深入研究 Win32 结构化异常处理

本文关键字：SEH, Windows, VisualC

摘要

就像人们常说的那样，Win32 结构化异常处理（SEH）是一个操作系统提供的服务。你能找到的所有关于 SEH 的文档讲的都是针对某个特定编译器的、建立在操作系统层之上的封装库。我将从 SEH 的最基本概念讲起。

Matt Pietrek 著
董岩 译
Victor 转载自 Xfocus 并整理

在所有 Win32 操作系统提供的机制中，使用最广泛的未公开的机制恐怕就要数结构化异常处理（structured exception handling，SEH）了。一提到结构化异常处理，可能就会令人想起 _try、_finally 和 _except 之类的词儿。在任何一本不错的 Win32 书中都会有对 SEH 详细的介绍。甚至连 Win32 SDK 里都对使用 _try、_finally 和 _except 进行结构化异常处理作了完整的介绍。既然有这么多地放都提到了 SEH，那我为什么还要说它是未公开的呢？本质上讲，Win32 结构化异常处理是操作系统提供的一种服务。编译器的运行时库对这种服务操作系统实现进行了封装，而所有能找到的介绍 SEH 的文档讲的都是针对某一特定编译器的运行时库。关键字 _try、_finally 和 _except 并没有什么神秘的。微软的 OS 和编译器定义了这些关键字以及它们的行为。其它的 C++ 编译器厂商也只需要尊从它们定好的语义就行了。在编译器的 SEH 层减少了直接使用纯操作系统的 SEH 所带来的危害的同时，也将纯操作系统的 SEH 从大家的面前隐藏了起来。

我收到过大量的电子邮件说他们都需要实现编译器级的 SEH 但却找不到公开的文档。本来，我可以指着 Visual C++ 和 Borlang C++ 的运行时库的源代码说看一下它们就行了。但是，不知道是什么原因，编译器级的 SEH 仍是个天大的秘密。微软和 Borland 都没有提供 SEH 最内层的源代码。

在本文中，我会从最基本的概念上讲解结构化异常处理。在讲解的时候，我会将操作系统所提供的与编译器代码生成和运行时库支持的分离开来。当深入关键性操作系统程序的代码时，我基于的都是 Intel 版的 Windows NT 4.0。然而。我所讲的大部分内容同样适用于其它的处理器。

我会避免提及实际的 C++ 的异常处理，C++ 下用的是 catch() 而不是 _except。其实，真正的 C++ 异常处理的实现方式和我所讲的方式也是极为相似的。但是，真正 C++ 异常处理特有的复杂性会影响到我这里所讲的概念。对于深挖那些晦涩的 .H 和 .INC 文件并拼凑出 Win32 SEH 的相关代码，最好的一个信息来源就是 IBM OS/2 的头文件（特别是 BSEXCPT.H）。这对有相关经验的人并没什么可希奇的，这里讲的 SEH 机制在微软开发 OS/2 时就定义了。因此，Win32 的 SEH 与 OS/2 的极为相似。

SEH in the Buff

若将 SEH 的细节都放到一起讨论，任务实在艰巨，因此，我会从简单的开始，一层一层往深里讲。如果之前从未使用过结构化异常处理，则正好心无杂念。若是用过，那就要努力将 _try、GetExceptionCode 和 EXCEPTION_EXECUTE_HANDLER 从脑子中扫出，假装这是一个全新的概念。Are you ready？Good。

当线程发生异常时，操作系统会将这个异常通知给用户使用户能够得知它的发生。更特别的是，当线程发生异常时，操作系统会调用用户定义的回调函数。这个回调函数想做什么就能做什么。例如，它可以修正引起异常的程序，也可以播放一段 .WAV 文件。无论回调函数干什么，函数最后的动作都是返回一个值告诉系统下面该干些什么（这样说并不严格，但目前可以认为是这样）。既然在用户代码引起异常后，操作系统会回调用户的代码，那这个回调函数又是什么样的呢？换句话说，关于异常都需要知道哪些信息呢？其实无所谓，因为 Win32 已经定义好了。异常的回调函数的样子如下：

```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
        struct _EXCEPTION_RECORD *ExceptionRecord,
        void * EstablisherFrame,
        struct _CONTEXT *ContextRecord,
        void * DispatcherContext
);
```

这个函数原型来自标准 Win32 头文件 EXCPT.H，初看上去让人有点眼晕。如果慢慢看的话，似乎情况还没那么严重。对于初学者来说，大可以忽略返回值的类型 （EXCEPTION_DISPOSITION）。所需知道的就是这个函数叫 _except_handler，需要四个参数。

第一个参数是一个指向 EXCEPTION_RECORD 的指针。这个结构体定义在 WINNT.H 中，定义如下：

```
typedef struct _EXCEPTION_RECORD {
        DWORD ExceptionCode;
        DWORD ExceptionFlags;
        struct _EXCEPTION_RECORD *ExceptionRecord;
        PVOID ExceptionAddress;
        DWORD NumberParameters;
        DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

参数 ExceptionCode 是操作系统分配给异常的号。在 WINNT.H 文件中查找开头为“STATUS_”的宏就能找到一大堆这样的异常代号。例如，大家熟知的 STATUS_ACCESS_VIOLATION 的代号就是 0xC0000005。更为完整的异常代号可以从 Windows NT DDK 中的

NTSTATUS.H 文件里找到。EXCEPTION_RECORD 结构体的第四个元素是异常发生处的地址。其余的 EXCEPTION_RECORD 域目前都可以忽略掉。_except_handler 函数的第二个参数是一个指向 establisher frame 结构体的指针。在 SEH 里这可是个重要的参数，不过现在先不用管它。第三个参数是一个指向 CONTEXT 结构体的指针。CONTEXT 结构体定义在 WINNT.H 文件中，它保存着某一线程的寄存器的值。图 1 即为 CONTEXT 结构体的域。

图 1：CONTEXT 结构

```
typedef struct _CONTEXT
{
        DWORD   ContextFlags;
        DWORD   Dr0;
        DWORD   Dr1;
        DWORD   Dr2;
        DWORD   Dr3;
        DWORD   Dr6;
        DWORD   Dr7;
        FLOATING_SAVE_AREA FloatSave;
        DWORD   SegGs;
        DWORD   SegFs;
        DWORD   SegEs;
        DWORD   SegDs;
        DWORD   Edi;
        DWORD   Esi;
        DWORD   Ebx;
        DWORD   Edx;
        DWORD   Ecx;
        DWORD   Eax;
        DWORD   Ebp;
        DWORD   Eip;
        DWORD   SegCs;
        DWORD   EFlags;
        DWORD   Esp;
        DWORD   SegSs;
} CONTEXT;
```

当用于 SEH 时，CONTEXT 结构体保存着发生异常时各寄存器的值。无独有偶，GetThreadContext 和 SetThreadContext 使用的也是相同的 CONTEXT 结构体。第四个也是最后的一个参数叫做 DispatcherContext，现在先不去管它。

简单总结一下，当发生异常时会调用一个回调函数。这个回调函数需要四个参数，其中三个都是结构体指针。在这些结构体中，有些域重要，有些并不重要。关键的问题是 _except_handler 回调函数收到了大量的信息，比如异常的类型和发生的位置。异常回调函数需要使用这些信息来决定所采取的行动。

我很想现在就给出一个样例程序来说明 _except_handler，只是仍有一些东西需要解释，即当异常发生时操作系统是如何知道在那里调用回调函数呢？答案在另一个叫 EXCEPTION_REGISTRATION 的结构体中。本文通篇都能见到这个结构体，因此对这部分还是不要囫囵吞枣为好。唯一能找到 EXCEPTION_REGISTRATION 正式定义的地方就是 Visual C++ 运行时库源代码中的 EXSUP.INC 文件：

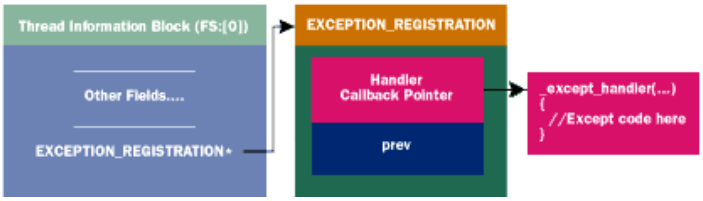
```
_EXCEPTION_REGISTRATION struc
        prev    dd              ?
        handler dd              ?
_EXCEPTION_REGISTRATION ends
```

可以看到，在 WINNT.H 的 NT_TIB 结构体定义中，这个结构体被称为 _EXCEPTION_REGISTRATION_RECORD。然而 _EXCEPTION_REGISTRATION_RECORD 的定义是没有的，因此我所能用的只能是 EXSUP.INC 中的汇编语言的 struc 定义。对于我前面提到的 SEH 的未公开，这就是一例。

不管怎样，我们回到目前的问题上来。当异常发生时，OS 是如何知道调用位置的呢？EXCEPTION_REGISTRATION 结构体有两个域，第一个先不用管。第二个域，handler，为一个指向 _except_ handler 回调函数的指针。有点儿接近答案了，但是还有个问题就是，OS 从哪里能找到这个 EXCEPTION_REGISTRATION 结构体呢？

为了回答这个问题，需要记住结构化异常处理是以线程为基础的。也就是说，每一个线程都有自己的异常处理回调函数。在 1996 年 5 月的专栏中，我讲了一个关键的 Win32 数据结构，线程信息块（TEB 或 TIB）。这个结构体中有一个域对于 Windows NT，Windows 95，Win32s 和 OS/2 都是相同的。TIB 中的第一个 DWORD 是一个指向线程的 EXCEPTION_REGISTRATION 结构体的指针。在 Intel 的 Win32 平台上，FS 寄存器永远指向当前的 TIB，因此，在 FS:[0] 就可以找到指向 EXCEPTION_REGISTRATION 结构体的指针。答案出来了！当异常发生时，系统察看出错线程的 TIB 并取回一个指向 EXCEPTION_REGISTRATION 结构体的指针，从而得到一个指向 _except_handler 回调函数的指针。现在操作系统已经有足够的信息来调用 _except_handler 函数了，见图 2。

图 2：_except_handler 函数



把目前这一小点儿东西凑到一起，我写了一个小程序来演示所讲到的这个非常简单的 OS 级的结构化异常处理。图 3 所示的就是 MYSEH.CPP，它只有两个函数。main 函数使用了三个内嵌的 ASM 块。第一个块使用两条 PUSH 指令（“PUSH handler”和“PUSH FS:[0]”）在堆栈上构建了一个 EXCEPTION_REGISTRATION 结构体。PUSH FS:[0] 将 FS:[0] 的上一个值保存为结构体的一部分，但是目前并不重要。重要的是堆栈上有一个 8 字节的 EXCEPTION_REGISTRATION 结构体。下一条指令（MOV FS:[0],ESP）将线程信息块的第一个 DWORD 指向新的 EXCEPTION_REGISTRATION 结构体。

图 3：MYSEH.cpp

```
//==============================================
// MYSEH - Matt Pietrek 1997
// Microsoft Systems Journal, January 1997
// FILE: MYSEH.CPP
// To compile: CL MYSEH.CPP
//==============================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

DWORD  scratch;

EXCEPTION_DISPOSITION
__cdecl
_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    unsigned i;

    // Indicate that we made it to our exception handler
    printf( "Hello from an exception handler\n" );

    // Change EAX in the context record so that it points to someplace
    // where we can successfully write
    ContextRecord->Eax = (DWORD)&scratch;

    // Tell the OS to restart the faulting instruction
    return ExceptionContinueExecution;
}

int main()
{
    DWORD handler = (DWORD)_except_handler;

    __asm
    {                           // Build EXCEPTION_REGISTRATION record:
        push    handler         // Address of handler function
        push    FS:[0]          // Address of previous handler
        mov     FS:[0],ESP      // Install new EXECEPTION_REGISTRATION
    }

    __asm
    {
```

```
        mov     eax,0           // Zero out EAX
        mov     [eax], 1        // Write to EAX to deliberately cause a fault
    }

    printf( "After writing!\n" );

    __asm
    {                           // Remove our EXCEPTION_REGISTRATION record
        mov     eax,[ESP]       // Get pointer to previous record
        mov     FS:[0], EAX     // Install previous record
        add     esp, 8          // Clean our EXCEPTION_REGISTRATION off stack
    }

    return 0;
}
```

在堆栈上构建 EXCEPTION_REGISTRATION 结构体而不是使用全局变量是有原因的。当使用编译器的 _try/_except 语义时，编译器也会在堆栈上构建 EXCEPTION_REGISTRATION 结构体。我只是要说明使用 _try/_except 后编译器所做的最起码的工作。回到 main 函数，下一个 __asm 块清零了 EAX 寄存器（MOV EAX,0）然后将寄存器的值作为内存地址，而下一条指令就向这个地址进行写入（MOV [EAX],1），这就引发了异常。最后的 __asm 块移除这个简单的异常处理：首先恢复以前的 FS:[0] 的内容，然后从堆栈中弹出 EXCEPTION_REGISTRATION 记录（ADD ESP,8）。

现在假设正在运行 MYSEH.EXE，看一下程序的执行情况。MOV [EAX],1 指令的执行引发了一个 access violation。系统察看 TIB 的 FS:[0] 并找到指向 EXCEPTION_REGISTRATION 结构体的指针。结构体中有一个指向 MYSEH.CPP 文件中的 _except_handler 函数的指针。系统将所需的四个参数入栈并调用 _except_handler 函数。一进入 _except_handler，代码首先用一条 printf 语句打印“Yo! I made it here!”。然后，_except_handler 修复引起异常的问题。问题在于 EAX 指向了不可写内存的地址（地址 0）。所做的修复就是修改 CONTEXT 中 EAX 的值，使其指向一个可写的内存单元。在这个简单的程序里，一个 DWORD 类型变量（scratch）就是用于此目的的。_except_handler 函数的最后的动作就是返回 ExceptionContinueExecution 类型的值，这个结构体定义在标准的 EXCPT.H 文件中。

当操作系统看到所返回的 ExceptionContinueExecution 时，就认为问题已被解决并重新执行引起异常的指令。因为我的 _except_handler 函数修改了 EAX 寄存器使其指向了有效的内存，MOV EAX,1 就再一次执行，main 函数正常继续。并不很复杂，不是吗？

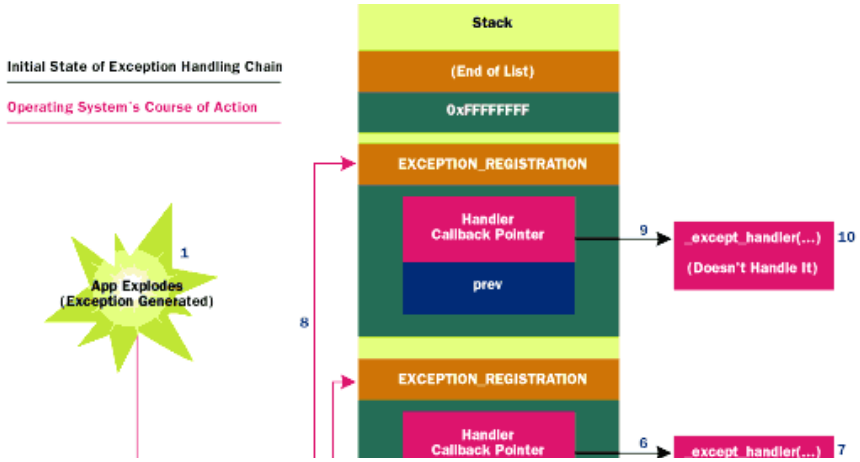
Moving In a Little Deeper

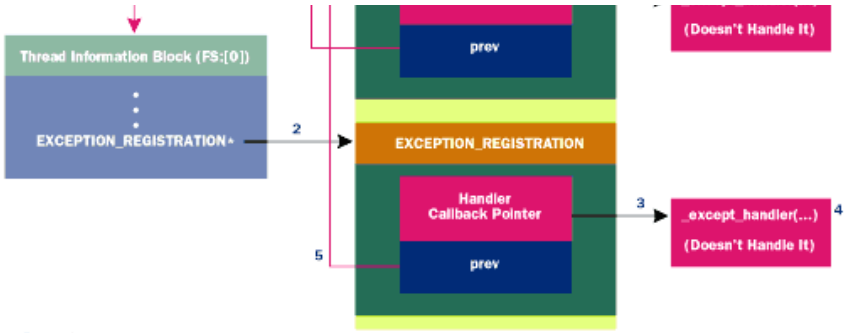
有了这个最简单的情形，我们再回来填补几个空白。尽管异常回调如此伟大，但并不完美。对于任意大小的程序，编写一个函数来处理程序中可能发生的所有异常，那这个函数恐怕会是一团糟。更为可行的情形是能有多个异常处理函数，每一个函数都用于程序的某一特定的部分。操作系统提供了这个功能。

还记得系统查找异常处理回调函数所用的 EXCEPTION_REGISTRATION 结构体吧？此结构体的第一个参数，就是我前面忽略的那个，它叫做 prev。它确实是指向另一个 EXCEPTION_REGISTRATION 结构体的指针。这个第二个 EXCEPTION_REGISTRATION 结构体可以有一个完全不同的处理函数。而且它的 prev 域还可以指向第三个 EXCEPTION_REGISTRATION 结构体，依次类推。简单讲，就是一个 EXCEPTION_REGISTRATION 结构体组成的链表。此链表的表头总是由线程信息块的第一个 DWORD （Intel 机器上的 FS:[0]）所指向。

操作系统用这个 EXCEPTION_REGISTRATION 结构体链表做什么？当异常发生时，系统遍历此链表并查找回调函数与异常相符的 EXCEPTION_REGISTRATION。对于 MYSEH.CPP 来说，回调函数返回 ExceptionContinueExecution 型的值，与异常相符合。回调函数也可能不适合所发生的异常，这时系统就移向链表中下一个 EXCEPTION_REGISTRATION 结构体并询问异常回调是否要处理此异常。图 4 所示即为此过程。

图 4：查找一个处理异常的结构体





1. Exception occurs
2. Operating system uses the thread information block to find the head pf the exception handler chain
3. Operating system calls the first exception handling callback function in the list
4. First callback declines to handle the exception
5. Operating system moves on to the previously registered handler
6. Operating system calls the second exception handling callback function
7. Second callback declines to handle exception
8. Operating system moves on to the previously registered handler
9. Operating system calls the third exception handling callback function
10.Third callback handles the exception

一旦系统找到了处理此异常的回调函数就停止对 EXCEPTION_REGISTRATION 链表的遍历。我给出了一个异常回调不能处理异常的例子，见图 5 的 MYSEH2.CPP。

图 5：MYSEH2.cpp

```
//==================================================
// MYSEH2 - Matt Pietrek 1997
// Microsoft Systems Journal, January 1997
// FILE: MYSEH2.CPP
// To compile: CL MYSEH2.CPP
//==================================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

EXCEPTION_DISPOSITION
__cdecl
_except_handler(
        struct _EXCEPTION_RECORD *ExceptionRecord,
        void * EstablisherFrame,
        struct _CONTEXT *ContextRecord,
        void * DispatcherContext )
{
    printf( "Home Grown handler: Exception Code: %08X Exception Flags %X",
            ExceptionRecord->ExceptionCode, ExceptionRecord->ExceptionFlags );

    if ( ExceptionRecord->ExceptionFlags & 1 )
        printf( " EH_NONCONTINUABLE" );
    if ( ExceptionRecord->ExceptionFlags & 2 )
        printf( " EH_UNWINDING" );
    if ( ExceptionRecord->ExceptionFlags & 4 )
        printf( " EH_EXIT_UNWIND" );
    if ( ExceptionRecord->ExceptionFlags & 8 )
        printf( " EH_STACK_INVALID" );
    if ( ExceptionRecord->ExceptionFlags & 0x10 )
        printf( " EH_NESTED_CALL" );

    printf( "\n" );

    // Punt... We don't want to handle this... Let somebody else handle it
    return ExceptionContinueSearch;
}

void HomeGrownFrame( void )
{
    DWORD handler = (DWORD)_except_handler;
```

```
        __asm
        {                           // Build EXCEPTION_REGISTRATION record:
            push    handler         // Address of handler function
            push    FS:[0]          // Address of previous handler
            mov     FS:[0],ESP      // Install new EXECEPTION_REGISTRATION
        }

        *(PDWORD)0 = 0;             // Write to address 0 to cause a fault

        printf( "I should never get here!\n" );

        __asm
        {                           // Remove our EXECEPTION_REGISTRATION record
            mov     eax,[ESP]       // Get pointer to previous record
            mov     FS:[0], EAX     // Install previous record
            add     esp, 8          // Clean our EXECEPTION_REGISTRATION off stack
        }
    }

    int main()
    {
        _try
        {
            HomeGrownFrame();
        }
        _except( EXCEPTION_EXECUTE_HANDLER )
        {
            printf( "Caught the exception in main()\n" );
        }

        return 0;
    }
```

为简单起见，我用了一点编译器级的异常处理。main 函数只是建立一个 _try/_except 块。_try 块中的是一个对 HomeGrownFrame 函数的调用。函数与前面的 MYSEH 程序中的代码很类似。它在堆栈上创建了一个 EXCEPTION_REGISTRATION 记录并使 FS:[0] 指向此纪录。在建立了新的处理程序后，函数主动引起异常，向 NULL 指针处进行写入：

```
*(PDWORD)0 = 0;
```

这里的异常回调函数，也就是 _except_handler，与前面的那个很不一样。代码先打印出函数的 ExceptionRecord 参数的异常代号和标志。后面会说明打印此异常标志的原因。因为这个 _except_handler 函数并不能修复引起异常的代码，它就返回 ExceptionContinueSearch。这就使得操作系统继续查找链表中的下一个 EXCEPTION_REGISTRATION 记录。下一个异常回调是用于 main 函数中的 _try/_except 代码的。_except 块只是打印“Caught the exception in main()”。此处的异常处理就是简单地将其忽略。此处的一个关键的问题就是执行控制流。当处理程序不能处理异常时，就是在拒绝使控制流在此处继续。接受异常的处理程序则在所有异常处理代码完成之后决定控制流在哪里继续。这一点并不那么显而易见。

当使用结构化异常处理时，如果一场处理程序没能处理异常，则函数可以用一种非正常的方式退出。例如，MYSEH2 的 HomeGrownFrame 函数中的处理程序并没有处理异常。因为异常处理链中后面的某个处理程序（main 函数）处理了此异常，所以引起异常的指令之后的 printf 从未获得执行。从某种意义上说，使用结构化异常处理和使用运行时库函数 setjmp 和 longjmp 差不多。

若是运行 MYSEH2，其输出可能会令人惊讶。看上去似乎调用了两次 _except_handler 函数。第一次是可以理解的，那第二次又是怎么回事呢？

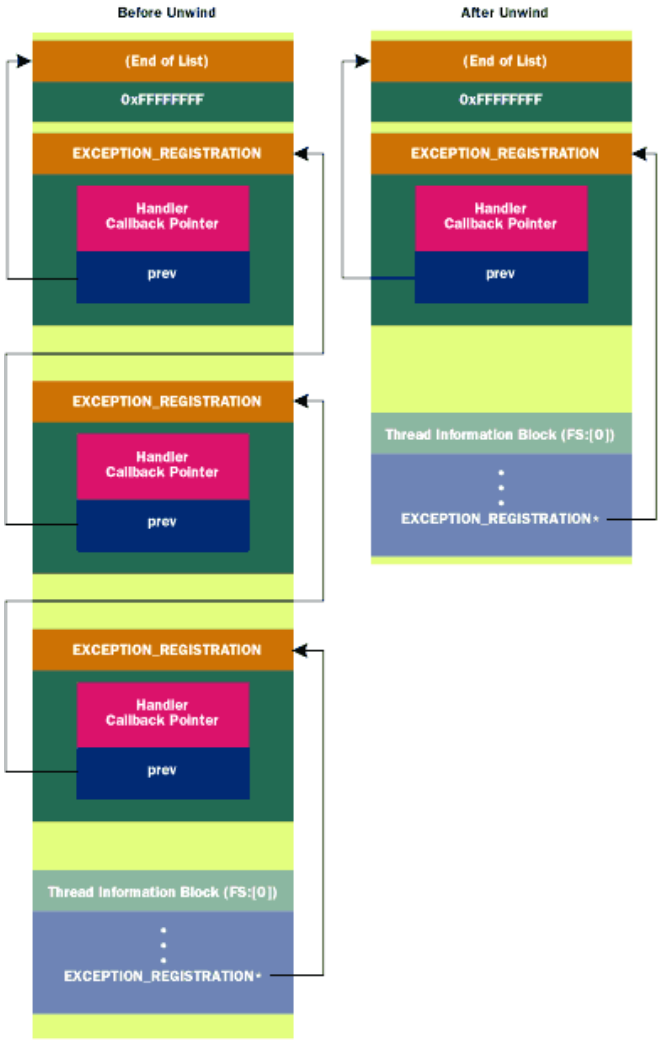
```
Home Grown handler: Exception Code: C0000005 Exception Flags 0
Home Grown handler: Exception Code: C0000027 Exception Flags 2 EH_UNWINDING
Caught the Exception in main()
```

比较由“Home Grown Handler”开始的两行，其区别是显然的，即第一次的异常标志为 0，而第二次则为 2。这里就需要提到 unwinding 的概念。进一步讲，当异常回调拒绝处理异常时，就又被调用了一次。这次回调并没有立即发生，而是更为复杂。我还需要再进一步明确异常发生时的情景。

当异常发生时，系统遍历 EXCEPTION_REGISTRATION 结构体链表直至找到处理此异常的处理程序。一旦找到了处理程序，系统再一次遍历此链表，直到处理异常的节点。在第二次遍历中，系统对所有的异常处理函数进行第二次调用。关键的区别就是在第二次调用中，异常标志被设为值 2。这个值对应着 EH_UNWINDING（EH_UNWINDING 的定义在 Visual C++ 运行时库源代码的 EXCEPT.INC 里，但 Win32 SDK 里并没有等价的定义）。

EH_UNWINDING 是什么意思呢？当异常回调被第二次调用时（带有 EH_UNWINDING 标志），操作系统就给处理函数一次做所需清理的机会。什么样的清理呢？一个很好的例子就是 C++ 类的析构函数。当函数的异常处理函数拒绝处理异常时，控制流一般并不会以正常的方式从函数中退出。现在考虑一个函数，此函数声明了一个局部的 C++ 类。C++ 规范指出析构函数是必须被调用的。第二次标志为 EH_UNWINDING 的异常处理回调就是为做调用析构函数和 _finally 块此类的清理工作提供机会。在异常被处理并且所有之前的 exception frames 都被调用以进行 unwind 之后，程序从回调函数选择的地方继续。但是记住，这并不等于将指令指针设为所要的代码地址并继续执行。继续执行出的代码要求堆栈和帧指针（Intel CPU 的 ESP 和 EBP 寄存器）都设为在处理异常的堆栈帧中相应的值。因此，接受某一异常的处理程序负责将堆栈指针和堆栈帧指针设为包含处理异常的 SEH 代码的堆栈帧中的值。

图 6：从异常中回滚



更一般地说，从异常中的 unwinding 使得位于处理帧的堆栈区域之下的所有的东西都被移除，几乎相当于从未调用过那些函数。unwinding 的另一个效果就是链表中位于处理异常的 EXCEPTION_REGISTRATION 之前的所有 EXCEPTION_REGISTRATIONs 都被从链表中移除。这是有意义的，因为这些 EXCEPTION_REGISTRATION 一般都是在堆栈上构建的。在异常被处理后，堆栈指针和堆栈帧指针在内存中的地址要比从链表中移除的那些 EXCEPTION_REGISTRATIONs 高。图 6 所示即为所述。

Help! Nobody Handled It!

到目前为止我都是假设操作系统总能在 EXCEPTION_REGISTRATION 链表中找到处理程序。要是没有相应的处理程序怎么办？这种情况几乎不会发生。原因是操作系统私地下为每个线程都准备了一个默认的异常处理程序。这个默认的异常处理程序总是链表的最后一个节点并总被选来处理异常。它的行为与一般的异常回调函数有些不同，我之后会说明。

我们来看一下系统在那里安插这个默认的、最终的异常处理程序。这显然要在线程执行的前期进行，要在任何用户代码执行之前。图 7 为我为 BaseProcessStart 写的伪码，BaseProcessStart 是 Windows NT 的 KERNEL32.DLL 的一个内部函数。BaseProcessStart 需要一个参数，即线程入口的地址。BaseProcessStart 运行在新进程的上下文中并调用入口点来启动进程第一个线程。

图 7：BaseProcessStart 的伪代码

```
BaseProcessStart( PVOID lpfnEntryPoint )
{
    DWORD retValue
```

```
        DWORD currentESP;
        DWORD exceptionCode;

        currentESP = ESP;

        _try
        {
            NtSetInformationThread( GetCurrentThread(),
                                    ThreadQuerySetWin32StartAddress,
                                    &lpfnEntryPoint, sizeof(lpfnEntryPoint) );

            retValue = lpfnEntryPoint();

            ExitThread( retValue );
        }
        _except(// filter-expression code
                exceptionCode = GetExceptionInformation(),
                UnhandledExceptionFilter( GetExceptionInformation() ) )
        {
            ESP = currentESP;

            if ( !_BaseRunningInServerProcess )         // Regular process
                ExitProcess( exceptionCode );
            else                                        // Service
                ExitThread( exceptionCode );
        }
    }
```

注意在伪码中，对 lpfnEntryPoint 的调用被封装在了一对 _try 和 _except 中。这个 _try 块就是用来在异常处理链表中安装那个默认的最终异常处理程序的。所有之后注册的异常处理程序都会插在链表中这个处理程序的前面。若 lpfnEntryPoint 函数返回，线程就运行至完成而不引起异常。若是这样，BaseProcessStart 调用 ExitThread 来结束线程。

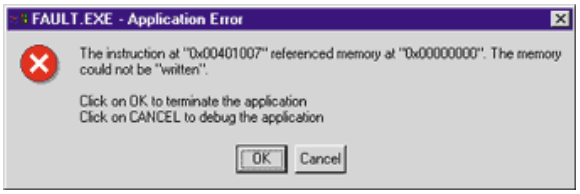
要是另一种情况，即线程发生了异常却再也没有异常处理程序了怎么办？在这种情况下，控制流流进 _except 关键字后的大括号里。在 BaseProcessStart 里，这段代码叫 UnhandledExceptionFilter API，我在后面还会回来介绍它。现在的关键是 UnhandledExceptionFilter API 包含着默认的异常处理函数。

若 UnhandledExceptionFilter 返回的是 EXCEPTION_EXECUTE_HANDLER，BaseProcessStart 的 _except 块就执行。_except 块代码所作的就是调用 ExitProcess 来结束当前进程。仔细考虑一下，这样做还是有意义的；一个常识就是，如果程序引起了异常又没有处理程序能处理此异常，系统就结束该进程。伪码中所展示的正是这种情况。

还要最后补充一点。如果引发异常的线程是作为服务运行的且是用于一个基于线程的服务，则 _except 块并不会调用 ExitProcess 而是调用 ExitThread。没有人会因为一个服务出错而结束整个服务进程。

UnhandledExceptionFilter 中的默认异常处理程序又作了些什么呢？当我在讨论班上提出这个问题时，没几个人能猜出未处理的异常发生时操作系统的默认行为。通过对默认处理程序行为的演示，答案一点即明，人们就都明白了。我只是运行了一个主动引起异常的程序，并指出其结果（见图 8）。

图 8：未捕获的异常对话框



UnhandledExceptionFilter 显示了一个对话框，告诉你发生了一个异常。此时，要么可以结束进程，要么就调试引发异常的进程。在这幕后还有相当多的操作，我在本文结束前再来讲这些东西。正如我所提到的，当异常发生时，用户编写的代码可以得到执行（通常是这样的）。类似地，在 unwind 操作过程中，用户编写的代码也可以得到执行。用户的代码可能仍有问题并引起另一个异常。因此，异常回调函数还可以返回另外两个值：ExceptionNestedException 和 ExceptionCollidedUnwind。显然这些内容就很深了，我并不想在这里介绍。其对于理解基本事实来说太难了。

Compiler-level SEH

尽管我偶尔会使用 _try 和 _except，但目前我所讲到的都是由操作系统实现的。然而，看看我那两个使用纯操作系统 SEH 的程序的变态样子，编译器对此的封装实在是必要的。我们来看一下 Visual C++ 是如何在操作系统级的 SEH 支持之上构建其结构化

异常处理的。

在继续进行之前要记住一件重要的事，那就是另一种编译器可能会与纯操作系统级的 SEH 的做法完全不同。没有人说过必须要实现 Win32 SDK 文档所描述的 _try/_except 模型。例如，Visual Basic 5.0 在其运行时代码里使用了结构化异常处理，但其数据结构与算法与我这里所讲的完全不同。若读一下 Win32 SDK 文档关于结构化异常处理的描述，就会找到所谓的“ frame-based ”的异常处理程序的语义，其形式如下：

```
try {
    // guarded body of code
}
except (filter-expression) {
    // exception-handler block
}
```

简单讲，try 中的所有的代码都被一个构建在函数堆栈帧上的 EXCEPTION_REGISTRATION 保护起来。在函数的入口，新的 EXCEPTION_REGISTRATION 被放入异常处理链表的表头。在 _try 块的结尾处，其 EXCEPTION_REGISTRATION 被从链表头移除。如前所述，异常处理链的表头保存在 FS:[0]。因此，若在调试器中的汇编代码中单步执行，就会看到以下的指令：

```
MOV DWORD PTR FS:[00000000],ESP
```

或是

```
MOV DWORD PTR FS:[00000000],ECX
```

可以十分确信代码正在建立或撤除一个 _try/_except 块。现在知道了一个 _try 块对应着堆栈上的一个 EXCEPTION_REGISTRATION 结构体，那 EXCEPTION_REGISTRATION 里的回调函数呢？使用 Win32 的术语，异常回调函数对应着 filter-expression 代号。filter-expression 就是关键字 _except 后括号中的代码。正是这个 filter-expression 代号决定了是否执行后面 {} 块中的代码。

因为 filter-expression 是程序员写的，程序员可以决定代码中某处发生的异常是否在该处处理。filter-expression 代码可以简单到只有一个“ EXCEPTION_EXECUTE_HANDLER ”，也可以调用一个函数把 p 算到两千万再返回一个代号告诉系统下一步做什么，这是程序员的选择。关键一点是：filter-expression 的代号正对应我前面提到的异常回调函数。

我刚才所讲的都十分简单，但是只是理想中的美好的情形。残酷的现实是事情要复杂的多。对于初学者，filter-expression 并不是由操作系统直接调用的。实际的情形是每个 EXCEPTION_REGISTRATION 的异常处理程序域都指向同一个函数。这个函数在 Visual C++ 的运行时库中，叫做 __except_handler3。是 __except_handler3() 调用了你的 filter-expression 代码，这是后话。

另外一点就是并不是每次进入或退出 _try 块都要建立或撤除 EXCEPTION_REGISTRATION。对于使用 SEH 的每个函数，只创建一个 EXCEPTION_REGISTRATION。换句话说，在一个函数里可以使用多个 _try/_except 组合，但只在堆栈上建立一个 EXCEPTION_REGISTRATION。类似地，可以在一个函数的 _try 块中嵌套另一个 _try 块，Visual C++ 仍然只创建一个 EXCEPTION_REGISTRATION。如果对于整个 EXE 或 DLL 来说一个异常处理程序就足够了以及如果用一个 EXCEPTION_REGISTRATION 就可以处理多个 _try 块，那显然还要有比所见到的更多的机制。这是通过一个一般情况下看不到的表中的数据来完成的。然而，既然本文的目的就是要解剖结构化异常处理，我们就来看一下这些数据结构。

The Extended Exception Handling Frame

Visual C++ 的 SEH 实现并没有使用纯粹的 EXCEPTION_REGISTRATION 结构，而是在结构体的末尾加入了额外的数据域。这个额外的数据的关键之处在于它允许一个函数（__except_handler3）来处理所有的异常并将控制流转向相应的 filter-expressions 和代码中的 _except 块。关于这个 Visual C++ 扩展的 EXCEPTION_REGISTRATION 的一点信息可以从 Visual C++ 的运行时库源代码中的 EXSUP.INC 文件里找到。在这个文件里，可以找到一下定义：

```
; struct _EXCEPTION_REGISTRATION {
;     struct _EXCEPTION_REGISTRATION *prev;
;     void (*handler)(PEXCEPTION_RECORD,
;                     PEXCEPTION_REGISTRATION,
;                     PCONTEXT,
;                     PEXCEPTION_RECORD);
;     struct scopetable_entry *scopetable;
;     int trylevel;
;     int _ebp;
;     PEXCEPTION_POINTERS xpointers;
; };
```

前两个域前面已经见过了，prev 和 handler。他们组成了最基本的 EXCEPTION_REGISTRATION 结构体。新加的是最后的三个域：scopetable、trylevel 和 _ebp。scopetable 域指向一个 scopetable_entries 类型结构体数组，而 trylevel 是这个数组的索引。最后一个域，_ebp，是创建 EXCEPTION_REGISTRATION 之前的堆栈帧指针（EBP）的值。

_ebp 域成为扩展的 EXCEPTION_REGISTRATION 结构体的一部分不是偶然的。结构体包含它是因为大多数函数都以一个 PUSH EBP 开始。这就使得所有其它的 EXCEPTION_REGISTRATION 域可以通过帧指针的负偏移来访问。例如，trylevel 在 [EBP-04]，scopetable 指针在 [EBP-08] 等等。

在扩展的 EXCEPTION_REGISTRATION 结构体后面，Visual C++ 压入了两个额外的值。第一个 DWORD 为一个指向 EXCEPTION_POINTERS 结构体（一个标准的 Win32 结构体）的指针保留空间。这个指针就是调用 GetExceptionInformation API 返回的指针。尽管 SDK 文档隐含提到 GetExceptionInformation 是一个标准的 Win32 API，但事实上 GetExceptionInformation 是一个编译器相关的函数。当调用此函数时，Visual C++ 生成下面的代码：

```
MOV EAX,DWORD PTR [EBP-14]
```

与 GetExceptionInformation 相同，GetExceptionCode 也依赖于编译器。GetExceptionCode 返回的值是 GetExceptionInformation 返回的数据结构中一个域的值。Visual C++ 会生成以下的代码，这些代码的作用留给读者作为练习。

```
MOV EAX,DWORD PTR [EBP-14]
MOV EAX,DWORD PTR [EAX]
MOV EAX,DWORD PTR [EAX]
```

回到扩展的 EXCEPTION_REGISTRATION 结构体，在结构体起始处之前的 8 个字节处，Visual C++ 保留了一个 DWORD 来保存所有 prologue 代码执行后最终的堆栈指针（ESP）。这个 DWORD 就是函数执行时一个普通的 ESP 寄存器的值（当然参数压栈是为了准备调用另外函数的情况除外）。

看起来我好像一股脑儿倒出了一大堆东西，确实是。在向下继续之前，我们先暂停一会儿，复习一下 Visual C++ 为用到结构化异常处理的函数生成的标准异常帧：

```
EBP-00 _ebp
EBP-04 trylevel
EBP-08 scopetable pointer
EBP-0C handler function address
EBP-10 previous EXCEPTION_REGISTRATION
EBP-14 GetExceptionPointers
EBP-18 Standard ESP in frame
```

从操作系统的观点来看，构成纯 EXCEPTION_REGISTRATION 的域仅有两个：[EBP-10] 处的 prev 指针和 [EBP-0Ch] 处的处理函数指针。帧中其它的东西都是依赖于 Visual C++ 实现的。记住这些后，我们来看包含着编译器级 SEH 的 Visual C++ 的运行时库函数，__except_handler3。

__except_handler3 and the scopetable

尽管我非常想将 Visual C++ 的运行时库源代码指点出来并让读者自己去研究 __except_handler3 函数，但是我不能，因为此函数的代码并未提供。这里只好用我仓促拼凑出的 __except_handler3 的伪码来应付一下了（见图 9）。

图 9：__except_handler3 的伪代码

```
int __except_handler3(
    struct _EXCEPTION_RECORD * pExceptionRecord,
    struct EXCEPTION_REGISTRATION * pRegistrationFrame,
    struct _CONTEXT *pContextRecord,
    void * pDispatcherContext )
{
    LONG filterFuncRet
    LONG trylevel
    EXCEPTION_POINTERS exceptPtrs
    PSCOPETABLE pScopeTable

    CLD     // Clear the direction flag (make no assumptions!)

    // if neither the EXCEPTION_UNWINDING nor EXCEPTION_EXIT_UNWIND bit
    // is set...  This is true the first time through the handler (the
    // non-unwinding case)

    if ( ! (pExceptionRecord->ExceptionFlags
            & (EXCEPTION_UNWINDING | EXCEPTION_EXIT_UNWIND)) )
    {
        // Build the EXCEPTION_POINTERS structure on the stack
        exceptPtrs.ExceptionRecord = pExceptionRecord;
        exceptPtrs.ContextRecord = pContextRecord;

        // Put the pointer to the EXCEPTION_POINTERS 4 bytes below the
        // establisher frame.  See ASM code for GetExceptionInformation
        *(PDWORD)((PBYTE)pRegistrationFrame - 4) = &exceptPtrs;
```

```
            // Get initial "trylevel" value
            trylevel = pRegistrationFrame->trylevel

            // Get a pointer to the scopetable array
            scopeTable = pRegistrationFrame->scopetable;

        search_for_handler:

            if ( pRegistrationFrame->trylevel != TRYLEVEL_NONE )
            {
                if ( pRegistrationFrame->scopetable[trylevel].lpfnFilter )
                {
                    PUSH EBP                        // Save this frame EBP

                    // !!!Very Important!!!  Switch to original EBP.  This is
                    // what allows all locals in the frame to have the same
                    // value as before the exception occurred.
                    EBP = &pRegistrationFrame->_ebp

                    // Call the filter function
                    filterFuncRet = scopetable[trylevel].lpfnFilter();

                    POP EBP                         // Restore handler frame EBP

                    if ( filterFuncRet != EXCEPTION_CONTINUE_SEARCH )
                    {
                        if ( filterFuncRet < 0 ) // EXCEPTION_CONTINUE_EXECUTION
                            return ExceptionContinueExecution;

                        // If we get here, EXCEPTION_EXECUTE_HANDLER was specified
                        scopetable == pRegistrationFrame->scopetable

                        // Does the actual OS cleanup of registration frames
                        // Causes this function to recurse
                        __global_unwind2( pRegistrationFrame );

                        // Once we get here, everything is all cleaned up, except
                        // for the last frame, where we'll continue execution
                        EBP = &pRegistrationFrame->_ebp

                        __local_unwind2( pRegistrationFrame, trylevel );

                        // NLG == "non-local-goto" (setjmp/longjmp stuff)
                        __NLG_Notify( 1 );  // EAX == scopetable->lpfnHandler

                        // Set the current trylevel to whatever SCOPETABLE entry
                        // was being used when a handler was found
                        pRegistrationFrame->trylevel = scopetable->previousTryLevel;

                        // Call the _except {} block.  Never returns.
                        pRegistrationFrame->scopetable[trylevel].lpfnHandler();
                    }
                }

                scopeTable = pRegistrationFrame->scopetable;
                trylevel = scopeTable->previousTryLevel

                goto search_for_handler;
            }
            else    // trylevel == TRYLEVEL_NONE
            {
                retvalue == DISPOSITION_CONTINUE_SEARCH;
            }
```

```
        }
        else    // EXCEPTION_UNWINDING or EXCEPTION_EXIT_UNWIND flags are set
        {
            PUSH EBP    // Save EBP
            EBP = pRegistrationFrame->_ebp  // Set EBP for __local_unwind2

            __local_unwind2( pRegistrationFrame, TRYLEVEL_NONE )

            POP EBP     // Restore EBP

            retvalue == DISPOSITION_CONTINUE_SEARCH;
        }
    }
```

尽管 __except_handler3 看上去是成堆的代码，但要记着它只是一个异常回调函数，就像我在文章开头介绍的那样。和 MYSEH.EXE 和 MYSEH2.EXE 中的 homegrown 异常回调函数一样，此函数也需要四个参数。在最高一级上，__except_handler3 被一个 if 语句分为了两部分。这是因为函数可以被调用两次，一次是正常调用，一次是在 unwind 过程中。大部分的代码都用在了 non-unwinding 的回调中。

这段代码的开头首先在堆栈上创建一个 EXCEPTION_POINTERS 结构体，并用 __except_handler3 的两个参数将其初始化。此结构体的地址，即伪码中的 exceptPtrs，被放在 [EBP-14]。这就初始化了 GetExceptionInformation 和 GetExceptionCode 函数用到的指针。接着，__except_handler3 从 EXCEPTION_REGISTRATION 帧（[EBP-04]）取得当前的 trylevel。trylevel 变量用作 scopetable 数组的索引，使得一个 EXCEPTION_REGISTRATION 可以用于一个函数中的多个 _try 块和嵌套的 _try 块。每一个 scopetable 的成员定义如下：

```
typedef struct _SCOPETABLE
{
    DWORD       previousTryLevel;
    DWORD       lpfnFilter
    DWORD       lpfnHandler
} SCOPETABLE, *PSCOPETABLE;
```

SCOPETABLE 中的第二个和第三个参数都很容易理解。它们是 filter-expression 和相应的 _except 块代码的地址。previousTryLevel 域有点复杂。简言之，它是用于嵌套 try 块的。重要的一点是对函数中每一个 _try 块都有一个 SCOPETABLE 成员。

如前所述，当前的 trylevel 指定了要使用的 scopetable 数组成员，也就指定了 filter-expression 和 _except 块的地址。现在，考虑一种情形，即一个 _try 块嵌套在另一个 _try 里。若内层 _try 块的 filter-expression 并没有处理异常，则外层的 _try 块的 filter-expression 就必须处理。 __except_handler3 如何知道哪一个 SCOPETABLE 成员对应着外层的 _try 呢？它的索引由 SCOPETABLE 成员的 previousTryLevel 域给出。使用这种机制，就可以创建任意嵌套的 _try 块。previousTryLevel 域为函数中可能的异常处理链表的一个节点。链表的结尾由一个 0xFFFFFFFF 的 trylevel 指示。

回到 __except_handler3 的代码，在取得当前 trylevel 之后，代码就指向了相应的 SCOPETABLE 成员并调用了 filter-expression 代码。若 filter-expression 返回 EXCEPTION_CONTINUE_SEARCH，__except_handler3 继续查找下一个 SCOPETABLE 成员，这个成员由 previousTryLevel 域指定。若在遍历过程中没有找到处理程序，__except_handler3 就返回 DISPOSITION_CONTINUE_SEARCH，这就使系统移向下一个 EXCEPTION_REGISTRATION 帧。

若 filter-expression 返回 EXCEPTION_EXECUTE_HANDLER，则意味着异常应该由相应的 _except 块代码来处理。这就意味着所有之前的 EXCEPTION_REGISTRATION 帧都要从链表中移除而且要执行 _except 块。第一个活儿是通过调用 __global_unwind2 来完成的，之后我再介绍。在一些清理代码之后，代码的执行就离开了 __except_handler3 并进入 _except 块。奇怪的是控制流从未从 _except 块返回，尽管 __except_handler3 调用了它。如何设置当前的 trylevel 呢？这是由编译器暗自处理的，编译器以 on-the-fly 的方式完成对扩展的 EXCEPTION_REGISTRATION 结构体中的 trylevel 域的修改。如果察看使用 SEH 的函数的汇编代码就会发现函数代码的不同位置都有修改 [EBP-04] 处的当前 trylevel 的代码。__except_handler3 如何调用的 _except，而控制流又为何从不返回呢？因为一个 CALL 指令将返回地址压入堆栈，可以认为这就打乱了堆栈。如果察看一下为 _except 块生成的代码，就会发现它所作的第一件事就是将 EXCEPTION_REGISTRATION 结构体之后的8字节处的 DWORD 加载到 ESP 寄存器中。作为其 prologue 代码的一部分，函数将 ESP 的值保存起来，这样 _except 之后还可以将其取回。

The ShowSEHFrames Program

此时是不是对 EXCEPTION_REGISTRATIONs、scopetables、trylevels、filter-expressions 和 unwinding 这些东西感到有些招架不住，我当初也是这样的。编译器级的结构化异常处理的主题对更多的学习并没有什么帮助。如果没有总体上的了解的话，其中的很多东西就没有意义。当面对一大堆理论时，我很自然地倾向于写些使用这些理论的代码。如果程序能工作，我就知道我的理解（通常是）是正确的。

图 10 是 ShowSEHFrames.EXE 的源代码。它使用 _try/_except 块来建立起由几个 Visual C++ SEH 帧构成的链表。之后，显示每一帧的信息，以及 Visual C++ 为每一帧建立的 scopetables。程序并不生成任何异常。我包含了所有的 _try 块来强制 Visual C++ 生成多个 EXCEPTION_ REGISTRATION 帧，每一帧有多个 scopetable 成员。

图 10：ShowSEHFrames.CPP

```
//=============================================
// ShowSEHFrames - Matt Pietrek 1997
// Microsoft Systems Journal, February 1997
// FILE: ShowSEHFrames.CPP
// To compile: CL ShowSehFrames.CPP
//=============================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#pragma hdrstop

//---------------------------------------------------------------------------
// !!! WARNING !!!  This program only works with Visual C++, as the data
// structures being shown are specific to Visual C++.
//---------------------------------------------------------------------------

#ifndef _MSC_VER
#error Visual C++ Required (Visual C++ specific information is displayed)
#endif

//---------------------------------------------------------------------------
// Structure Definitions
//---------------------------------------------------------------------------

// The basic, OS defined exception frame

struct EXCEPTION_REGISTRATION
{
    EXCEPTION_REGISTRATION* prev;
    FARPROC                 handler;
};


// Data structure(s) pointed to by Visual C++ extended exception frame

struct scopetable_entry
{
    DWORD       previousTryLevel;
    FARPROC     lpfnFilter;
    FARPROC     lpfnHandler;
};

// The extended exception frame used by Visual C++

struct VC_EXCEPTION_REGISTRATION : EXCEPTION_REGISTRATION
{
    scopetable_entry *  scopetable;
    int                 trylevel;
    int                 _ebp;
};

//---------------------------------------------------------------------------
// Prototypes
//---------------------------------------------------------------------------

// __except_handler3 is a Visual C++ RTL function.  We want to refer to
// it in order to print it's address.  However, we need to prototype it since
// it doesn't appear in any header file.

extern "C" int _except_handler3(PEXCEPTION_RECORD, EXCEPTION_REGISTRATION *,
                                PCONTEXT, PEXCEPTION_RECORD);
```

```
//--------------------------------------------------------------------------
// Code
//--------------------------------------------------------------------------

//
// Display the information in one exception frame, along with its scopetable
//

void ShowSEHFrame( VC_EXCEPTION_REGISTRATION * pVCExcRec )
{
    printf( "Frame: %08X  Handler: %08X  Prev: %08X  Scopetable: %08X\n",
            pVCExcRec, pVCExcRec->handler, pVCExcRec->prev,
            pVCExcRec->scopetable );

    scopetable_entry * pScopeTableEntry = pVCExcRec->scopetable;

    for ( unsigned i = 0; i <= pVCExcRec->trylevel; i++ )
    {
        printf( "    scopetable[%u] PrevTryLevel: %08X  "
                "filter: %08X  __except: %08X\n", i,
                pScopeTableEntry->previousTryLevel,
                pScopeTableEntry->lpfnFilter,
                pScopeTableEntry->lpfnHandler );

        pScopeTableEntry++;
    }

    printf( "\n" );
}

//
// Walk the linked list of frames, displaying each in turn
//

void WalkSEHFrames( void )
{
    VC_EXCEPTION_REGISTRATION * pVCExcRec;

    // Print out the location of the __except_handler3 function
    printf( "_except_handler3 is at address: %08X\n", _except_handler3 );
    printf( "\n" );

    // Get a pointer to the head of the chain at FS:[0]
    __asm   mov eax, FS:[0]
    __asm   mov [pVCExcRec], EAX

    // Walk the linked list of frames.  0xFFFFFFFF indicates the end of list
    while (  0xFFFFFFFF != (unsigned)pVCExcRec )
    {
        ShowSEHFrame( pVCExcRec );
        pVCExcRec = (VC_EXCEPTION_REGISTRATION *)(pVCExcRec->prev);
    }
}

void Function1( void )
{
    // Set up 3 nested _try levels (thereby forcing 3 scopetable entries)
    _try
    {
        _try
        {
            _try
            {
```

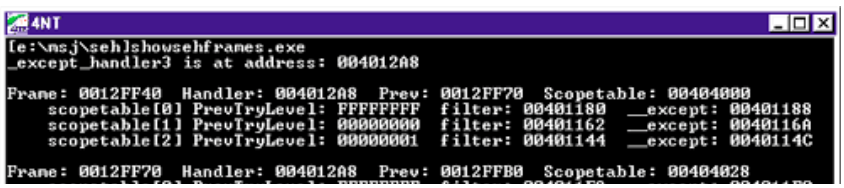
```
                {
                    WalkSEHFrames();    // Now show all the exception frames
                }
                _except( EXCEPTION_CONTINUE_SEARCH )
                {
                }
            }
            _except( EXCEPTION_CONTINUE_SEARCH )
            {
            }
        }
        _except( EXCEPTION_CONTINUE_SEARCH )
        {
        }
    }

    int main()
    {
        int i;

        // Use two (non-nested) _try blocks.  This causes two scopetable entries
        // to be generated for the function.

        _try
        {
            i = 0x1234;     // Do nothing in particular
        }
        _except( EXCEPTION_CONTINUE_SEARCH )
        {
            i = 0x4321;     // Do nothing (in reverse)
        }

        _try
        {
            Function1();    // Call a function that sets up more exception frames
        }
        _except( EXCEPTION_EXECUTE_HANDLER )
        {
            // Should never get here, since we aren't expecting an exception
            printf( "Caught Exception in main\n" );
        }

        return 0;
    }
```

ShowSEHFrames 里重要的函数是 WalkSEHFrames 和 ShowSEHFrame。WalkSEHFrames 首先打印出 __except_handler3 的地址，原因一会儿再讲。接着，函数从 FS:[0] 得到一个指向异常链表表头的指针然后遍历链表中的每一个节点。每个节点都是 VC_EXCEPTION_REGISTRATION 类型的，我定义这个结构体是为了描述 Visual C++ 的异常处理帧。对于链表中的每一个节点，WalkSEHFrames 将指向节点的指针传递给 ShowSEHFrame 函数。

ShowSEHFrame 首先打印异常帧的地址、回调函数地址、前一异常帧的地址和一个指向 scopetable 的指针。接着，对于每一个 scopetable 成员，代码打印出 previous trylevel，filter-expression 的地址和 _except 块的地址。我又是怎么知道 scopetable 中到底有多少个成员的呢？其实我并不知道。我假设 VC_EXCEPTION_REGISTRATION 中的当前 trylevel 比 scopetable 的成员总数少一。

图 11 所示即为 ShowSEHFrames 的运行结果。

图 11：ShowSEHFrames 的运行结果



首先看以“Frame:”开头的每一行。注意每个后继的实例是如何显示堆栈上高地址的异常帧的。接着，在前三个 Frame: 行里，注意 Handler 的值都是相同的（004012A8）。看看输出的开头就知道这个 004012A8 就是 Visual C++ 运行时库的 __except_handler3 函数的地址。这就证实了我前面所说的一个成员处理多个异常。

也许有人会疑惑，因为 ShowSEHFrames 只有两个使用 SEH 的函数，而却有三个使用 __except_handler3 作为回调函数的异常帧。第三个异常帧来自于 Visual C++ 的运行时库。Visual C++ 的运行时库的 CRT0.C 源代码显示对 main 或 WinMain 的调用被封装在了 _try/_except 块中。这个 _try 块的 filter-expression 代码在 WINXFLTR.C 文件中。

回到 ShowSEHFrames，最后一帧的 Handler：此行包含一个不同的地址，77F3AB6C。经过查找，就会发现这个地址是在 KERNEL32.DLL 里。这个特殊的帧是由 KERNEL32.DLL 的 BaseProcessStart 函数安装的，这个函数我在前面讲到过。

Unwinding

在深挖 unwinding 的实现代码之前，我们先来简要总结一下 unwinding 的含义。前面我曾提到异常处理程序信息是如何保存在链表里的，又是如何由线程信息块的第一个 DWORD （FS:[0]）来指向的。因为某一异常的处理程序不一定是链表的头节点，这就需要有一种有序的方法来移除此实际处理程序之前链表中的所有异常处理程序。

正如在 Visual C++ 的 __except_handler3 函数中见到的，unwinding 是由 __global_unwind2 RTL 函数完成的。此函数是对未公开的 RtlUnwind API 的非常简单的封装：

```
__global_unwind2(void * pRegistFrame)
{
    _RtlUnwind( pRegistFrame,
                &__ret_label,
                0, 0 );
    __ret_label:
}
```

尽管 RtlUnwind 是实现编译器级 SEH 的关键的 API，但却没有公开。它是一个 KERNEL32 函数，Windows NT 将 KERNEL32.DLL 调用 forward 到了 NTDLL.DLL，在 NTDLL.DLL 里的也是一个 RtlUnwind 函数。我拼凑了这个函数的一些伪码，即图 12 所示。

图 12：RtlUnwind 的伪代码

```
void _RtlUnwind( PEXCEPTION_REGISTRATION pRegistrationFrame,
                 PVOID returnAddr,  // Not used! (At least on i386)
                 PEXCEPTION_RECORD pExcptRec,
                 DWORD _eax_value )
{
    DWORD   stackUserBase;
    DWORD   stackUserTop;
    PEXCEPTION_RECORD pExcptRec;
    EXCEPTION_RECORD  exceptRec;
    CONTEXT context;

    // Get stack boundaries from FS:[4] and FS:[8]
    RtlpGetStackLimits( &stackUserBase, &stackUserTop );

    if ( 0 == pExcptRec )   // The normal case
    {
        pExcptRec = &excptRec;

        pExcptRec->ExceptionFlags = 0;
        pExcptRec->ExceptionCode = STATUS_UNWIND;
        pExcptRec->ExceptionRecord = 0;
        // Get return address off the stack
        pExcptRec->ExceptionAddress = RtlpGetReturnAddress();
        pExcptRec->ExceptionInformation[0] = 0;
    }

    if ( pRegistrationFrame )
        pExcptRec->ExceptionFlags |= EXCEPTION_UNWINDING;
```

```
    pExcptRec->ExceptionFlags |= EXCEPTION_UNWINDING;
else
    pExcptRec->ExceptionFlags|=(EXCEPTION_UNWINDING|EXCEPTION_EXIT_UNWIND);

context.ContextFlags =
    (CONTEXT_i486 | CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_SEGMENTS);

RtlpCaptureContext( &context );

context.Esp += 0x10;
context.Eax = _eax_value;

PEXCEPTION_REGISTRATION pExcptRegHead;

pExcptRegHead = RtlpGetRegistrationHead();  // Retrieve FS:[0]

// Begin traversing the list of EXCEPTION_REGISTRATION
while ( -1 != pExcptRegHead )
{
    EXCEPTION_RECORD excptRec2;

    if ( pExcptRegHead == pRegistrationFrame )
    {
        _NtContinue( &context, 0 );
    }
    else
    {
        // If there's an exception frame, but it's lower on the stack
        // then the head of the exception list, something's wrong!
        if ( pRegistrationFrame && (pRegistrationFrame <= pExcptRegHead) )
        {
            // Generate an exception to bail out
            excptRec2.ExceptionRecord = pExcptRec;
            excptRec2.NumberParameters = 0;
            excptRec2.ExceptionCode = STATUS_INVALID_UNWIND_TARGET;
            excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

            _RtlRaiseException( &exceptRec2 );
        }
    }

    PVOID pStack = pExcptRegHead + 8; // 8==sizeof(EXCEPTION_REGISTRATION)

    if (    (stackUserBase <= pExcptRegHead )   // Make sure that
        &&  (stackUserTop >= pStack )           // pExcptRegHead is in
        &&  (0 == (pExcptRegHead & 3)) )        // range, and a multiple
    {                                           // of 4 (i.e., sane)
        DWORD pNewRegistHead;
        DWORD retValue;

        retValue = RtlpExecutehandlerForUnwind(
                        pExcptRec, pExcptRegHead, &context,
                        &pNewRegistHead, pExceptRegHead->handler );

        if ( retValue != DISPOSITION_CONTINUE_SEARCH )
        {
            if ( retValue != DISPOSITION_COLLIDED_UNWIND )
            {
                excptRec2.ExceptionRecord = pExcptRec;
             excptRec2.NumberParameters = 0;
                excptRec2.ExceptionCode = STATUS_INVALID_DISPOSITION;
                excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

                RtlRaiseException( &excptRec2 );
```

```
                    RtlRaiseException( &excptRec2 );
                }
                else
                    pExcptRegHead = pNewRegistHead;
            }

            PEXCEPTION_REGISTRATION pCurrExcptReg = pExcptRegHead;
            pExcptRegHead = pExcptRegHead->prev;

            RtlpUnlinkHandler( pCurrExcptReg );
        }
        else    // The stack looks goofy!  Raise an exception to bail out
        {
            excptRec2.ExceptionRecord = pExcptRec;
            excptRec2.NumberParameters = 0;
            excptRec2.ExceptionCode = STATUS_BAD_STACK;
            excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

            RtlRaiseException( &excptRec2 );
        }
    }

    // If we get here, we reached the end of the EXCEPTION_REGISTRATION list.
    // This shouldn't happen normally.

    if ( -1 == pRegistrationFrame )
        NtContinue( &context, 0 );
    else
        NtRaiseException( pExcptRec, &context, 0 );

}

PEXCEPTION_REGISTRATION
RtlpGetRegistrationHead( void )
{
    return FS:[0];
}

_RtlpUnlinkHandler( PEXCEPTION_REGISTRATION pRegistrationFrame )
{
    FS:[0] = pRegistrationFrame->prev;
}

void _RtlpCaptureContext( CONTEXT * pContext )
{
    pContext->Eax = 0;
    pContext->Ecx = 0;
    pContext->Edx = 0;
    pContext->Ebx = 0;
    pContext->Esi = 0;
    pContext->Edi = 0;
    pContext->SegCs = CS;
    pContext->SegDs = DS;
    pContext->SegEs = ES;
    pContext->SegFs = FS;
    pContext->SegGs = GS;
    pContext->SegSs = SS;
    pContext->EFlags = flags; // __asm{ PUSHFD / pop [xxxxxxxx] }
    pContext->Eip = return address of the caller of the caller of this function
    pContext->Ebp = EBP of the caller of the caller of this function
    pContext->Esp = Context.Ebp + 8
}
```

尽管 RtlUnwind 看起来很繁琐，但如果合理地划分一下还是不难理解的。此 API 首先从 FS:[4] 和 FS:[8] 取得线程堆栈的当前栈顶和栈底。这两个值对于后面的健壮性检查是很重要的，这里的健壮性检查就是保证所有被 unwound 的异常帧都落在堆栈的范围内。

接着，RtlUnwind 在堆栈上建立一个 EXCEPTION_RECORD，并将 ExceptionCode 域设为 STATUS_UNWIND。而且 EXCEPTION_RECORD 的 ExceptionFlags 域中的 EXCEPTION_UNWINDING 标志也要置位。指向此 EXCEPTION_RECORD 结构体的指针之后会作为参数传递给每一个异常回调函数。此后，代码调用 _RtlpCaptureContext 函数来创建一个 CONTEXT 结构体，此结构体也会作为异常回调的 unwind 调用的一个参数。RtlUnwind 后面的部分就遍历 EXCEPTION_REGISTRATION 结构体的链表。对于每一帧，代码调用 RtlpExecuteHandlerForUnwind 函数，后面会讲到此函数。正是这个函数用 EXCEPTION_UNWINDING 标志调用了异常回调函数。每次回调之后，相应的异常帧通过调用 RtlpUnlinkHandler 将其移除。当 RtlUnwind 到达第一个参数指定地址的帧时，就停止 unwinding 帧。这些代码中间还有许多用于错误检查的代码，这些代码保证了程序的正常执行。如果出现了问题，RtlUnwind 就会引起异常来告知所遇到的问题，而且此异常的 EXCEPTION_NONCONTINUABLE 标志是置位的。当此标志置位时，是不允许进程继续执行的，因此进程必须结束。

Unhandled Exceptions

本文前面部分我完整描述了 UnhandledExceptionFilter API。一般不用直接调用这个 API（尽管可以）。大多数情况下，它是由 KERNEL32 的默认异常回调的 filter-expression 代码来调用的。前面的 BaseProcessStart 伪码说明了这一点。

图 13 是我给出的 UnhandledExceptionFilter 的伪码。这个 API 的开头有些奇怪（至少在我看来是这样的）。若是一个 EXCEPTION_ACCESS_ VIOLATION 异常，代码就调用 _BasepCheckForReadOnlyResource。尽管我没有提供此函数的伪码，但我这里可以大概说一下。如果是因为向 EXE 或 DLL 的 resource section（.rsrc）进行写入而发生异常，_BasepCurrentTopLevelFilter 就修改引起异常的页的属性，从而允许写操作，UnhandledExceptionFilter 返回 EXCEPTION_CONTINUE_EXECUTION，并重新执行引起异常的指令。

图 13：UnhandledExceptionFilter 的伪代码

```
UnhandledExceptionFilter( STRUCT _EXCEPTION_POINTERS *pExceptionPtrs )
{
    PEXCEPTION_RECORD pExcptRec;
    DWORD currentESP;
    DWORD retValue;
    DWORD DEBUGPORT;
    DWORD   dwTemp2;
    DWORD   dwUseJustInTimeDebugger;
    CHAR    szDbgCmdFmt[256];   // Template string retrieved from AeDebug key
    CHAR    szDbgCmdLine[256];  // Actual debugger string after filling in
    STARTUPINFO startupinfo;
    PROCESS_INFORMATION pi;
    HARDERR_STRUCT  harderr;    // ???
    BOOL fAeDebugAuto;
    TIB * pTib;                 // Thread information block

    pExcptRec = pExceptionPtrs->ExceptionRecord;

    if (   (pExcptRec->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
        && (pExcptRec->ExceptionInformation[0]) )
    {
        retValue =
            _BasepCheckForReadOnlyResource(pExcptRec->ExceptionInformation[1]);

        if ( EXCEPTION_CONTINUE_EXECUTION == retValue )
            return EXCEPTION_CONTINUE_EXECUTION;
     }

    // See if this process is being run under a debugger...
    retValue = NtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPort,
                                         &debugPort, sizeof(debugPort), 0 );

    if ( (retValue >= 0) && debugPort )     // Let debugger have it
        return EXCEPTION_CONTINUE_SEARCH;

    // Did the user call SetUnhandledExceptionFilter?  If so, call their
    // installed proc now.

    if ( _BasepCurrentTopLevelFilter )
    {
```

```
    retValue = _BasepCurrentTopLevelFilter( pExceptionPtrs );

    if ( EXCEPTION_EXECUTE_HANDLER == retValue )
        return EXCEPTION_EXECUTE_HANDLER;

    if ( EXCEPTION_CONTINUE_EXECUTION == retValue )
        return EXCEPTION_CONTINUE_EXECUTION;

    // Only EXCEPTION_CONTINUE_SEARCH goes on from here
}

// Has SetErrorMode(SEM_NOGPFAULTERRORBOX) been called?
if ( 0 == (GetErrorMode() & SEM_NOGPFAULTERRORBOX) )
{
    harderr.elem0 = pExcptRec->ExceptionCode;
    harderr.elem1 = pExcptRec->ExceptionAddress;

    if ( EXCEPTION_IN_PAGE_ERROR == pExcptRec->ExceptionCode )
        harderr.elem2 = pExcptRec->ExceptionInformation[2];
    else
        harderr.elem2 = pExcptRec->ExceptionInformation[0];

    dwTemp2 = 1;
    fAeDebugAuto = FALSE;

    harderr.elem3 = pExcptRec->ExceptionInformation[1];

    pTib = FS:[18h];

    DWORD someVal = pTib->pProcess->0xC;

    if ( pTib->threadID != someVal )
    {
        __try
                    {
            char szDbgCmdFmt[256]
            retValue = _GetProfileStringA( "AeDebug", "Debugger", 0,
                                szDbgCmdFmt, sizeof(szDbgCmdFmt)-1 );

            if ( retValue )
                dwTemp2 = 2;

            char szAuto[8]

            retValue = GetProfileStringA(   "AeDebug", "Auto", "0",
                                            szAuto, sizeof(szAuto)-1 );
            if ( retValue )
                if ( 0 == strcmp( szAuto, "1" ) )
                    if ( 2 == dwTemp2 )
                        fAeDebugAuto = TRUE;
        }
        __except( EXCEPTION_EXECUTE_HANDLER )
        {
            ESP = currentESP;
            dwTemp2 = 1
            fAeDebugAuto = FALSE;
        }
    }

    if ( FALSE == fAeDebugAuto )
    {
        retValue =  NtRaiseHardError(
                            STATUS_UNHANDLED_EXCEPTION | 0x10000000,
```

```
                                4, 0, &harderr,
                                _BasepAlreadyHadHardError ? 1 : dwTemp2,
                                &dwUseJustInTimeDebugger );
            }
            else
            {
                dwUseJustInTimeDebugger = 3;
                retValue = 0;
            }

            if (    retValue >= 0
                &&  ( dwUseJustInTimeDebugger == 3)
                &&  ( !_BasepAlreadyHadHardError )
                &&  ( !_BaseRunningInServerProcess ) )
            {
                _BasepAlreadyHadHardError = 1;

                SECURITY_ATTRIBUTES secAttr = { sizeof(secAttr), 0, TRUE };

                HANDLE hEvent = CreateEventA( &secAttr, TRUE, 0, 0 );

                memset( &startupinfo, 0, sizeof(startupinfo) );

                sprintf(szDbgCmdLine, szDbgCmdFmt, GetCurrentProcessId(), hEvent);

                startupinfo.cb = sizeof(startupinfo);
                startupinfo.lpDesktop = "WinSta0\Default"

                CsrIdentifyAlertableThread();   // ???

                retValue = CreateProcessA(
                                0,              // lpApplicationName
                                szDbgCmdLine,   // Command line
                                0, 0,           // process, thread security attrs
                                1,              // bInheritHandles
                                0, 0,           // creation flags, environment
                                0,              // current directory.
                                &statupinfo,    // STARTUPINFO
                                &pi );          // PROCESS_INFORMATION

                if ( retValue && hEvent )
                {
                    NtWaitForSingleObject( hEvent, 1, 0 );
                    return EXCEPTION_CONTINUE_SEARCH;
                }
            }

            if ( _BasepAlreadyHadHardError )
                NtTerminateProcess(GetCurrentProcess(), pExcptRec->ExceptionCode);
        }

        return EXCEPTION_EXECUTE_HANDLER;
    }

    LPTOP_LEVEL_EXCEPTION_FILTER
    SetUnhandledExceptionFilter(
        LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter );
    {
        // _BasepCurrentTopLevelFilter is a KERNEL32.DLL global var
        LPTOP_LEVEL_EXCEPTION_FILTER previous= _BasepCurrentTopLevelFilter;

        // Set the new value
        _BasepCurrentTopLevelFilter = lpTopLevelExceptionFilter;
```

```
    return previous;    // return the old value
}
```

UnhandledExceptionFilter 的下一个任务是决定进程是否要在 Win32 调试器下运行。也就是说，进程是以 DEBUG_PROCESS 或 DEBUG_ONLY_THIS_PROCESS 标志创建的。UnhandledExceptionFilter 使用 NtQueryInformationProcess 函数来判断进程是否正在被调试。如果是，此 API 就返回 EXCEPTION_CONTINUE_SEARCH，说明系统的其它部分会唤醒调试器进程并告知调试器被调试进程引起了异常。如果有 user-installed unhandled exception filter，则调用它。一般都没有 user-installed 的回调，但是可以用 SetUnhandledExceptionFilter API 装一个。我提供了这个 API 的伪码。这个 API 只是用新的用户回调的地址来修改一个全局变量，然后返回旧回调的值。

做好准备工作后，UnhandledExceptionFilter 就可以进行其主要的工作：用那个老面孔的应用程序错误对话框来通知程序的错误。有两种办法可以避免此对话框的出现。第一种就是进程调用了 SetErrorMode 并设置了 SEM_NOGPFAULTERRORBOX 标志。另一种就是将 AeDebug 注册键值下的 Auto 值设为 1。这时，UnhandledExceptionFilter 略过程序错误对话框并自动启动由 AeDebug 键的 Debugger 值所指定的调试器。如果对“just in time debugging”比较熟悉的话，这就是操作系统对其的支持，之后还会讨论。

大多数情况下，这两种逃避此对话框的条件都为假，UnhandledExceptionFilter 就调用 NTDLL.DLL 函数中的 NtRaiseHardError 函数。正是这个函数唤出了程序错误对话框。这个对话框等待用户点击 OK 结束进程或 Cancel 调试进程。

若点击了 OK，UnhandledExceptionFilter 就返回 EXCEPTION_EXECUTE_HANDLER。调用 UnhandledExceptionFilter 的代码通常以结束自己来回应（就像 BaseProcessStart 代码中那样的）。这就带来一个有趣的问题。多数人认为系统没有处理异常而将进程结束。实际上更准确的说法是系统作了一些工作，这样未处理的异常使进程自己将自己结束。

如果点击了程序错误对话框的 Cancel 才执行了 UnhandledExceptionFilter 真正有意思的代码，这时会调试器加载引起异常的进程。在调试器 attach 到出错进程后，代码首先调用 CreateEvent 来创建一个事件以通知调试器。事件句柄和当前进程 ID 都要传递给 sprintf，sprintf 格式化启动调试器的命令行。万事俱备后，UnhandledExceptionFilter 调用 CreateProcess 来启动调试器。若 CreateProcess 成功，代码对前面创建的事件调用 NtWaitForSingleObject。此调用一直阻塞直到调试器进程通知此事件，指示调试器已经成功地 attach 到出错进程上。UnhandledExceptionFilter 还有其它的零星代码，但我这里只捡重要的说了说。

Into the Inferno

到了目前这个地步，如果还保留什么就太不公平了。我已经讲了发生异常时操作系统如何调用用户定义的函数；讲了一般回调的内部运行以及编译器如何使用它们来实现 _try 和 _catch；讲了没人处理异常时的情况以及系统对其的处理。所剩下的只有起初异常回调是从何处开始的。是的，我们来深入系统内幕来看看结构化异常处理的开始阶段。

图 14 所示为我为 KiUserExceptionDispatcher 和一些相关函数写的伪码。KiUserExceptionDispatcher 位于 NTDLL.DLL 中，它是异常发生后执行的起点。这样说也不是百分之百的准确。例如，在 Intel 体系下，异常会使控制转到一个 ring 0 （内核模式）的处理程序。此处理程序由对应此异常的中断描述符表表项所定义。我将跳过所有的内核模式代码并假设发生异常时 CPU 直接执行 KiUserExceptionDispatcher。

图 14：KiUserExceptionDispatcher 的伪代码

```
KiUserExceptionDispatcher( PEXCEPTION_RECORD pExcptRec, CONTEXT * pContext )
{
    DWORD retValue;

    // Note: If the exception is handled, RtlDispatchException() never returns
    if ( RtlDispatchException( pExceptRec, pContext ) )
        retValue = NtContinue( pContext, 0 );
    else
        retValue = NtRaiseException( pExceptRec, pContext, 0 );

    EXCEPTION_RECORD excptRec2;

    excptRec2.ExceptionCode = retValue;
    excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
    excptRec2.ExceptionRecord = pExcptRec;
    excptRec2.NumberParameters = 0;

    RtlRaiseException( &excptRec2 );
}

int RtlDispatchException( PEXCEPTION_RECORD pExcptRec, CONTEXT * pContext )
{
    DWORD   stackUserBase;
    DWORD   stackUserTop;
```

```
PEXCEPTION_REGISTRATION pRegistrationFrame;
DWORD hLog;

// Get stack boundaries from FS:[4] and FS:[8]
RtlpGetStackLimits( &stackUserBase, &stackUserTop );

pRegistrationFrame = RtlpGetRegistrationHead();

while ( -1 != pRegistrationFrame )
{
    PVOID justPastRegistrationFrame = &pRegistrationFrame + 8;
    if ( stackUserBase > justPastRegistrationFrame )
    {
        pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
        return DISPOSITION_DISMISS; // 0
    }

    if ( stackUsertop < justPastRegistrationFrame )
    {
        pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
        return DISPOSITION_DISMISS; // 0
    }

    if ( pRegistrationFrame & 3 )   // Make sure stack is DWORD aligned
    {
        pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
        return DISPOSITION_DISMISS; // 0
    }

    if ( someProcessFlag )
    {
        // Doesn't seem to do a whole heck of a lot.
        hLog = RtlpLogExceptionHandler( pExcptRec, pContext, 0,
                                        pRegistrationFrame, 0x10 );
    }

    DWORD retValue, dispatcherContext;

    retValue= RtlpExecuteHandlerForException(pExcptRec, pRegistrationFrame,
                                             pContext, &dispatcherContext,
                                             pRegistrationFrame->handler );

    // Doesn't seem to do a whole heck of a lot.
    if ( someProcessFlag )
        RtlpLogLastExceptionDisposition( hLog, retValue );

    if ( 0 == pRegistrationFrame )
    {
        pExcptRec->ExceptionFlags &= ~EH_NESTED_CALL;   // Turn off flag
    }

    EXCEPTION_RECORD excptRec2;

    DWORD yetAnotherValue = 0;

    if ( DISPOSITION_DISMISS == retValue )
    {
        if ( pExcptRec->ExceptionFlags & EH_NONCONTINUABLE )
        {
            excptRec2.ExceptionRecord = pExcptRec;
            excptRec2.ExceptionNumber = STATUS_NONCONTINUABLE_EXCEPTION;
            excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
            excptRec2.NumberParameters = 0
```

```
                RtlRaiseException( &excptRec2 );
            }
            else
                return DISPOSITION_CONTINUE_SEARCH;
        }
        else if ( DISPOSITION_CONTINUE_SEARCH == retValue )
        {
        }
        else if ( DISPOSITION_NESTED_EXCEPTION == retValue )
        {
            pExcptRec->ExceptionFlags |= EH_EXIT_UNWIND;
            if ( dispatcherContext > yetAnotherValue )
                yetAnotherValue = dispatcherContext;
        }
        else    // DISPOSITION_COLLIDED_UNWIND
        {
            excptRec2.ExceptionRecord = pExcptRec;
            excptRec2.ExceptionNumber = STATUS_INVALID_DISPOSITION;
            excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
            excptRec2.NumberParameters = 0
            RtlRaiseException( &excptRec2 );
        }

        pRegistrationFrame = pRegistrationFrame->prev;  // Go to previous frame
    }

    return DISPOSITION_DISMISS;
}


_RtlpExecuteHandlerForException:    // Handles exception (first time through)
    MOV     EDX,XXXXXXXX
    JMP     ExecuteHandler


RtlpExecutehandlerForUnwind:        // Handles unwind (second time through)
    MOV     EDX,XXXXXXXX



int ExecuteHandler( PEXCEPTION_RECORD pExcptRec
                    PEXCEPTION_REGISTRATION pExcptReg
                    CONTEXT * pContext
                    PVOID pDispatcherContext,
                    FARPROC handler ) // Really a ptr to an _except_handler()

    // Set up an EXCEPTION_REGISTRATION, where EDX points to the
    // appropriate handler code shown below
    PUSH    EDX
    PUSH    FS:[0]
    MOV     FS:[0],ESP

    // Invoke the exception callback function
    EAX = handler( pExcptRec, pExcptReg, pContext, pDispatcherContext );

    // Remove the minimal EXCEPTION_REGISTRATION frame
    MOV     ESP,DWORD PTR FS:[00000000]
    POP     DWORD PTR FS:[00000000]

    return EAX;
}

Exception handler used for _RtlpExecuteHandlerForException:
```

```
{
    // If unwind flag set, return DISPOSITION_CONTINUE_SEARCH, else
    // assign pDispatcher context and return DISPOSITION_NESTED_EXCEPTION

    return pExcptRec->ExceptionFlags & EXCEPTION_UNWIND_CONTEXT
                ? DISPOSITION_CONTINUE_SEARCH
                : *pDispatcherContext = pRegistrationFrame->scopetable,
                  DISPOSITION_NESTED_EXCEPTION;
}

Exception handler used for _RtlpExecuteHandlerForUnwind:
{
    // If unwind flag set, return DISPOSITION_CONTINUE_SEARCH, else
    // assign pDispatcher context and return DISPOSITION_COLLIDED_UNWIND

    return pExcptRec->ExceptionFlags & EXCEPTION_UNWIND_CONTEXT
                ? DISPOSITION_CONTINUE_SEARCH

                : *pDispatcherContext = pRegistrationFrame->scopetable,
                  DISPOSITION_COLLIDED_UNWIND;
}
```

KiUserExceptionDispatcher 的关键就是对 RtlDispatchException 的调用。这个调用启动了对注册的异常处理程序的查找。如果处理程序处理了异常并继续执行，则对 RtlDispatchException 的调用不再返回。如果 RtlDispatchException 返回了，则有两种可能：要么调用了 NtContinue 使进程继续，要么就是产生了另一个异常。若是后者，异常就不能再继续了，进程必须结束。接着说 RtlDispatchExceptionCode，这就是遍历异常帧的代码。函数获得一个指向 EXCEPTION_REGISTRATIONs 链表的指针并遍历每一个节点查找处理程序。因为堆栈可能崩溃掉，这个函数非常谨慎。在调用每个 EXCEPTION_REGISTRATION 指定的处理程序之前，代码要保证在线程堆栈中 EXCEPTION_REGISTRATION 是 DWORD 对齐的且前面的 EXCEPTION_REGISTRATION 的地址高。

RtlDispatchException 并不直接调用 EXCEPTION_REGISTRATION 结构体中指定的地址，而是调用 RtlpExecuteHandlerForException 来做这个脏累活儿。根据 RtlpExecuteHandlerForException 内部发生的情况，RtlDispatchException 要么继续遍历异常帧要么产生另一个异常。这个二级异常指示异常回调函数中出现问题不能继续执行。RtlpExecuteHandlerForException 的代码和另一个函数 RtlpExecutehandlerForUnwind 紧密相关。我在前面讲 unwinding 时曾提到这个函数。这两个函数都在将控制送到 ExecuteHandler 函数之前用不同的值加载 EDX 寄存器。换种说法就是 RtlpExecuteHandlerForException 和 RtlpExecutehandlerForUnwind 是同一个 ExecuteHandler 函数的不同的前端。

ExecuteHandler 就是 EXCEPTION_REGISTRATION 的 handler 域被取出和执行的地方。也许看上去有些奇怪，对异常回调函数的调用本身也被一个结构化异常处理程序封装了起来。在这里使用 SEH 尽管有点儿怪，但认真考虑一下还是合理的。如果异常回调引起了另一个异常，操作系统需要知道此事件。根据异常是发生在初始的回调还是 unwind 中的回调，ExecuteHandler 返回 DISPOSITION_NESTED_EXCEPTION 或 DISPOSITION_COLLIDED_UNWIND。这两个可都是“红色警戒！立即关闭！”级别的代号。读者也许像我一样很难让所有的函数都与 SEH 直接关联。类似地，也很难记住谁调用了谁。为了帮助我自己，我画了个图即图 15。

图 15：SEH 中的调用关系

```
KiUserExceptionDispatcher()

    RtlDispatchException()

        RtlpExecuteHandlerForException()

            ExecuteHandler() // Normally goes to __except_handler3

---------

__except_handler3()

    scopetable filter-expression()

    __global_unwind2()

        RtlUnwind()

            RtlpExecuteHandlerForUnwind()

    scopetable __except block()
```

现在，在执行 ExecuteHandler 前设置 EDX 寄存器干什么呢？其实很简单。若调用用户的处理程序时出错，则不管 EDX 里是什么 ExecuteHandler 都会将其作为纯粹的异常处理程序。它将 EDX 寄存器压栈作为最小 EXCEPTION_REGISTRATION 结构体的 handler 域。本质上讲，ExecuteHandler 使用的纯粹的异常处理和我在 MYSEH 和 MYSEH2 程序里使用的差不多。

Conclusion

结构化异常处理是 Win32 的一个奇妙特性。多亏了像 Visual C++ 这样的编译器在它上面加上的支持层，一般的程序员才能用较少的学习代价而从 SEH 中受益。然而，在操作系统这一级，事情可就比 Win32 文档所讲的复杂多了。不幸的是，因为几乎所有的人都觉得系统级 SEH 是个很难的课题，所以至今没有什么这方面的文章。系统级细节方面的文档的缺乏状况一直未得改善。在本文中，我已经展示了系统级的 SEH 是围绕一个相对简单的回调函数展开的。如果理解了回调函数的本质，再在此基础上层曾构建其它的理解层次，系统级的结构化异常处理其实也没那么难掌握。

相关文章

- 微软 MSJ 网站上的英文原文：A Crash Course on the Depths of Win32 Structured Exception Handling
- 本文在 Xfocus 上原始的中文版本
- 下载演示代码（链接到 Microsoft Download）
- 对于结构化异常处理（SEH）的进一步探索
- NT 中的异常帧结构和异常嵌套