January 18, 2022

//

you're reading...

C++, SEHException

Understanding Windows Structured Exception Handling Part 2 – Digging Deeper

Introduction

1. In part 1 of this series of articles, we touched on the basics of Windows Structured Exception Handling (SEH) in Win32.

2. We learned how to manually setup a SEH Frame in C/C++ without the use of the __try/__except block.

3. We also saw how SEH can be used to fix errors that led to exceptions (e.g. recovery from a divide by zero exception).

4. In this part 2, we shall study SEH in greater low-level detail. We will learn how the Visual C++ compiler generate code for __try/__except/__finally blocks in a C/C++ function and get a basic idea how these code work to effect Exception Handling (the execution of __except handlers) and Termination Handling (i.e. the calling of __finally blocks).

5. Along the way, I shall attempt to provide clear concise definitions of terms which are connected with SEH. The Reader may well have heard these terms mentioned while referring to SEH related information but their meaning may not have been explained properly.

Producing Assembly Language Code

1. We will be studying a lot of assembly language code in this article.

2. This will be produced by the Visual C/C++ Compiler by specifying the following option :

C/C++ | Output Files | Assembler Output : Assembly With Source Code (/FAs)

3. With this option turned on, upon every successful compilation, a x86 assembly code source file TestSEH02.asm will be produced and stored in the Project's Debug folder.

SEH Frames

1. What is a SEH Frame ?

2. A SEH Frame is a low-level code construct contained inside a function that enables the handling of Windows Exceptions. It also enables the automatic calling of Termination Handlers inside the function.

3. A SEH Frame is constructed by the Visual C/C++ compiler for a function which contains __try/__except and/or a __try/__finally block(s).

```
void TestSEH()
{
        __try
        {
                __try
                {
                        int* pInt = NULL;
                        *pInt = 100;
                }
                __finally
                {
                        printf("__finally @ TestSEH()\r\n");
                }
        }
        __except (FilterFunction())
        {
                printf("__except @ TestSEH()\r\n");
        }
}
```

4. In the above TestSEH() function, the presence of the __try/__except/__finally blocks will prompt the compiler to emit low-level assembly code that constitutes a SEH Frame.

5. A SEH Frame essentially consists of the following :

- Code to adding a SEH Exception Handler Function to the head of the Linked List of Exception Handler routines of the current TIB.
- Code to insert something known as a Scope Table into the SEH Frame.
- Code to update something known as a Try Level as code is executed through the function.

Adding a SEH Exception Handler Function

1. Upon detecting one or more __try/__except/__finally blocks in a function, the compiler emits code that will access the current TIB at runtime.

2. In the source codes of part 1, we accessed the TIB by calling the NtCurrentTeb() function :

```
NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();
```

NtCurrentTeb() is a macro which expands to :

```
__inline struct _TEB * NtCurrentTeb( void ) { return (struct _TEB *) (ULONG_PTR)
__readfsdword (PcTeb); }
```

3. It uses the intrinsic function __readfsdword() to read an offset from the beginning of the FS register. In the case of NtCurrentTeb(), this offset is PcTeb (== 0x18). The return value of NtCurrentTeb() is thus FS:[0x18] which is a self-reference to the TEB structure itself (see Contents of the TIB on Windows). The TEB structure is synonymous with a NT_TIB structure. Hence we can assume that FS:[0x18] points to a NT_TIB structure.

4. We also saw in part 1 that to add a new SEH handler, we update the TIB's ExceptionList member which is a linked list of EXCEPTION_REGISTRATION_RECORD :

```
    EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.Handler = (PEXCEPTION_ROUTINE)(&MyDivisionByZero02ExceptionRoutine);
    Registration.Next = TIB->ExceptionList;
    TIB->ExceptionList = &Registration;
```

In the above code, we defined a EXCEPTION_REGISTRATION_RECORD, filled in its members and then updated the TIB's ExceptionList member.

5. Now the interesting thing is : when emitting the code to update the SEH handler of the TIB, the compiler does not produce equivalent assembly code that models after code like that above. It takes a more optimized approach. The following is how the assembly code at the start of the TestSEH() function looks like during debugging :

```
void TestSEH()
{
00AD1190 55                   push        ebp
00AD1191 8B EC                mov         ebp,esp
00AD1193 6A FF                push        0FFFFFFFFh
00AD1195 68 30 5F AD 00       push        0AD5F30h
00AD119A 68 26 3F AD 00       push        offset __except_handler3 (0AD3F26h)
00AD119F 64 A1 00 00 00 00    mov         eax,dword ptr fs:[00000000h]
00AD11A5 50                   push        eax
00AD11A6 64 89 25 00 00 00 00 mov         dword ptr fs:[0],esp
...
```

The following is the equivalent assembly code generated for the .asm file output (ignoring the prolog code i.e. the "push ebp" and "mov ebp, esp" code) :

```
1       push    -1
2       push    OFFSET __sehtable$?TestSEH@@YAXXZ
3       push    OFFSET __except_handler3
4       mov     eax, DWORD PTR fs:0
5       push    eax
6       mov     DWORD PTR fs:0, esp
```

6. To see how this code updates the SEH Handler, observe the code at line 3. In TestSEH(), we used the standard SEH Exception Handler and for this, the compiler has chosen the __except_handler3() function. Note the sequence of code :

- At line 3, a pointer to the __except_handler3() function is pushed onto the stack.
- At line 4, the contents of fs:0 (i.e. FS:[0x00]) is moved to the eax register. This is in preparation for the value of FS:[0x00] to be pushed onto the stack.
- At line 5, the contents of eax is pushed onto the stack, thus placing the value of FS:[0x00] onto the stack.
- At line 6, the contents of the esp register is moved into the memory location FS:[0x00]. This action will be explained below.

Believe it or not, this set of code accomplishes the same objective as the code in point 4 above.

7. To understand how the assembly code works, note 2 things :

- The value at the FS:[0x00] memory location.
- The EXCEPTION_REGISTRATION_RECORD structure.

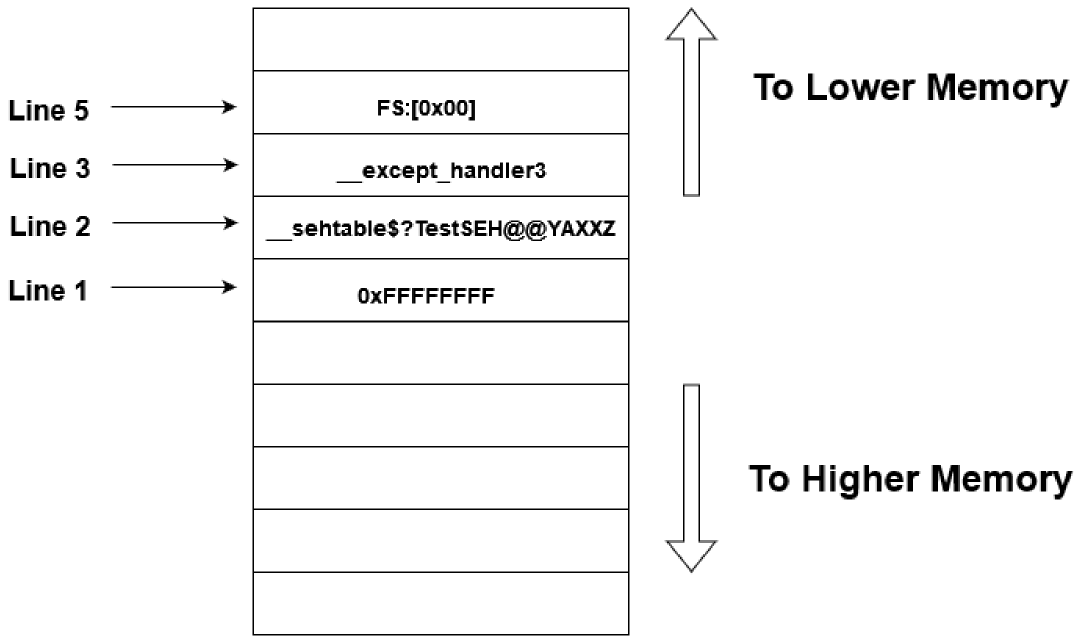
First, note that FS:[0x00] is used by the Windows OS to contain the pointer to the linked list of SEH Handlers (i.e a linked list of EXCEPTION_REGISTRATION_RECORD structures).

Second, observe the EXCEPTION_REGISTRATION_RECORD structure :

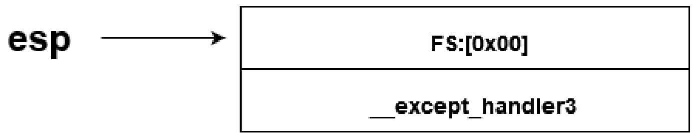
```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD;
```

It has only 2 members : a pointer to the next record and a pointer to the current SEH handler.

8. Back to the assembly code in point 5 above : after the code in lines 1 through 5 have been performed, the memory layout on the stack is as follows :



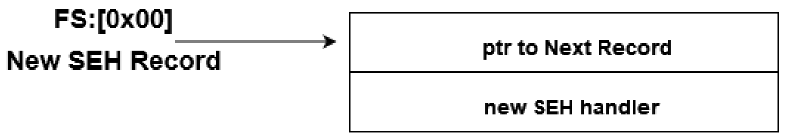
9. We will soon be talking about the __sehtable$?TestSEH@@YAXXZ symbol and the -1 value. But for now observe the stack memory which contains FS:[0x00] and __except_handler3. Notice that these immediately form the field values of a EXCEPTION_REGISTRATION_RECORD record and that with the value of FS:[0x00] being just pushed onto the stack, the esp register will point to its stack memory location :



10. Hence, with the execution of :

```
mov     DWORD PTR fs:0, esp
```

we have the following stack situation :



11. This is how the linked list of SEH Handlers is updated with the latest SEH Handler.

The SEH Scope Table and The TryLevel

1. Associated with every SEH Frame is something known as a Scope Table. The Scope Table is an array of Scope Table Entries which has the following format :

```
typedef struct _EH4_SCOPETABLE_RECORD
{
    ULONG                           EnclosingLevel;
    PEXCEPTION_FILTER_X86           FilterFunc;
    union
    {
        PEXCEPTION_HANDLER_X86      HandlerAddress;
        PTERMINATION_HANDLER_X86    FinallyFunc;
    } u;
} EH4_SCOPETABLE_RECORD, *PEH4_SCOPETABLE_RECORD;
```

The above EH4_SCOPETABLE_RECORD structure declaration is taken from chandler4.c (on my machine, this is located in C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\crt\src\i386\chandler4.c)

2. However, for the purposes of our article, I defined an equivalent structure which is simpler :

```
typedef void (FAR WINAPI* VOIDPROC)();

struct SCOPETABLE_ENTRY
{
    int         EnclosingLevel;
    FARPROC     lpfnFilter;
    VOIDPROC    lpfnHandler;
};
```

2. In a C/C++ function that contains __try/__except/__finally blocks, the compiler divides the function into separate Scope Entries. The compiler also internally generate an integer value known as a TryLevel.

> Each Scope Entry is uniquely linked with a specific __try block.
>
> There is only one TryLevel for the function and this value changes as code moves into and out of __try blocks within the function.

We'll see this with some example code below :

```
void DemoSEHScoping()
{
    printf("TryLevel == -1. No associated ScopeEntry.\r\n");

    __try
    {
        printf("TryLevel == 0. ScopeEntry[0].\r\n");

        __try
        {
            printf("TryLevel == 1. ScopeEntry[1].\r\n");

            __try
            {
                printf("TryLevel == 2. ScopeEntry[2].\r\n");

                int* pInt = NULL;
                *pInt = 100;
            }
            __finally
            {
                printf("__finally for TryLevel == 2. ScopeEntry[2].\r\n");
            }

        }
        __finally
        {
            printf("__finally for TryLevel == 1. ScopeEntry[1].\r\n");
        }
    }
    __except (FilterFunction())
    {
        printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");
    }

    printf("TryLevel == -1. No associated ScopeEntry.\r\n");

    __try
    {
        printf("TryLevel == 3. ScopeEntry[3].\r\n");

        __try
        {
            printf("TryLevel == 4. ScopeEntry[4].\r\n");

            int* pInt = NULL;
            *pInt = 100;
        }
        __finally
        {
            printf("__finally for TryLevel == 4. ScopeEntry[4].\r\n");
        }
    }
    __except (FilterFunction())
    {
        printf("__except for TryLevel == 3. ScopeEntry[3].\r\n");
    }

    printf("TryLevel == -1. No associated ScopeEntry.\r\n");
}
```

3. In the above DemoSEHTryLevels() function, there are 5 Scope Entries. Each Scope Entry is uniquely associated with a __try block.

4. Notice that the TryLevel starts at value -1, then changes values as it enters and leaves __try blocks. TryLevel reverts to -1 whenever it is out of any __try blocks and may not necessarily increment/decrement by 1 whenever it enters/leaves __try blocks. In fact TryLevel helps to identify the current active Scope Entry for the current __try block. This is important when working with SEH filters and exception or termination handlers as we shall see.

5. How do the Scope Entry Table and the TryLevel fit into Structured Exception Handling ? To address this, we need to expand our knowledge of the EstablisherFrame parameter of a SEH Exception Handler function. This is explained next.

The EstablisherFrame

1. Recall in part 1 we had a SEH handler function MyDivisionByZero01ExceptionRoutine() :

```
EXCEPTION_DISPOSITION NTAPI _Function_class_(EXCEPTION_ROUTINE)
MyDivisionByZero01ExceptionRoutine
(
    _Inout_ struct _EXCEPTION_RECORD* pExceptionRecord,
    _In_ PVOID EstablisherFrame,
    _Inout_ struct _CONTEXT* pContextRecord,
    _In_ PVOID DispatcherContext
)
{
    DISPLAY_EXCEPTION_INFO(pExceptionRecord)

    g_iDivisor = 1;

    return ExceptionContinueExecution;
}
```

2. The 2nd parameter EstablisherFrame is in actual fact a pointer to a EXCEPTION_REGISTRATION_RECORD. However, this record is actually part of a larger structure named EH4_EXCEPTION_REGISTRATION_RECORD :

```
typedef struct _EH4_EXCEPTION_REGISTRATION_RECORD
{
    PVOID                               SavedESP;
    PEXCEPTION_POINTERS                 ExceptionPointers;
    EXCEPTION_REGISTRATION_RECORD       SubRecord;
    UINT_PTR                            EncodedScopeTable;
    ULONG                               TryLevel;
} EH4_EXCEPTION_REGISTRATION_RECORD, *PEH4_EXCEPTION_REGISTRATION_RECORD;
```

This structure is defined in the chandler4.c file. On my development machine, it is found in C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\crt\src\i386\chandler4.c

3. The EstablisherFrame parameter points to the SubRecord member. Hence every SEH Handler is able to access the ScopeTable and the current TryLevel by simple memory address offsetting.

4. This can be seen in the function _except_handler4() (chandler4.c) :

```
    //
    // We are passed a registration record which is a field offset from the
    // start of our true registration record.
    //

    RegistrationNode =
        (PEH4_EXCEPTION_REGISTRATION_RECORD)
        ( (PCHAR)EstablisherFrame -
          FIELD_OFFSET(EH4_EXCEPTION_REGISTRATION_RECORD, SubRecord) );
```

5. The FIELD_OFFSET macro is defined as :

```
#define FIELD_OFFSET(type, field)    ((LONG)(LONG_PTR)&(((type *)0)->field))
```

In the above code, the outcome of the subtraction yields a pointer to a full EH4_EXCEPTION_REGISTRATION_RECORD.

6. In the assembly code for DemoSEHScoping(), a EH4_EXCEPTION_REGISTRATION_RECORD structure being formed at the beginning as it was in TestSEH(). The following is taken from the assembly code output for DemoSEHScoping() generated by Visual Studio :

```
push    -1
push    OFFSET __sehtable$?DemoSEHScoping@@YAXXZ
push    OFFSET __except_handler3
mov     eax, DWORD PTR fs:0
push    eax
mov     DWORD PTR fs:0, esp
add     esp, -16
```

7. We can see the same SEH Frame setup as seen for TestSEH(). The __except_handler3() function is placed as the SEH Exception Handler. The current EXCEPTION_REGISTRATION_RECORD is positioned as the next record and FS:[0x00] is then updated to point to the new EXCEPTION_REGISTRATION_RECORD inside the DemoSEHScoping() function.

8. This time, we can see the significance of the -1 value and the __sehtable$?DemoSEHScoping@@YAXXZ symbol being pushed onto the stack. These form the field values of TryLevel and EncodedScopeTable respectively.

The Scope Table Entries

1. __sehtable$?DemoSEHScoping@@YAXXZ is the Scope Table for the DemoSEHScoping() function which can be observed from the assembly code output :

```
__sehtable$?DemoSEHScoping@@YAXXZ DD 0ffffffffH
        DD      FLAT:$LN36@DemoSEHSco
        DD      FLAT:$LN10@DemoSEHSco
        DD      00H
        DD      00H
        DD      FLAT:$LN35@DemoSEHSco
        DD      01H
        DD      00H
        DD      FLAT:$LN34@DemoSEHSco
        DD      0ffffffffH
        DD      FLAT:$LN38@DemoSEHSco
        DD      FLAT:$LN22@DemoSEHSco
        DD      03H
        DD      00H
        DD      FLAT:$LN37@DemoSEHSco
```

2. Here, we can see the direct memory layout for the Scope Table for DemoSEHScoping(). It can be divided into 5 sets of 3 DWORD values. It starts with :

```
        DD      0ffffffffH
        DD      FLAT:$LN36@DemoSEHSco
        DD      FLAT:$LN10@DemoSEHSco
```

which can be directly cast into the fields of a SCOPETABLE_ENTRY structure with :

- EnclosingLevel == -1
- lpfnFilter == FLAT:$LN36@DemoSEHSco
- lpfnHandler == FLAT:$LN10@DemoSEHSco

These 5 sets of SCOPETABLE_ENTRY structures corresponds directly with the fact that there are 5 __try blocks inside the DemoSEHScoping() function. Together they form an array.

3. Now reference back to the DemoSEHScoping() function. You will see that the first SCOPETABLE_ENTRY corresponds with the first __try block :

```
...
printf("TryLevel == -1. No associated ScopeEntry.\r\n");

__try
{
    printf("TryLevel == 0. ScopeEntry[0].\r\n");
    ...
    ...
    ...
}
__except (FilterFunction())
{
    printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");
}
```

Note that although the TryLevel of this __try block is 0, its *enclosing try level* is **-1**.

4. And what do we make of lpfnFilter (FLAT:$LN36@DemoSEHSco) and lpfnHandler (FLAT:$LN10@DemoSEHSco) ? :

- lpfnFilter is the __except filter function.
- lpfnHandler is either the __except or __finally block code.

5. Examining the assembly code output for the TestSEH02.cpp file, we see the following :

```
$LN36@DemoSEHSco:

; 89   :     __except (FilterFunction())

        call    ?FilterFunction@@YAHXZ                  ; FilterFunction
$LN11@DemoSEHSco:
$LN31@DemoSEHSco:
        ret     0
$LN10@DemoSEHSco:
        mov     esp, DWORD PTR __$SEHRec$[ebp]

; 90   :     {
; 91   :         printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");

        push    OFFSET ??_C@_0CN@NNAANJNA@__except?5for?5TryLevel?5?$DN?$DN?50?4?
5Sco@
        call    _printf
        add     esp, 4

; 87   :         }
```

6. $LN36@DemoSEHSco and $LN10@DemoSEHSco are **labels** which point to specific parts of the DemoSEHScoping() function :

- $LN36@DemoSEHSco points to the part of the code which calls FilterFunction().
- $LN10@DemoSEHSco points to the start of the __except handler block.

7. These code locations have to be stored as labels at compile time due to the fact that the exact addresses cannot be identified until at least at link time.

8. A few other important points to note about the arrangement of the low-level code in point 5 are :

- $LN36@DemoSEHSco points to a code location that will return.
- Hence the SCOPETABLE_ENTRY's lpfnFilter member can be called as a function which can return to the caller.
- This is logical since in C/C++ the Filter Function is to return a value of either EXCEPTION_EXECUTE_HANDLER or EXCEPTION_CONTINUE_SEARCH or EXCEPTION_CONTINUE_EXECUTION.
- Hence the label $LN36@DemoSEHSco points to a mini function.
- $LN10@DemoSEHSco on the other hand points to a code location that is not expected to return.
- This is true for an __except block which is expected to continue with the rest of the function's code after it has completed execution.

9. Let's examine the next SCOPETABLE_ENTRY :

```
DD      00H
DD      00H
DD      FLAT:$LN35@DemoSEHSco
```

Here, we see that the EnclosingLevel member is 0, the lpfnFilter member is also 0 and the lpfnHandler points to the label $LN35@DemoSEHSco.

10. This SCOPETABLE_ENTRY corresponds to second __try block of the function and so its TryLevel is 1 but its enclosing level is the previous __try block which is of TryLevel 0.

11. The lpfnFilter member is 0 (i.e. NULL) which means that there is no Filter Function for this __try block. In other words, this is a __try/__finally block.

12. Hence lpfnHandler will point to the start address of the __finally block of the second __try :

```
$LN35@DemoSEHSco:
$LN15@DemoSEHSco:

; 84   :         __finally
; 85   :         {
; 86   :             printf("__finally for TryLevel == 1. ScopeEntry[1].\r\n");

        push    OFFSET ??_C@_0CO@EMPMPPGM@__finally?5for?5TryLevel?5?$DN?$DN?51?4?
5Sc@
        call    _printf
        add     esp, 4
$LN14@DemoSEHSco:
$LN30@DemoSEHSco:
        ret     0
$LN16@DemoSEHSco:

; 87   :         }
```

13. Before we leave this section, note that although the SCOPETABLE_ENTRY array was defined as a global array, it could well have been defined as a local array inside the DemoSEHScoping() function itself. In fact, it would have been more size-efficient if it was defined locally since the SCOPETABLE_ENTRY array is no longer needed once the function exits.

14. In the next section, we will look more closely at the TryLevel and see how its value affects the SCOPETABLE_ENTRY array.

The TryLevel

1. The TryLevel of the function's EH4_EXCEPTION_REGISTRATION_RECORD structure keeps track of the current __try block which is being executed.

2. We have seen that at the beginning of the function, the TryLevel is set to -1.

3. Again referencing the assembly code for DemoSEHScoping() in TestSEH02.asm, we see the following :

```
        __try
        mov     DWORD PTR __$SEHRec$[ebp+20], 0
        {
                printf("TryLevel == 0. ScopeEntry[0].\r\n");

                __try
                mov     DWORD PTR __$SEHRec$[ebp+20], 1
                {
                        printf("TryLevel == 1. ScopeEntry[1].\r\n");

                        __try
                        mov     DWORD PTR __$SEHRec$[ebp+20], 2
                        {
                                printf("TryLevel == 2. ScopeEntry[2].\r\n");
                        }
                        mov     DWORD PTR __$SEHRec$[ebp+20], 1
                        __finally
                        {
                                printf("__finally for TryLevel == 2.
ScopeEntry[2].\r\n");
                        }
                }
                mov     DWORD PTR __$SEHRec$[ebp+20], 0
                __finally
                {
                        printf("__finally for TryLevel == 1. ScopeEntry[1].\r\n");
                }
        }
        mov     DWORD PTR __$SEHRec$[ebp+20], -1
        __except (FilterFunction())
        {
                printf("__except for TryLevel == 0. ScopeEntry[0].\r\n");
        }

        mov     DWORD PTR __$SEHRec$[ebp+20], -1

        printf("TryLevel == -1. No associated ScopeEntry.\r\n");
```

4. __$SEHRec$[ebp+20] essentially works out to be [ebp – 4] which points to the TryLevel member of the local EH4_EXCEPTION_REGISTRATION_RECORD structure.

5. We can see that as the code enters a __try block, TryLevel increments by 1 and as the code exits the same __try block, TryLevel decrements by 1.

Source Codes

1. The source codes for this part 2 can be found in GitHub.

2. Note that I have deliberately set some settings in order to simplify our study of the assembly language code generated for the C++ source codes.

3. These settings include :

- No optimization.
- C/C++ | Code Generation | Basic Runtime Checks : Default.
- C/C++ | Code Generation | Security Checks : Disable Security Checks (/GS-)

4. Optimization has been turned off to enable the Visual C++ compiler to produce template assembly codes for our C++ functions and SEH constructs. These non-optimized code will be easier to understand and follow.

5. The use of Default Basic Runtime Checking code will avoid the inclusion of additional runtime code which will clutter up our assembly language code.

6. By Disabling of Security Checks, the compiler will use the __except_handler3() exception handler which we will use in later parts to study show exception handling code works internally. If Security Checks is enabled, the compiler will emit code to use __except_handler4 instead which is a more advanced exception handler that includes security features. This will be beyond the scope of this series of articles.

Summary

1. In this part 2, we have studied some very low-level SEH code.

2. This will prepare us for the next part in which we study what happens when an Structured Exception occurs. How the Filter Function (if any) is executed, how the __except or __finally blocks are executed.

References

1. Win32 Thread Information Block

2. Microsoft-specific exception handling mechanisms

About Lim Bio Liong

I've been in software development for nearly 20 years specializing in C , COM and C#. It's truly an exicting time we live in, with so much resources at our disposal to gain and share knowledge. I hope my blog will serve a small part in this global knowledge sharing network. For many years now I've been deeply involved with C development work. However since circa 2010, my current work has required me to use more and more on C# with a particular focus

on COM interop. I've also written several articles for CodeProject. However, in recent years I've concentrated my time more on helping others in the MSDN forums. Please feel free to leave a comment whenever you have any constructive criticism over any of my blog posts.
View all posts by Lim Bio Liong »