Basic Exception Handler

What do we have in this session?

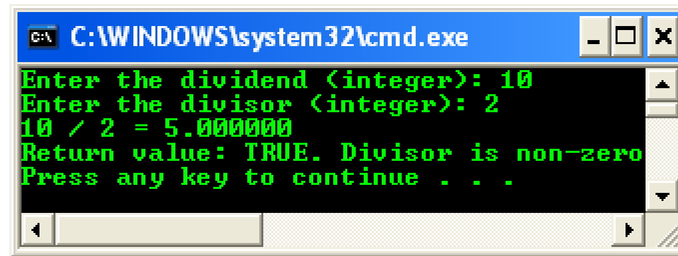
Using an Exception Handler
Example 1: __try and __except
Example 2: __try and __except
Example 3: __try, __except and __finally

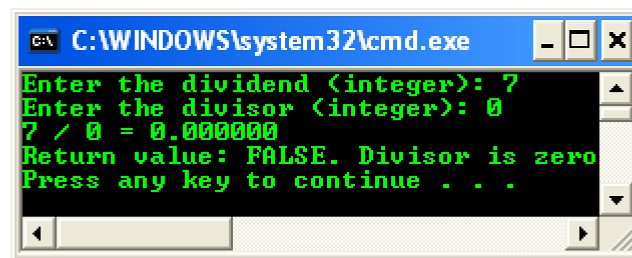
Using an Exception Handler

Example 1: The following examples demonstrate the use of an exception handler.

```
#include <windows.h>
#include <stdio.h>

BOOL SafeDiv(INT32 dividend, INT32 divisor, FLOAT *pResult)
{
    __try
    {
            // Cast to float
        *pResult = (FLOAT)(dividend / divisor);
    }
    __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
            // Zero divisor, return FALSE
        return FALSE;
    }
      // Non-zero divisor, returns TRUE
    return TRUE;
}

int wmain(int argc, WCHAR *argv[])
{
      INT32 dividend = 0, divisor = 0;
      FLOAT *pResult = NULL;
      FLOAT TempVal = 0.0;
      BOOL bRetVal;

      pResult = &TempVal;

      wprintf(L"Enter the dividend (integer): ");
      wscanf_s(L"%d", &dividend, 2);
      wprintf(L"Enter the divisor (integer): ");
      wscanf_s(L"%d", &divisor, 2);
```

```
        bRetVal = SafeDiv(dividend, divisor, pResult);

        wprintf(L"%d / %d = %f\n", dividend, divisor, *pResult);

        // If TRUE
        if(bRetVal)
              wprintf(L"Return value: TRUE. Divisor is non-zero\n");
        else
              wprintf(L"Return value: FALSE. Divisor is zero\n");

        return 0;
}
```



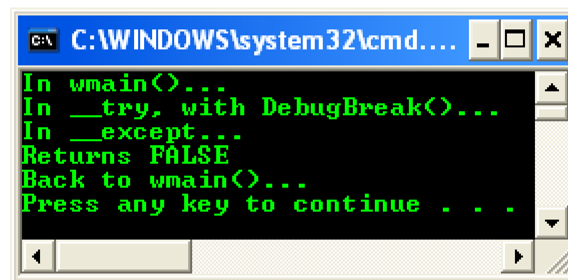


Example 2

The following example function calls the DebugBreak() function and uses structured exception handling to check for a breakpoint exception. If one occurs, the function returns FALSE — otherwise it returns TRUE.
The filter expression in the example uses the GetExceptionCode() function to check the exception type before executing the handler. This enables the system to continue its search for an appropriate handler if some other type of exception occurs.
Also, use of the return statement in the __try block of an exception handler differs from the use of return in the __try block of a termination handler, which causes an abnormal termination of the __try block. This is a valid use of the return statement in an exception handler.

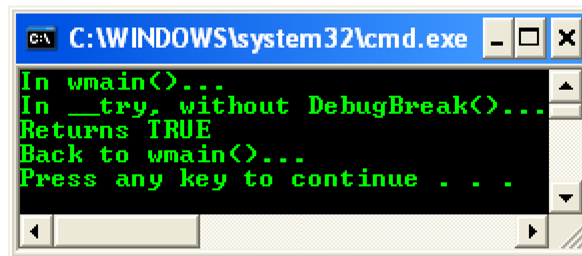
```
#include <windows.h>
#include <stdio.h>

BOOL CheckForDebugger()
{
    __try
```

```
    {
            wprintf(L"In __try, with DebugBreak()...\n");
            // DebugBreak() function will cause a breakpoint exception to
            // occur in the current process. This allows the calling thread
            // to signal the debugger to handle the exception.
            // To cause a breakpoint exception in another process,
            // use the DebugBreakProcess() function.
        DebugBreak();
    }
    __except(GetExceptionCode() == EXCEPTION_BREAKPOINT ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
            wprintf(L"In __except...\n");
        // No debugger is attached, so return FALSE
        // and continue.
        return FALSE;
    }
    return TRUE;
}

int wmain(int argc, WCHAR *argv[])
{
      BOOL bRetVal;

      wprintf(L"In wmain()...\n");
      bRetVal = CheckForDebugger();
      if(bRetVal)
            wprintf(L"Returns TRUE\n");
      else
            wprintf(L"Returns FALSE\n");

      wprintf(L"Back to wmain()...\n");

      return 0;
}
```




```
#include <windows.h>
#include <stdio.h>

BOOL CheckForDebugger()
{
    __try
    {
            wprintf(L"In __try, without DebugBreak()...\n");
            // DebugBreak() function will cause a breakpoint exception to
```

```
            // occur in the current process. This allows the calling thread
            // to signal the debugger to handle the exception.
            // To cause a breakpoint exception in another process,
            // use the DebugBreakProcess() function.
        // DebugBreak();
    }
    __except(GetExceptionCode() == EXCEPTION_BREAKPOINT ?
             EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
            wprintf(L"In __except...\n");
        // No debugger is attached, so return FALSE
        // and continue.
        return FALSE;
    }
    return TRUE;
}

int wmain(int argc, WCHAR *argv[])
{
      BOOL bRetVal;

      wprintf(L"In wmain()...\n");
      bRetVal = CheckForDebugger();
      if(bRetVal)
            wprintf(L"Returns TRUE\n");
      else
            wprintf(L"Returns FALSE\n");

      wprintf(L"Back to wmain()...\n");

      return 0;
}
```



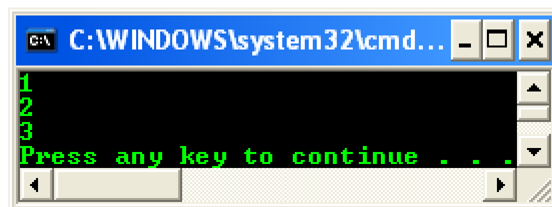
Only return EXCEPTION_EXECUTE_HANDLER from an exception filter when the exception type is expected and the faulting address is known. You should allow the default exception handler to process unexpected exception types and faulting addresses.

Example 3

The following example shows the interaction of nested handlers. The RaiseException() function causes an exception in the guarded body of a termination handler that is inside the guarded body of an exception handler. The exception causes the system to evaluate the FilterFunction() function,

whose return value in turn causes the exception handler to be invoked. However, before the exception-handler block is executed, the __finally block of the termination handler is executed because the flow of control has left the __try block of the termination handler.

```
#include <windows.h>
#include <stdio.h>

DWORD FilterFunction()
{
    wprintf(L"1\n");  // printed first
    return EXCEPTION_EXECUTE_HANDLER;
}

void wmain(int argc, WCHAR *argv[])
{
    __try
    {
        __try
        {
                  // Raises an exception in the calling thread.
            RaiseException(
                1,         // exception code
                0,         // continuable exception
                0, NULL);  // no arguments
        }
        __finally
        {
            wprintf(L"2\n");  // this is printed second
        }
    }
    __except (FilterFunction())
    {
             wprintf(L"3\n");     // this is printed last
    }
}
```



The RaiseException() function enables a process to use structured exception handling to handle private, software-generated, application-defined exceptions. Raising an exception causes the exception dispatcher to go through the following search for an exception handler:

1. The system first attempts to notify the process's debugger, if any.

2. If the process is not being debugged, or if the associated debugger does not handle the exception, the system attempts to locate a frame-based exception handler by searching the stack frames of the thread in which the exception occurred. The system searches the current stack frame first, and then proceeds backward through preceding stack frames.
3. If no frame-based handler can be found, or no frame-based handler handles the exception, the system makes a second attempt to notify the process's debugger.
4. If the process is not being debugged, or if the associated debugger does not handle the exception, the system provides default handling based on the exception type. For most exceptions, the default action is to call the ExitProcess() function.

The values specified in the dwExceptionCode, dwExceptionFlags, nNumberOfArguments, and lpArguments parameters can be retrieved in the filter expression of a frame-based exception handler by calling the GetExceptionInformation() function. A debugger can retrieve these values by calling the WaitForDebugEvent() function.
The C++ `throw` cannot be mixed with SEH. This means that C++ function that implements a __try block cannot declare or use any throwable C++ objects (compiled with /GX) unless they are declared with __declspec(nothrow). Well on the other side, with SEH, it is not possible to catch C++ exception, and C++ typed exception cannot catch SEH selectively, because it is not typed in a way of C++.