//

you're reading...

C++, SEHException

Understanding Windows Structured Exception Handling Part 4 – Pseudo __try/__except

Posted by Lim Bio Liong · January 23, 2022 · Leave a comment

Introduction

1. In previous parts of this series of articles, we did a lot of meticulous study of Win32 Structured Exception Handling.

2. Here in part 4, I present a sample code that demonstrates how we can not only construct a custom SEH Frame but also our own pseudo __except filter function and __except block.

3. This is for the benefit of research enthusiasts who may be keen to study SEH deeper and create more customized code constructs.

4. The source codes can be found in GitHub.

Defining a Customized SEH Frame

1. We have seen early in part 1 how a custom SEH Frame can be setup in a function. We have also seen how custom exception handlers can be linked to these SEH Frames. We defined 3 examples of this.

2. The sample codes presented here takes a step further and adds SCOPETABLE_ENTRYs, exception filters and exception blocks to the mix.

3. Two main functions are used for this purpose :

- DoCustomExceptionHandling()
- MyExceptionHandler()

Let’s examine these functions.

4. The following is a listing of DoCustomExceptionHandling() :

```
void DoCustomExceptionHandling()
{
    NT_TIB* TIB = (NT_TIB*)NtCurrentTeb();

    SCOPETABLE_ENTRY scopeentry[1];
    scopeentry[0].EnclosingLevel = -1;
    scopeentry[0].lpfnFilter = MyFilter;
    scopeentry[0].lpfnHandler = MyHandler;

    EH4_EXCEPTION_REGISTRATION_RECORD Registration;
    Registration.EncodedScopeTable = (UINT_PTR)scopeentry;
    Registration.SubRecord.Handler = (PEXCEPTION_ROUTINE)(MyExceptionHandler);
    Registration.SubRecord.Next = TIB->ExceptionList;
    TIB->ExceptionList = &(Registration.SubRecord);

    // Exception raising code below.
    int jmpret = setjmp(mark);

    if (jmpret == 0)
    {
        // Raise the TryLevel.
        Registration.TryLevel = 0;
        // Exception raising code below.
        Foo();
    }
    else
    {
        printf("Exception occurred\r\n");
    }

    // Restore the TryLevel.
    Registration.TryLevel = -1;

    TIB->ExceptionList = TIB->ExceptionList->Next;
}
```

The following is a summary of how it works :

- Instead of using a EXCEPTION_REGISTRATION_RECORD structure, we used the larger and more detailed EH4_EXCEPTION_REGISTRATION_RECORD which also contains a pointer to a SCOPETABLE_ENTRY array.
- The registration of a EH4_EXCEPTION_REGISTRATION_RECORD structure is done as usual via a NT_TIB structure.
- However, this time, we have defined a local SCOPETABLE_ENTRY array of one element.
- This SCOPETABLE_ENTRY element signifies a single __try block of TryLevel 0 and and an EnclosingLevel of -1.
- The SCOPETABLE_ENTRY's __except Filter Function is set to MyFilter().

- The __except Handler is set to MyHandler().
- We then call on the services of the setjmp() function to mark the start of a "guarded" code.
- As documented in MSDN, setjmp() saves the stack environment of the function at which it is called. The stack environment is stored in a global jmp_buf object named mark.
- When setjmp() is called for the first time, it always returns 0. It can later be invoked directly by a call to longjmp().
- To realistically mimic an actual SEH Frame, we set the TryLevel to 0.
- We then call Foo() which will cause an Access Violation Exception.
- To make things interesting, Foo() itself encloses a __try/__finally block. We will later see its __finally block executed in a local unwind operation.
- When the exception occurs in Foo(), a longjmp() will be called and code control returns to setjmp().
- All stack environment values previously stored in mark are put in place again when setjmp() returns a second time.
- longjmp() will also provide the return value (-1) for setjmp() and this time, the else part of the if statement will be run and the printf() statement will be carried out.
- The TryLevel is reset to -1 and the SEH Frame is removed from TIB's ExceptionList.

We next study the MyExceptionHandler() function :

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

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

    EXCEPTION_POINTERS exception_pointers;
    exception_pointers.ExceptionRecord = pExceptionRecord;
    exception_pointers.ContextRecord = pContextRecord;
    RegistrationNode->ExceptionPointers = &exception_pointers;

    int iCurrentTryLevel = RegistrationNode->TryLevel;
    SCOPETABLE_ENTRY* scopetable = (SCOPETABLE_ENTRY*)(RegistrationNode-
>EncodedScopeTable);

    if (pExceptionRecord->ExceptionFlags == 0)
    {
        int iFilterFuncRet = 0;

        for
        (
            int i = iCurrentTryLevel;
            i != -1;
            i = scopetable[i].EnclosingLevel
        )
        {
            if (scopetable[i].lpfnFilter == NULL)
            {
                // The current TryLevel does not have an __except Filter Function.
                // We skip this TryLevel and continue to the TryLevel of the
                // Enclosing __try block.
                continue;
            }

            iFilterFuncRet = scopetable[iCurrentTryLevel].lpfnFilter();

            switch (iFilterFuncRet)
            {
                case EXCEPTION_CONTINUE_SEARCH:
                {
                    // Move on to the next enclosing TryLevel's Scope Table.
                    continue;
                }
                case EXCEPTION_CONTINUE_EXECUTION:
                {
                    // The Filter has resolved the Exception Cause.
                    // We can now continue execution at the point
                    // of the original Exception.
                    return ExceptionContinueExecution;
                }

                case EXCEPTION_EXECUTE_HANDLER:
                {
                    if (scopetable[iCurrentTryLevel].lpfnHandler != NULL)
                    {
                        // First do a Global Unwind. This is to inform all SEH
Exception Handlers
                        // which have been installed -AFTER- the current SEH Handler
to do Unwinding.
                        //
                        // Here, RegistrationNode->SubRecord is the TIB's
ExceptionList Item
                        // which points to the current SEH Exception Handler.
                        //
                        // RtlUnwind() will perform the following :
                        // 1. Get each of these handlers to do Local Unwinding.
                        // 2. Uninstall each of these handlers off the TIB's
ExceptionList.
                        //
                        // Note that the first parameter indicates to RtlUnwind() to
                        // do Unwinding for all SEH Handlers -UP TO- the current SEH
Handler.
                        //
                        RtlUnwind((EXCEPTION_REGISTRATION_RECORD*)(&
(RegistrationNode->SubRecord)), NULL, pExceptionRecord, NULL);

                        // Do a local unwind up to the current TryLevel.
                        // Local unwinding is necessary for the SEH frame that
contains a SEH Handler.
                        // This is because there could be a __finally block beneath
the __except block.
                        _local_unwind2((EXCEPTION_REGISTRATION_RECORD*)(&
(RegistrationNode->SubRecord)), iCurrentTryLevel);

                        // Execute the Exception
                        scopetable[iCurrentTryLevel].lpfnHandler();

                        // Jump to the location of setjmp()
                        // and never return.
                        longjmp(mark, -1);
                    }
                    else
                    {
                        continue;
                    }
                }
            }
        }
    }
    else if (pExceptionRecord->ExceptionFlags == EXCEPTION_UNWINDING)
    {
        _local_unwind2((EXCEPTION_REGISTRATION_RECORD*)(&(RegistrationNode-
>SubRecord)), -1);
    }

    return ExceptionContinueSearch;
}
```

MyExceptionHandler() functions in a similar way to __except_handler3(). The following is a summary of how it works :

- A pointer to a EH4_EXCEPTION_REGISTRATION_RECORD is accessed from the EstablisherFrame and set to the local pointer RegistrationNode.
- The pExceptionRecord and pContextRecord pointers are then kept in a EXCEPTION_POINTERS structure which is then passed to RegistrationNode->ExceptionPointers for possible later use.
- The TryLevel when the exception occurred is set in iCurrentTryLevel.
- A pointer to the SCOPETABLE_ENTRY array is set in local pointer scopetable.
- The ExceptionFlags is then checked to see if MyExceptionHandler() is called for exception handling or for unwinding.
- If unwinding is at work, we call _local_unwind2() to perform a local unwinding of the SEH Frame contained inside DoCustomExceptionHandling().
- If exception handling is at work, a for loop is started to loop through the SCOPETABLE_ENTRYs of the current SEH Frame starting from the item at index iCurrentTryLevel.
- Upon each loop, we check the SCOPETABLE_ENTRY item whether it has an __except filter function.
- If so, we call this Filter Function and note its return value in iFilterFuncRet.
- If not, we set the i index to the EnclosingLevel and search again.
- If iFilterFuncRet == EXCEPTION_CONTINUE_SEARCH, we continue with the next available SCOPETABLE_ENTRY item. If all items have been processed, the handler function returns ExceptionContinueSearch.
- If iFilterFuncRet == EXCEPTION_CONTINUE_EXECUTION, the handler function returns ExceptionContinueExecution immediately.

- If iFilterFuncRet == EXCEPTION_EXECUTE_HANDLER, we perform much the same actions as __except_handler3() assuming that the __except handler is non-NULL :
    - RtlUnwind() is called to perform Global Unwinding.
    - _local_unwind2() is then called to perform local unwinding in the current SEH Frame in which the exception is handled.
    - The pseudo __except handler pointed to by scopetable[iCurrentTryLevel].lpfnHandler() is called.
    - When the handler finished its run, control *will* return to MyExceptionHandler(). However, we mimic the non-local-goto done by __except_handler3() by calling longjmp() with a value of -1.

Noteworthy Points

1. Because Foo() itself contains a SEH Frame, and it is called by DoCustomExceptionHandling(), its SEH Frame is set at a level higher in the TIB's ExceptionList than DoCustomExceptionHandling().

2. As such, when Global Unwinding is done inside MyExceptionHandler(), the SEH Handler of Foo() will be called with the EXCEPTION_UNWINDING flag. The __finally block will be run and so "__finally @ Foo()" will be printed.

3. I have deliberately commented out a code to return EXCEPTION_CONTINUE_SEARCH in MyFilter() :

```
int __stdcall MyFilter()
{
    printf("MyFilter()\r\n");

    return EXCEPTION_EXECUTE_HANDLER;
    //return EXCEPTION_CONTINUE_SEARCH;
}
```

Try swapping the return value to EXCEPTION_CONTINUE_SEARCH. You will see a different outcome :

- This time, MyExceptionHandler() will decline to handle the exception.
- The OS will move to the next lower handler which is the one for the SEH Frame in main().
- The SEH handler for main() will be the one to do the Global Unwind and so Foo's __finally block will be run followed by a call to MyExceptionHandler() with the EXCEPTION_UNWINDING flag.
- MyExceptionHandler() will attempt to do a local unwind for the SEH frame of DoCustomExceptionHandling().
- DoCustomExceptionHandling() does not have a pseudo __finally block associated with it and so nothing will be done.

- Finally, the __except block of main() will be called and “__except @ main()” is printed.

Summary

1. I hope the Reader has found the sample source codes of this part interesting.

2. I also hope it will inspire some to continue to research and emerge some useful findings.

3. I myself am currently doing more research including creating multiple pseudo __try/__except/__finally blocks.

4. So watch out for a possible part 5.

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 »
« Understanding Windows Structured Exception Handling Part 3 – Under The Hood