITTEXT        IDC_ROW,60,52,40,14,ES_AUTOHSCROLL | ES_NUMBER
    LTEXT           "&Column (0-1023):",IDC_STATIC,4,76,54,8
    EDITTEXT        IDC_COLUMN,60,72,40,14,ES_AUTOHSCROLL | ES_NUMBER
    PUSHBUTTON      "&Read Cell",IDC_READCELL,108,72,50,14
    LTEXT           "&Value:",IDC_STATIC,4,96,21,8
    EDITTEXT        IDC_VALUE,60,92,40,14,ES_AUTOHSCROLL | ES_NUMBER
    PUSHBUTTON      "&Write Cell",IDC_WRITECELL,108,92,50,14
    LTEXT           "Execution lo&g:",IDC_STATIC,4,118,48,8
    EDITTEXT        IDC_LOG,4,132,156,28,ES_MULTILINE | ES_AUTOHSCROLL |
                    ES_READONLY
END


////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SPREADSHEET         ICON    DISCARDABLE     "Spreadsheet.Ico"
#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////////////////////
```

```
#endif    // not APSTUDIO_INVOKED
```

VMArray.h

```
/******************************************************************************
Module:  VMArray.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once


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


// NOTE: This C++ class is not thread safe. You cannot have multiple threads
// creating and destroying objects of this class at the same time.

// However, once created, multiple threads can access different CVMArray
// objects simultaneously and you can have multiple threads accessing a single
// CVMArray object if you manually synchronize access to the object yourself.


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


template <class TYPE>
class CVMArray {
public:
   // Reserves sparse array of elements
   CVMArray(DWORD dwReserveElements);

   // Frees sparse array of elements
   virtual ~CVMArray();

   // Allows accessing an element in the array
   operator TYPE*()             { return(m_pArray); }
   operator const TYPE*() const { return(m_pArray); }

   // Can be called for fine-tuned handling if commit fails
   LONG ExceptionFilter(PEXCEPTION_POINTERS pep,
      BOOL fRetryUntilSuccessful = FALSE);

protected:
   // Override this to fine-tune handling of access violation
   virtual LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
      PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);

private:
   static CVMArray* sm_pHead;    // Address of first object
   CVMArray* m_pNext;            // Address of next  object

   TYPE* m_pArray;               // Pointer to reserved region array
   DWORD m_cbReserve;            // Size of reserved region array (in bytes)
```

```
private:
   // Address of previous unhandled exception filter
   static PTOP_LEVEL_EXCEPTION_FILTER sm_pfnUnhandledExceptionFilterPrev;

   // Our global unhandled exception filter for instances of this class
   static LONG WINAPI UnhandledExceptionFilter(PEXCEPTION_POINTERS pep);
};


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


// The head of the linked-list of objects
template <class TYPE>
CVMArray<TYPE>* CVMArray<TYPE>::sm_pHead = NULL;

// Address of previous unhandled exception filter
template <class TYPE>
PTOP_LEVEL_EXCEPTION_FILTER CVMArray<TYPE>::sm_pfnUnhandledExceptionFilterPrev;


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


template <class TYPE>
CVMArray<TYPE>::CVMArray(DWORD dwReserveElements) {

   if (sm_pHead == NULL) {
      // Install our global unhandled exception filter when
      // creating the first instance of the class.
      sm_pfnUnhandledExceptionFilterPrev =
         SetUnhandledExceptionFilter(UnhandledExceptionFilter);
   }
    m_pNext = sm_pHead;  // The next node was at the head
    sm_pHead = this;     // This node is now at the head

    m_cbReserve = sizeof(TYPE) * dwReserveElements;

    // Reserve a region for the entire array
    m_pArray = (TYPE*) VirtualAlloc(NULL, m_cbReserve,
       MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
    chASSERT(m_pArray != NULL);
}


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


template <class TYPE>
CVMArray<TYPE>::~CVMArray() {

   // Free the array's region (decommitting all storage within it)
   VirtualFree(m_pArray, 0, MEM_RELEASE);

   // Remove this object from the linked list
```

```
   CVMArray* p = sm_pHead;
   if (p == this) {     // Removing the head node
      sm_pHead = p->m_pNext;
   } else {

      BOOL fFound = FALSE;

      // Walk list from head and fix pointers
      for (; !fFound && (p->m_pNext != NULL); p = p->m_pNext) {
         if (p->m_pNext == this) {
            // Make the node that points to us point to the next node
            p->m_pNext = p->m_pNext->m_pNext;
            break;
         }
      }
      chASSERT(fFound);
   }
}


//////////////////////////////////////////////////////////////////////////////
 // Default handling of access violations attempts to commit storage
 template <class TYPE>
 LONG CVMArray<TYPE>::OnAccessViolation(PVOID pvAddrTouched,
    BOOL fAttemptedRead, PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

    BOOL fCommittedStorage = FALSE;  // Assume committing storage fails

    do {
       // Attempt to commit storage
       fCommittedStorage = (NULL != VirtualAlloc(pvAddrTouched,
          sizeof(TYPE), MEM_COMMIT, PAGE_READWRITE));

       // If storage could not be committed and we're supposed to keep trying
       // until we succeed, prompt user to free storage
       if (!fCommittedStorage && fRetryUntilSuccessful) {
          MessageBox(NULL,
             TEXT("Please close some other applications and Press OK."),
             TEXT("Insufficient Memory Available"), MB_ICONWARNING | MB_OK);
       }
    } while (!fCommittedStorage && fRetryUntilSuccessful);

    // If storage committed, try again. If not, execute the handler
    return(fCommittedStorage
       ? EXCEPTION_CONTINUE_EXECUTION : EXCEPTION_EXECUTE_HANDLER);
}

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

// The filter associated with a single CVMArray object
template <class TYPE>
LONG CVMArray<TYPE>::ExceptionFilter(PEXCEPTION_POINTERS pep,
   BOOL fRetryUntilSuccessful) {

   // Default to trying another filter (safest thing to do)
```

```
   LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

   // We only fix access violations
   if (pep->ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION)
      return(lDisposition);

   // Get address of attempted access and get attempted read or write
   PVOID pvAddrTouched = (PVOID) pep->ExceptionRecord->ExceptionInformation[1];
   BOOL fAttemptedRead = (pep->ExceptionRecord->ExceptionInformation[0] == 0);

   // Is attempted access within this VMArray's reserved region?
   if ((m_pArray <= pvAddrTouched) &&
       (pvAddrTouched < ((PBYTE) m_pArray + m_cbReserve))) {

      // Access is in this array, try to fix the problem
      lDisposition = OnAccessViolation(pvAddrTouched, fAttemptedRead,
         pep, fRetryUntilSuccessful);
   }

   return(lDisposition);
}

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

// The filter associated with all CVMArray objects
template <class TYPE>
LONG WINAPI CVMArray<TYPE>::UnhandledExceptionFilter(PEXCEPTION_POINTERS pep) {

   // Default to trying another filter (safest thing to do)
   LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

   // We only fix access violations
   if (pep->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {

      // Walk all the nodes in the linked-list
      for (CVMArray* p = sm_pHead; p != NULL; p = p->m_pNext) {

         // Ask this node if it can fix the problem.
         // NOTE: The problem MUST be fixed or the process will be terminated!
         lDisposition = p->ExceptionFilter(pep, TRUE);

         // If we found the node and it fixed the problem, stop the loop
         if (lDisposition != EXCEPTION_CONTINUE_SEARCH)
            break;
      }
   }

   // If no node fixed the problem, try the previous exception filter
   if (lDisposition == EXCEPTION_CONTINUE_SEARCH)
      lDisposition = sm_pfnUnhandledExceptionFilterPrev(pep);

   return(lDisposition);
}

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

25.6 C++异常与结构性异常的对比

笔者常碰到开发人员询问，在开发程序时应该使用结构化异常处理，还是使用 C++异常处理？本节对此作出回答。

首先要提醒读者，SEH是可用于任何编程语言的操作系统设施，而异常处理只能用于编写C++代码。如果你在编写C++程序，你应该使用C++异常处理而不是结构化异常处理。理由是C++异常处理是语言的一部分，编译器知道 C++类对象是什么。也就是说编译器能够自动生成代码来调用C++对象析构函数，保证对象的清除。

但是也应该知道，Microsoft Visual C++编译器已经利用操作系统的结构化异常处理实现了C++异常处理。所以当你建立一个 C++ try块时，编译器就生成一个 SEH__try块。一个C++ catch测试变成一个SEH异常过滤器，并且catch中的代码变成SEH__except块中的代码。实际上，当你写一条C++ throw语句时，编译器就生成一个对 Windows的RaiseException函数的调用。用于throw语句的变量传递给 RaiseException作为附加的参数。

下面的代码段可以使上面的叙述更清楚一些。左边的函数使用 C++异常处理，右边的函数说明了C++编译器如何生成等价的结构化异常处理。

```
void ChunkyFunky() {
   try {
      // Try body
      .
      .
      .
      throw 5;

   }
   catch (int x) {

      // Catch body
      .
      .
      .
   }
   .
   .
   .
}
```

```
void ChunkyFunky() {
   __try {
      // Try body
      .
      .
      .
      RaiseException(Code=0xE06D7363,
         Flag=EXCEPTION_NONCONTINUABLE,
         Args=5);
   }
   __except ((ArgType == Integer) ?
      EXCEPTION_EXECUTE_HANDLER :
      EXCEPTION_CONTINUE_SEARCH) {
      // Catch body
      .
      .
      .
   }
   .
   .
   .
}
```

你可能注意到上面代码中一些有趣的细节。首先，函数 RaiseException的调用使用了异常代码0xE06D7363。这是由Visual C++的开发人员选择的软件异常代码，在引发C++异常时使用。事实上你可以验证这一点，方法是打开调试程序的 Exceptions对话框，滚动到异常列表的底部，见图25-15。

当引发了C++异常时，总要使用EXCEPTION_NONCONTINUEABLE标志。C++异常不能被重新执行。对于诊断C++异常的过滤器，如果返回EXCEPTION_CONTINUE_EXECUTION，那将是个错误。实际上，我们看一看上面程序段右边函数中的 __except过滤器，就会发现它只能计算成EXCEPTION_EXECUTE_HANDLER或EXCEPTION_CONTINUE_SEARCH。

传递到RaiseException的其余参数是用来作为一种机制，用于实际引发指定的变量。被引发的变量信息是如何传递到 RaiseException的，这一点没有公开。但不难想像编译器的开发人员可以实现这一点。

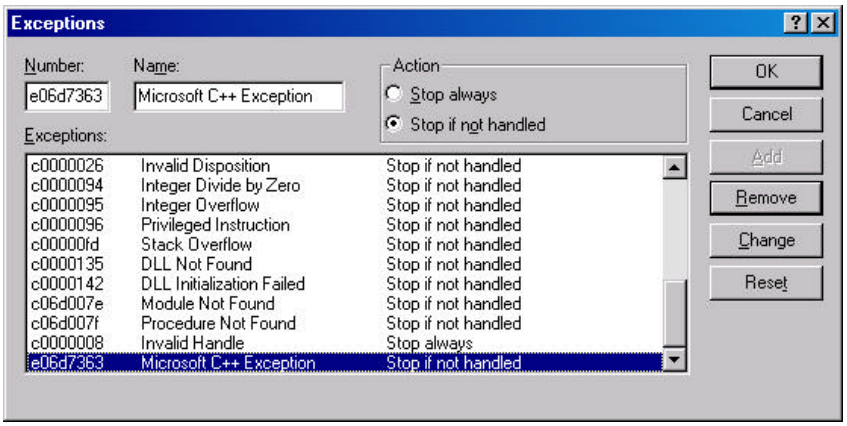

图25-15　Exceptions 对话框

最后要指出的是__except过滤器。这个过滤器的用途是将throw变量的数据类型同用于C++ catch语句的变量类型相比较。如果数据类型相同，过滤器返回 EXCEPTION_EXECUTE_HANDLER，导致catch块（__except块）中的语句执行。如果数据类型不同，过滤器返回EXCEPTION_CONTNUE_SEARCH，导致catch过滤器上溯要计算的调用树。

注意　由于C++异常在内部是由结构性异常实现的，所以可以在一个程序中使用两种机制。例如，当引起存取违规时，我乐意使用虚拟内存来提交存储区。而 C++语言完全不支持这种类型的再恢复异常处理（resumptive exception handling）。我可以在我的代码的某些部分使用结构化处异常处理。在这些部分需要利用这种结构的优点，可使我自己的__except过滤器返回EXCEPTION_CONTINUE_EXECUTION。对于代码的其他部分，如果不要求这种再恢复异常处理，我还是使用C++异常处理。

用C++来捕获结构性异常

正常情况下，C++异常处理不能使程序从硬件异常中恢复，硬件违规就是存取违规或零作除数这种异常。但微软已经对其编译器增加了这种支持能力。例如，下面的代码可以防止进程不正常地结束：

```
void main() {
   try {
      * (PBYTE) 0 = 0;       // Access violation
   }
   catch (...) {
      // This code handles the access-violation exception
   }
   // The process is terminating normally
}
```

这很棒，它可以使程序从硬件异常中很自然地恢复。但是如果 catch的异常描述能够区分不同的异常代码，那就更棒了。例如，如果能够按下面的方式编写代码，就更好了。

```
void Functastic() {

   try {
      * (PBYTE) 0 = 0;       // Access violation
```

```
            int x = 0;
            x = 5 / x;              // Division by zero
         }
         catch (StructuredException) {
            switch (StructuredExceptionCode) {
               case EXCEPTION_ACCESS_VIOLATION:
                  // This code handles an access-violation exception
                  break;

               case EXCEPTION_INT_DIVIDE_BY_ZERO:
                  // This code handles a division-by-zero exception
                  break;

               default:
                  // We don't handle any other exceptions
                  throw;   // Maybe another catch is looking for this
                  break;   // Never executes
            }
         }
      }
```

使我们高兴的是，Visuad C++有一种机制可以实现这一点。你需要做的是建立你自己的C++类，在代码中用来标识结构性异常。这里是一个例子：

```
#include <eh.h>                 // For _set_se_translator
.
.
.
class CSE {
public:
   // Call this function for each thread.
   static void MapSEtoCE() { _set_se_translator(TranslateSEtoCE); }

   operator DWORD() { return(m_er.ExceptionCode); }

private:
   CSE(PEXCEPTION_POINTERS pep) {
      m_er      = *pep->ExceptionRecord;
      m_context = *pep->ContextRecord;
   }

   static void _cdecl TranslateSEtoCE(UINT dwEC,
      PEXCEPTION_POINTERS pep) {
      throw CSE(pep);
   }

private:
   EXCEPTION_RECORD m_er;       // CPU independent exception information
   CONTEXT          m_context; // CPU dependent exception information
};
```

在每个线程的进入点函数里，调用静态成员函数MapSEtoCE。这个函数调用C运行时函数_set_se_translator，并传递CSE类的TranslateSEtoCE函数的地址作为参数。通过调用_set_se_translator，告诉C++运行时系统，在结构性异常发生时调用TranslateSEtoCE函数。这个函数构造一个CSE类对象并初始化两个数据成员以包含有关异常的CPU独立和CPU依赖的信息。在构造了CSE对象之后，它就被引发，如同任何正常的变量可被引发一样。现在你的C++

代码可以通过捕获一个这类的变量来处理结构性异常。

下面是如何捕获这个C++对象的例子。

```
void Functastic() {

   CSE::MapSEtoCE(); // Must be called before any exceptions are raised

   try {
      * (PBYTE) 0 = 0;       // Access violation

      int x = 0;
      x = 5 / x;             // Division by zero
   }
   catch (CSE se) {
      switch (se) {    // Calls the operator DWORD() member function
         case EXCEPTION_ACCESS_VIOLATION:
            // This code handles an access-violation exception
            break;

         case EXCEPTION_INT_DIVIDE_BY_ZERO:
            // This code handles a division-by-zero exception
            break;

         default:
            // We don't handle any other exceptions
            throw;   // Maybe another catch is looking for this
            break;   // Never executes
      }
   }
}
```