Effective Exception Handling in Visual C++

在 C++中进行有效的异常处理

出自 CodeProject，原文链接：
http://www.codeproject.com/Articles/207464/Exception-Handling-in-Visual-Cplusplus

译：
Sleepy
sleepysoft@gmail.com
2012 年 11 月 4 日星期日

介绍

这篇文章描述的是捕获和处理运行在 Windows 下的 Visual C++程序中异常与错误的标准技术。

异常（或者说关键性错误、崩溃）一般来说是你的程序运行不正常，从面不得不停止运行的情况。比如说，如果你的程序访问一块无效的内存地址（如 NULL 指针）、无法分配一个 Buffer（内存不足）、C 语言库的运行时（C run-time libraries，CRT）发现一个错误，并且需要程序立即停止运行等等，这些情况下都会产生一个异常。

一个 C++程序可能会处理几种异常：SEH 异常，这是由操作系统结构化异常处理系统产生的;CRT 错误,这是由 C 语言库运行时——还有——信号(signals)产生的。每种类型的错误都需要安装一个异常处理函数，以便能在出错时拦截这个消息，并做一些错误恢复的处理。

如果你的程序有好几个线程，那么事情或许会更复杂一些。有些异常处理机制是对整个程序有效的，而另一些则仅对当前线程有效，这种情况下，你必须为每个线程都安装异常处理。

你程序中的每个模块（EXE 或 DLL）如果都连接了 CRT 库（无论是静态还是动态的），那么异常处理的方法就主要基于 CRT 的连接方式（？）。

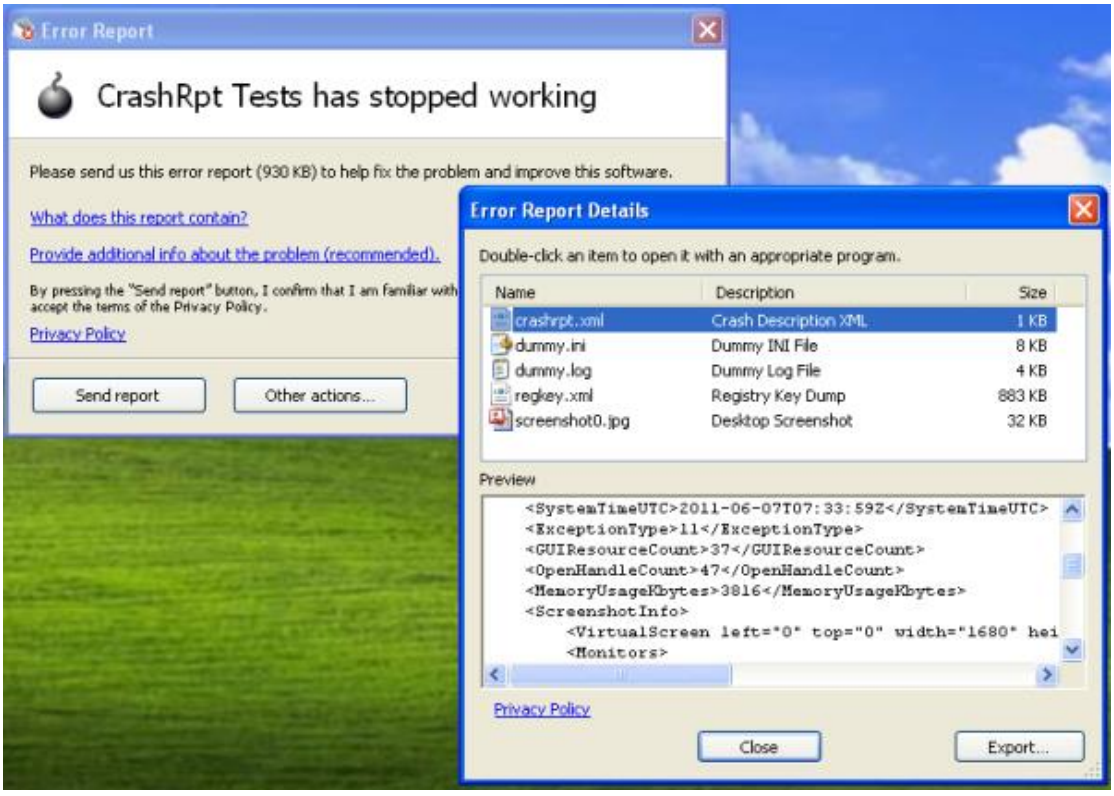
各种各样类型的错误,多线程下对异常的各种不同的捕获方法,还有基于 CRT 连接方式异常处理，这让你的程序捕获所有可能的异常变得困难且麻烦。这篇文章的目的就在于帮助你更好的了解异常机制，并在你的 C++程序中更好更有效的利用它。

这篇文章附带了一个小控制台程序 *ExceptionHandler* 作为例子，这一个例子能产生和捕获各种不同的异常，并且能生成一个崩溃时的 Dump 文件，通过这个文件你可以定位到出错的代码行。

背景知识

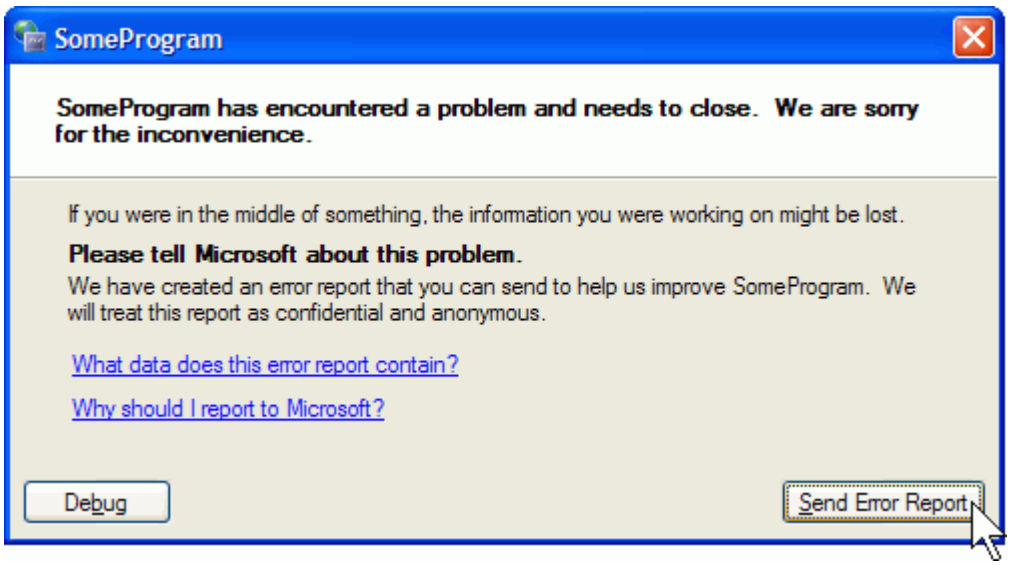
之前有一段时间，我想找到一种可以拦截程序异常的方法，好用在我的一个开源工程上，CrashRpt - A crash reporting library for Windows applications. CrashRpt 库会捕获程序中出现的异常，并收集相关的错误信息（比如崩溃时的 Dump 文件、错误 Log、出错时的屏幕截图等），并且还可以将这些错误报告发送到网络上（图 1）。

图 1 - CrashRpt 库的错误报告窗口及错误详细信息窗口



你可能曾经遇到过 Windows 的错误报告窗口（图 2）突然在屏幕上弹出的情况，CrashRpt 库其实做的是同样的事情，只不过它会将错误报告发送到你的 web 服务器上，而不是像 Winodws 自身错误报告一样，会将错误报告发送到微软的服务器上。

图 2 - Windows 错误报告（传说中的华生医生）窗口



浏览了一下 MSDN，上面说的 `SetUnhandledExceptionFilter()`这个函数我用来捕获非法访问的异常。不过不久后我就发现程序中的一些错误还是捕获不到，我的 CrashRpt 窗口没有弹出，华生医生还是会出现。

我继续在 MDSN 搜寻，发现 CRT 中提供了不少的函数可以用来处理 CRT 的异常。比如说这几个函数 `set_terminate()`, `_set_invalid_parameter_handler()`, `_set_purecall_handler()`。

然后我还发现有些 CRT 的异常处理函数仅对当前线程有效，而有些却对整个进程都有效。

继续研究后，我发现开发人员想要有效的使用异常处理机制，必须了解许多细节方面的问题。

关于异常的只言片语

正如你已经知道的，一个异常或严重错误表明了程序已经无法正常执行，需要立即停止运行。

比如，下面这些情况都会导致异常：

程序访问了一块非法的内存地址（比如 NULL 指针）
在无限递归中，栈溢出
大块数据被写入一片小缓冲区
C++类中的纯虚函数被调用
内存无法分配（内存不足）
向 C++的系统函数中传入非法的参数
C 运行库遇到错误，需要停止程序运行

有两种类型的异常：SEH 异常（结构化异常处理）和标准 C++异常。关于它们的原理与实现，你可以去看看 Vishal Kochhar 写的一篇优秀的文章，在里面有深入的描述：How a Compiler Implements Exception Handling 。

结构化异常处理系统是由操作系统提供的（这意味着所有的 Windows 程序都能产生和处理 SEH 异常）。SEH 异常最初是为 C 语言设计的，但在 C++中也可以使用。

SEH异常是通过__try{}__except(){}这样的结构来处理的。程序中的main()函数就被这样的结构包围着，所以所有没有被处理的 SEH 异常默认都会被捕获，华生医生会弹出来。SEH 异常处理是由 Visual C++编译器指定的。如果你要写兼容性强的代码，你应该在 SEH 结构两端加上#ifdef/#endif（就是说如果 SEH 没被定义，那么 SEH 的代码就不要参与编译）。

示例代码如下：

```
int* p = NULL;   // pointer to NULL
__try
{
    // Guarded code
    *p = 13; // causes an access violation exception
}
__except(EXCEPTION_EXECUTE_HANDLER) // Here is exception filter expression
{
    // Here is exception handler
    // Terminate program
    ExitProcess(1);
}
```

另一方面，C++形式的异常处理系统是由是由 C 运行时库提供的（这意味着只有 C++程序可以产生和处理这种异常）。C++形式异常处理是通过 try{}catch{} 这样的结构来处理的。

示例的代码如下：

（这段代码来自 http://www.cplusplus.com/doc/tutorial/exceptions/）

```
// exceptions
#include <iostream>
using namespace std;
int main () {
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << "An exception occurred. Exception Nr. " << e << endl;
    }
    return 0;
}
```

结构化异常处理

当产生了一个 SEH，一般情况下你会看到华生医生的窗口弹出来（图 2），它提供将错误发送到微软服务器的功能。你甚至还可以利用 RaiseException() 函数手动产生一个异常。

每一个 SEH 异常都有一个错误码，你可以在__except 代码段内通过内部函数 GetExceptionInformation()来将错误码提取出来。为了使用这个内部函数，通常你需要创建一个自定义异常过滤器，就像下面这段代码一样：

这下面的示例代码展示了如何使用 SEH 异常过滤器：

```
int seh_filter(unsigned int code, struct _EXCEPTION_POINTERS* ep)
{
  // Generate error report
  // Execute exception handler
  return EXCEPTION_EXECUTE_HANDLER;
}
void main()
{
  __try
  {
    // .. some buggy code here
  }
  __except(seh_filter(GetExceptionCode(), GetExceptionInformation()))
  {
    // Terminate program
    ExitProcess(1);
  }
}
```

__try{}__except(){}这样的结构几乎可以说是完全面向 C 语言的，然而，你也可以将它复用在 C++上，它也可以像使用 C++形式的异常一样，捕获 C++的异常。如果你要这么做的话，可以使用 C++运行库中的_set_se_translator()函数。

以下是示例代码（出自 MSDN）：

```
// crt_settrans.cpp
// compile with: /EHa
#include <stdio.h>
#include <windows.h>
#include <eh.h>
```

```
void SEFunc();
void trans_func( unsigned int, EXCEPTION_POINTERS* );
class SE_Exception
{
private:
    unsigned int nSE;
public:
    SE_Exception() {}
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};
int main( void )
{
    try
    {
        _set_se_translator( trans_func );
        SEFunc();
    }
    catch( SE_Exception e )
    {
        printf( "Caught a __try exception with SE_Exception.\n" );
    }
}
void SEFunc()
{
    __try
    {
        int x, y=0;
        x = 5 / y;
    }
    __finally
    {
        printf( "In finally\n" );
    }
}
void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp )
{
    printf( "In trans_func.\n" );
    throw SE_Exception();
}
```

然而，__try{}__catch(Expression){}结构的缺点在于，你可能会忘了在可能出现异常的代码两端加上它们，以致于这个你的程序捕获不到这个异常。这些

没有被捕获和处理的异常会被函数所设置的顶层异常处理函数（top-level unhandled exception filter）所捕获和处理。

注：“顶层（top-level）”这个词的意思是，后面调用SetUnhandledExceptionFilter()设置的回调函数会替换前面的。这可以算是一个缺点，因为设置的回调函数无法形成一个调用链，下面将提到的“矢量异常处理机制（Vectored Exception Handling mechanism）”能弥补这个缺点。

异常信息（异常发生前的 CPU 状态）会通过 EXCEPTION_POINTERS 结构传递到异常处理函数中。

以下是示例代码：

```
LONG WINAPI MyUnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionPtrs)
{
  // Do something, for example generate error report
  //..
  // Execute default exception handler next
  return EXCEPTION_EXECUTE_HANDLER;
}
void main()
{
  SetUnhandledExceptionFilter(MyUnhandledExceptionFilter);
  // .. some unsafe code here
}
```

顶层的 SEH 异常处理函数对这个进程中所有线程都是有效的，所以只用在main()函数开始的地方设置一次就够了。

顶层的 SEH 异常处理函数运行在抛出这个异常的线程上下文之中（译注：可以这样理解：哪个线程抛出异常，就是哪个线程去调用它，此时调用如GetCurrentThreadID()这样的函数，返回的就是抛出异常的线程的 ID），这样的好处是这个异常处理函数同时也有能力去恢复一些特定的错误，比如无效的栈等（译注：因为就跑在这个线程里，所以栈什么的就是出错线程的栈）。

如果你的异常处理函数在 DLL 里面，那么使用SetUnhandledExceptionFilter()的时候需要特别小心，如果你的 DLL 在发生 CRUSH 的时候已经被卸载掉了，那么结果会是不确定的（译注：就是会跑飞了，这是当然的不是么）。

注：在 Windows7 下，提供了一个新的函数：RaiseFailFastException()。这个函数允许忽略所有的异常处理函数（包括 SEH 和向量异常处理），直接把异常传递给化生医生。典型的使用情况是，如果你的程序出错，情况非常糟糕，你想立即终止程序并产生一个 Windows 的错误报告。想知道更多信息，请参阅下面提到的参考文献。

向量异常处理

向量异常处理（Vectored Exception Handling (VEH)）是结构化异常处理的扩展，它是在 XP 中被引进的。

要加入一个向量异常处理，你可以使用 AddVectoredExceptionHandler()函数。这种方法的缺点是只适用于 WindowsXP 以及以后的 Windows 版本，所以用它的话，你应该在运行的时候检查一下系统是否有这个函数。

要移除之前安装的异常处理函数，可以使用 RemoveVectoredExceptionHandler()函数。

VEH 允许查看和处理程序中所有的 SEH 异常。为了保证兼容性，当程序某部分触发了 SEH 异常后，会先查找 SEH 异常处理，然后才去执行已安装的 VEH 异常处理。

VEH 的优点在于它的异常处理链，即使有人在你之后安装了一个 VEH 异常处理函数，你之前安装的处理函数仍能捕获到这个异常。

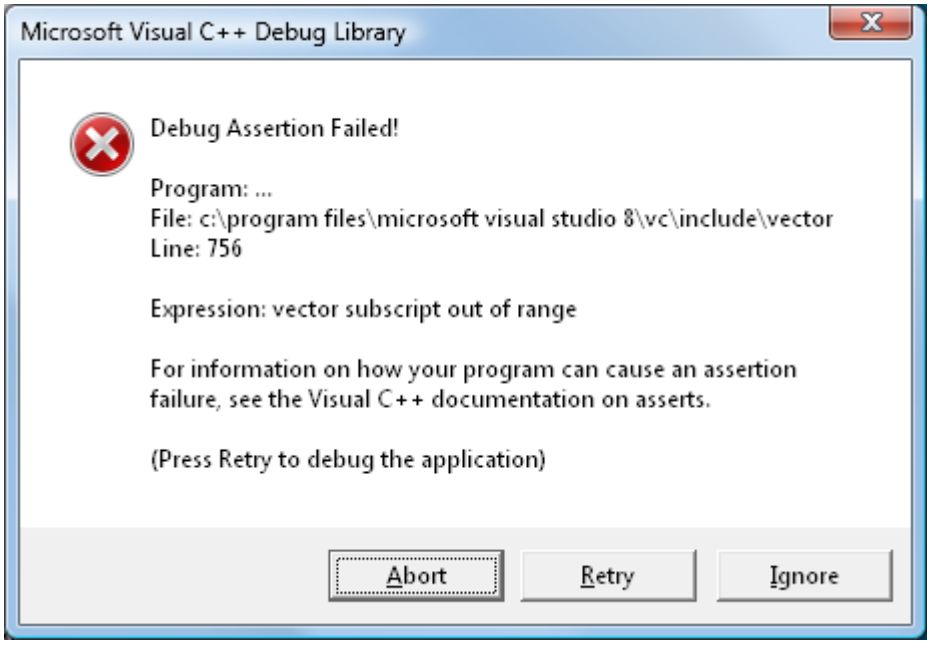
向量异常处理适用于当你需要监控所有 SEH 异常的情况，比如调试器。只不过问题在于，你必须决定哪些异常需要处理，而哪些需要跳过。在程序的代码中，一些可能会发生异常的代码已经特意的被__try{}__except(){}包围并进行处理了，一旦这样的异常经由 VEH 捕获，将不会继续传递到基于 SEH 框架的错误处理中，这可能会引入逻辑上的 BUG（译注：我的理解是，这样的话__except 中的代码就不会被执行，原本依靠 throw 出异常退出数层函数的机制也无法生效，待验证）。

我认为使用 SetUnhandledExceptionFilter()函数比 VEH 更恰当，因为它设置的是顶层的 SEH 异常处理，如果没人处理这个异常，顶层 SEH 异常处理才会被调用，因此你无需去纠结这个异常是应该跳过还是应该处理（译注：因为能到 top-level SEH 这里来的异常都是无人认领的，不像 VEH 一把抓）。

CRT 错误处理

除了 SEH 异常和 C++类型的异常外，C 运行时库提供了它自己的错误处理机制，这种错误机制在程序中也是需要考虑的。当一个 CRT 错误发生的时候，你一般会看到一个 CRT 错误信息窗口（图 3）。

图 3 - CRT 错误信息



终止处理函数（Terminate Handler）

当 CRT 遭遇一个未经处理 C++类型的异常，它会调用 terminate()函数。为了阻止这个调用并做出适当的动作，你应该使用 set_terminate()函数设置一个出错处理函数。

以下是示例代码：

```
void my_terminate_handler()
{
  // Abnormal program termination (terminate() function was called)
  // Do something here
  // Finally, terminate program
  exit(1);
}
void main()
{
  set_terminate(my_terminate_handler);
  terminate();
}
```

此外还有一个 unexpected()函数，这个函数在当前的 Visual C++异常处理中没有实现，尽管如此，可以考虑使用 set_unexpected()函数为 unexpected()设置一个处理函数。（原文：There is the unexpected() function that is not used with the current implementation of Visual C++ exception handling. However, consider using the set_unexpected() function to set a handler for the unexpected() function, too.不大好理解）

纯虚函数调用处理函数（Pure Call Handler）

用_set_purecall_handler()函数可以设置一个捕获纯虚函数被调用错误的错误处理函数，这个函数可以用在 VC++.NET 2003 与之后的版本，这个函数对调用进程的所有线程有效。

以下是示例代码（来自 MSDN）：

```
// _set_purecall_handler.cpp
// compile with: /W1
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
class CDerived;
class CBase
{
public:
   CBase(CDerived *derived): m_pDerived(derived) {};
   ~CBase();
   virtual void function(void) = 0;
   CDerived * m_pDerived;
};
class CDerived : public CBase
{
public:
   CDerived() : CBase(this) {};   // C4355
   virtual void function(void) {};
};
CBase::~CBase()
{
   m_pDerived -> function();
}
void myPurecallHandler(void)
{
   printf("In _purecall_handler.");
   exit(0);
}
int _tmain(int argc, _TCHAR* argv[])
{
   _set_purecall_handler(myPurecallHandler);
   CDerived myDerived;
}
```

new 操作符错误处理

使用_set_new_handler()函数可以设置一个捕获内存分配错误的错误处理函数，这个函数可以用在 VC++.NET 2003 与之后的版本，这个函数对调用进程的所有线程有效。可以考虑使用_set_new_mode()函数来定义 malloc()函数发生错误后的行为。

以下是示例代码（来自 MSDN）：

```
#include <new.h>
int handle_program_memory_depletion( size_t )
{
   // Your code
}
int main( void )
{
   _set_new_handler( handle_program_memory_depletion );
   int *pi = new int[BIG_NUMBER];
}
```

无效参数处理函数（Invalid Parameter Handler）

使用_set_invalid_parameter_handler()函数可以设置当 CRT 调用系统函数时发现一个不会法参数时产生错误的错误处理函数，这个函数可以用在 VC++.NET 2005 与之后的版本，这个函数对调用进程的所有线程有效。

以下是示例代码（来自 MSDN）：

```
// crt_set_invalid_parameter_handler.c
// compile with: /Zi /MTd
#include <stdio.h>
#include <stdlib.h>
#include <crtdbg.h>  // For _CrtSetReportMode
void myInvalidParameterHandler(const wchar_t* expression,
   const wchar_t* function,
   const wchar_t* file,
   unsigned int line,
   uintptr_t pReserved)
{
   wprintf(L"Invalid parameter detected in function %s."
           L" File: %s Line: %d\n", function, file, line);
   wprintf(L"Expression: %s\n", expression);
}
int main( )
{
   char* formatString;
   _invalid_parameter_handler oldHandler, newHandler;
```

```
    newHandler = myInvalidParameterHandler;
    oldHandler = _set_invalid_parameter_handler(newHandler);
    // Disable the message box for assertions.
    _CrtSetReportMode(_CRT_ASSERT, 0);
    // Call printf_s with invalid parameters.
    formatString = NULL;
    printf(formatString);
}
```

C++信号处理

C++提供了程序中断机制，这个机制称为信号。你可以通过 signal()函数去处理这个信号。

在 Visual C++中，有六种不同的信号类型：

SIGABRT 异常终止
SIGFPE 浮点数错误
SIGILL 非法指令
SIGINTCTRL+C 信号
SIGSEGV 非法内存访问
SIGTERM 终止请求

MSDN 上说，SIGILL（非法指令），SIGSEGV（非法内存访问），和 SIGTERM（终止请求）信号在 Windows 下是不会产生的，只是为了保证 ANSI 兼容性。然而实践证明，如果你在主线程中设置了 SIGSEGV 信号处理函数，当异常发生的时候，它会被 CRT 所调用，而不去触发 SetUnhandledExceptionFilter()函数所设置的 SEH 异常处理函数，同时全局变量_pxcptinfoptrs 中包含了异常信息的指针。而在另一个线程中，SIGSEGV 信号不会被触发，用 SetUnhandledExceptionFilter()函数设置的 SEH 异常处理函数仍会被调用。

注：在 Linux 下，信号是异常处理的主要方式（Linux 对 C 运行库，glibc 的实现，也提供了 set_unexpected()和 set_terminate()）。正如你所见到的，在 Windows 下，信号这一机制没有得到应有的重视与应用。取代信号机制的是 C 运行时库提供的几个 Visual C++指定的错误处理函数，比如 _invalid_parameter_handler()等。

全局变量_pxcptinfoptrs 也可用于 SIGFPE 信号处理中，在其它的信号中，它应该会是 NULL。

当浮点数错误产生时，SIGFPE 信号处理函数会被 CRT 调用，比如 0 除的错误。然而，在默认的情况下，浮点数异常是不会产生的，取而代之的是运算结果返回一个 NaN 或无穷大的数。使用_controlfp_s()函数可以控制开启浮点数异常。

你还可以使用 raise()函数手动产生这所有的六个异常。

以下是示例代码：

```
void sigabrt_handler(int)
{
  // Caught SIGABRT C++ signal
  // Terminate program
  exit(1);
}
void main()
{
  signal(SIGABRT, sigabrt_handler);

  // Cause abort
  abort();
}
```

注：尽管在 MSDN 上没有关于 CRT 错误比较全面的文档，不过看起来，你应该为程序中每个新创建的线程手动安装 SIGFPE, SIGILL, 和 SIGSEGV 信号处理函数。而 SIGABRT, SIGINT, 和 SIGTERM 信号处理函数则是对调用进程中的所有线程有效，所以你只需要在 main()函数中安装一次就够了。

获取异常信息

当一个异常产生，一般来说你都希望获取到 CPU 状态，从而定位代码中的错误。你可能还会打算将这些信息传递到 MiniDumpWriteDump()函数中，将信息保存下来留待日后分析（关于如何去做这件事，请参考 Hans Dietrich 的文章：XCrashReport: Exception Handling and Crash Reporting - Part 3）。获得异常信息的途径因采用的异常处理方法不同而不同。

在用 SetUnhandledExceptionFilter()设置的 SEH 异常处理函数中，异常信息是通过作为参数传入函数中的 EXCEPTION_POINTERS 结构得到的。

在__try{}__catch(Expression){}结构中，异常信息是通过调用 GetExceptionInformation()这个内部函数得到的，随后这个值还会作为参数传给 SEH 异常处理函数（原文：and pass it to the SEH exception filter function as a parameter.）。

在 SIGFPE 和 SIGSEGV 信号处理函数中，你可以从 CRT 的全局变量 _pxcptinfoptrs 中获得信息，这个变量定义在<signal.h>里。关于这个变量，在 MSDN 里面没有很好的文档。

在其它的信号处理函数及 CRT 处理函数中，你很难提取到异常信息，不过我在 CRT 的代码中找到了一个变通的方案（参看 CRT8.0 源代码，*invarg.c*, 104 行）。

以下代码展示了如何取得当然 CPU 的状态作为异常信息：

```
#if _MSC_VER>=1300
#include <rtcapi.h>
#endif
#ifndef _AddressOfReturnAddress
// Taken from: http://msdn.microsoft.com/en-us/library/s975zw7k(VS.71).aspx
#ifdef __cplusplus
#define EXTERNC extern "C"
#else
#define EXTERNC
#endif
// _ReturnAddress and _AddressOfReturnAddress should be prototyped before use
EXTERNC void * _AddressOfReturnAddress(void);
EXTERNC void * _ReturnAddress(void);
#endif
// The following function retrieves exception info
void GetExceptionPointers(DWORD dwExceptionCode,
  EXCEPTION_POINTERS** ppExceptionPointers)
{
  // The following code was taken from VC++ 8.0 CRT (invarg.c: line 104)

  EXCEPTION_RECORD ExceptionRecord;
  CONTEXT ContextRecord;
  memset(&ContextRecord, 0, sizeof(CONTEXT));

#ifdef _X86_
  __asm {
      mov dword ptr [ContextRecord.Eax], eax
      mov dword ptr [ContextRecord.Ecx], ecx
      mov dword ptr [ContextRecord.Edx], edx
      mov dword ptr [ContextRecord.Ebx], ebx
      mov dword ptr [ContextRecord.Esi], esi
      mov dword ptr [ContextRecord.Edi], edi
      mov word ptr [ContextRecord.SegSs], ss
      mov word ptr [ContextRecord.SegCs], cs
      mov word ptr [ContextRecord.SegDs], ds
      mov word ptr [ContextRecord.SegEs], es
      mov word ptr [ContextRecord.SegFs], fs
      mov word ptr [ContextRecord.SegGs], gs
      pushfd
      pop [ContextRecord.EFlags]
  }
  ContextRecord.ContextFlags = CONTEXT_CONTROL;
#pragma warning(push)
```

```
#pragma warning(disable:4311)
  ContextRecord.Eip = (ULONG)_ReturnAddress();
  ContextRecord.Esp = (ULONG)_AddressOfReturnAddress();
#pragma warning(pop)
  ContextRecord.Ebp = *((ULONG *)_AddressOfReturnAddress()-1);
#elif defined (_IA64_) || defined (_AMD64_)
  /* Need to fill up the Context in IA64 and AMD64. */
  RtlCaptureContext(&ContextRecord);
#else  /* defined (_IA64_) || defined (_AMD64_) */
  ZeroMemory(&ContextRecord, sizeof(ContextRecord));
#endif  /* defined (_IA64_) || defined (_AMD64_) */
  ZeroMemory(&ExceptionRecord, sizeof(EXCEPTION_RECORD));
  ExceptionRecord.ExceptionCode = dwExceptionCode;
  ExceptionRecord.ExceptionAddress = _ReturnAddress();

  EXCEPTION_RECORD* pExceptionRecord = new EXCEPTION_RECORD;
  memcpy(pExceptionRecord, &ExceptionRecord, sizeof(EXCEPTION_RECORD));
  CONTEXT* pContextRecord = new CONTEXT;
  memcpy(pContextRecord, &ContextRecord, sizeof(CONTEXT));
  *ppExceptionPointers = new EXCEPTION_POINTERS;
  (*ppExceptionPointers)->ExceptionRecord = pExceptionRecord;
  (*ppExceptionPointers)->ContextRecord = pContextRecord;
}
```

异常处理函数和 CRT 的连接方式

程序中的每个模块（EXE,DLL）都连接到了 CRT（C 运行时库）。你可能是以多线程静态库方式或以多线程动态库方式连接的 CRT。当你设置了 CRT 错误处理函数，比如终止处理函数（Terminate Handler）、意外处理函数（unexpected handler，好像前面没提到过）、纯虚函数调用处理函数（Pure Call Handler）、new 操作符错误处理函数、无效参数处理函数（Invalid Parameter Handler）、信号处理函数，它们会为调用者连接到的 CRT 模块工作，而不会接受其它不同的 CRT 模块的异常（如果存在的话），因为每个 CRT 模块都有其自己内部状态。

几个工程模块可能会同享单个 CRT 的 DLL，这样会使连接到 CRT 的代码尺寸减小到最少，这时 CRT 中所有异常都可以一次捕获完。这正是为什么推荐使用多线程 CRT DLL 来进行连接的原因。尽管如此，许多开发者仍宁可使用 CRT 的静态连接，因为它比起比动态连接的方式来说，更容易发布为一个独立的可执行模块（要获得更多的信息，请参阅 Martin Richter 的文章：Create projects easily with private MFC, ATL, and CRT assemblies）。

如果你打算以静态连接的方式使用 CRT 库（这不推荐），并且想使用一些异常处理的功能，你需要把这些功能函数编译成静态库，作法是为连接器指定 /NODEFAULTLIB 参数，然后将其连接到程序中的每一个 EXE 和 DLL 模块。你还必

须为程序中的每一个模块安装一次 CRT 错误处理函数，不过 SEH 异常处理函数仍只需安装一次。

Visual C++编译标志

有几个 Visual C++编译开关是和异常处理有关的，你可以在”工程属性（解决方案窗口的工程上点右键）->配置配置->C/C++->代码生成“下面找到（英文版的是 project *Properties->Configuration Properties->C/C++->Code Generation*）。

异常处理模式

你可以为你的 Visual C++编译器设置一个异常处理模式，通过`/EHs`（或 `EHsc`）来指定同步异常处理模式，或用`/EHa` 来指定异步异常处理模式。异步异常处理模式可以用来强制`try{}catch(){}`结构同时用来捕获 SEH 异常和 C++类型的异常（使用`_set_se_translator()`函数可以达到同样的效果）。如果同步模式被采用，SEH 异常不会被 `try{}catch(){}`结构捕获。在之前版本的 Visual C++中，默认采用的是异步模式，但在后续版本中，默认采用的是同步模式。

浮点异常

可以使用*/fp:except* 编译标记来启用浮点异常，它默认是不启用的，因此浮点异常是不会产生的。想查阅更多信息，请参照引用部分的*/fp:except* compiler flag 文章。

缓冲区安全检查

默认情况下，编译标记中指明了*/GS*（缓冲区安全检查），这会强制编译器加入检查缓冲区溢出的代码。当向一个小缓冲区中写入一大块数据时，会导致缓冲区溢出的情况。

注：在 Visual C++ .NET (CRT 7.1)中，你可以使用`_set_security_error_handler()`函数注册一个处理函数，这个处理函数会在发生缓冲区溢出时被 CRT 调用。但是这个函数在 CRT 后面的版本中被弃用。

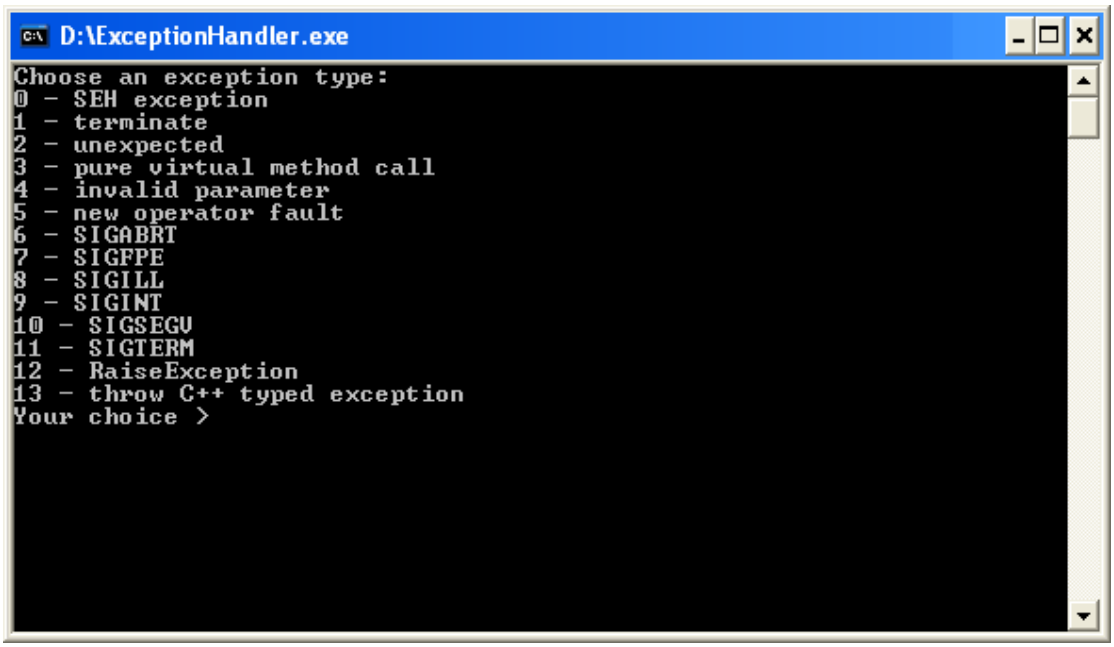
从 CRT8.0 起，你不能在代码中捕获缓冲区溢出错误，当发现缓冲区溢出时，CRT 会立即调用华生医生，而不是调用默认异常处理函数。这么做是出于安全原因，微软也并不打算改变这种方式。想知道更多的信息，请参看以下链接：

- https://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=101337
- http://blog.kalmbachnet.de/?postid=75

关于代码的使用

文章附有一个名为 *ExceptionHandler* 的 Demo，是一个控制台程序，这个 Demo 可以产生和捕获各种不同的异常，并产生一个 minidump 文件，这个程序运行界面如下：

图 4 - *ExceptionHandler* 例子程序



来看看它是怎么工作的，你需要选择一个异常类型（输入一个 0 到 13 的数字），然后按下回车键。然后一个异常会产生出来，并且会被捕获。随后异常处理函数会调用产生 minidump 的代码，一个由 *dbghelp.dll* 提供的名为 `MiniDumpWriteDump()`的函数。minidump 文件生成在当前目录下，名字为 *crashdump.dmp*。你可以双击它，在 Visual Studio 中将其打开，然后按 F5 开始运行，之后就可以看到产生异常的代码了。

程序中包含两大部分：一是产生异常的部分，二是捕获异常并产生崩溃时 minidump 的部分。这两个部分会在下面进行说明和描述：

main.cpp 文件包含了 main()函数，main()函数里面是一个大大的 switch case 函数，用来选择产生异常的类型。为了产生异常，main 函数调用了 `raise()` 或 `RaiseException()`函数、抛出一个 C++类型异常、调用一些辅助代码，像 `RecurseAlloc()`或 `sigfpe_test()`等等。

在 main()函数的一开始，就创建了一个 `CCrashHandler` 的实例，这个实例在后面用来捕获一个异常。调用 `CCrashHandler::SetProcessExceptionHandlers()` 函数来设置一个对整个程序有效的异常处理函数（如 SEH）。调用 `CCrashHandler::SetThreadExceptionHandlers()`函数来设置仅对当前个线程有效的异常处理函数（如预料外异常或终止处理函数）。

下面贴出来的就是程序中的 main 函数

```
void main()
{
    CCrashHandler ch;
    ch.SetProcessExceptionHandlers();
```

```
    ch.SetThreadExceptionHandlers();

    printf("Choose an exception type:\n");
    printf("0 - SEH exception\n");
    printf("1 - terminate\n");
    printf("2 - unexpected\n");
    printf("3 - pure virtual method call\n");
    printf("4 - invalid parameter\n");
    printf("5 - new operator fault\n");
    printf("6 - SIGABRT\n");
    printf("7 - SIGFPE\n");
    printf("8 - SIGILL\n");
    printf("9 - SIGINT\n");
    printf("10 - SIGSEGV\n");
    printf("11 - SIGTERM\n");
    printf("12 - RaiseException\n");
    printf("13 - throw C++ typed exception\n");
    printf("Your choice >  ");

    int ExceptionType = 0;
    scanf_s("%d", &ExceptionType);

    switch(ExceptionType)
    {
    case 0: // SEH
        {
            // Access violation
            int *p = 0;
#pragma warning(disable : 6011)
// warning C6011: Dereferencing NULL pointer 'p'
            *p = 0;
#pragma warning(default : 6011)
        }
        break;
    case 1: // terminate
        {
            // Call terminate
            terminate();
        }
        break;
    case 2: // unexpected
        {
            // Call unexpected
            unexpected();
```

```
        }
        break;
    case 3: // pure virtual method call
        {
            // pure virtual method call
            CDerived derived;
        }
        break;
    case 4: // invalid parameter
        {
            char* formatString;
            // Call printf_s with invalid parameters.
            formatString = NULL;
#pragma warning(disable : 6387)
// warning C6387: 'argument 1' might be '0': this does
// not adhere to the specification for the function 'printf'
            printf(formatString);
#pragma warning(default : 6387)

        }
        break;
    case 5: // new operator fault
        {
            // Cause memory allocation error
            RecurseAlloc();
        }
        break;
    case 6: // SIGABRT
        {
            // Call abort
            abort();
        }
        break;
    case 7: // SIGFPE
        {
            // floating point exception ( /fp:except compiler option)
            sigfpe_test();
        }
        break;
    case 8: // SIGILL
        {
            raise(SIGILL);
        }
        break;
```

```
    case 9: // SIGINT
        {
            raise(SIGINT);
        }
        break;
    case 10: // SIGSEGV
        {
            raise(SIGSEGV);
        }
        break;
    case 11: // SIGTERM
        {
            raise(SIGTERM);
        }
        break;
    case 12: // RaiseException
        {
            // Raise noncontinuable software exception
            RaiseException(123, EXCEPTION_NONCONTINUABLE, 0, NULL);
        }
        break;
    case 13: // throw
        {
            // Throw typed C++ exception.
            throw 13;
        }
        break;
    default:
        {
            printf("Unknown exception type specified.");
            _getch();
        }
        break;
    }
}
```

CrashHandler.h 和 *CrashHandler.cpp*包含了异常处理函数的实现和崩溃后 minidump 文件的生成相关代码，下面贴出来的是这个类的定义。

```
class CCrashHandler
{
public:

    // Constructor
    CCrashHandler();
```

```
    // Destructor
    virtual ~CCrashHandler();

    // Sets exception handlers that work on per-process basis
    void SetProcessExceptionHandlers();

    // Installs C++ exception handlers that function on per-thread basis
    void SetThreadExceptionHandlers();

    // Collects current process state.
    static void GetExceptionPointers(
        DWORD dwExceptionCode,
        EXCEPTION_POINTERS** pExceptionPointers);

    // This method creates minidump of the process
    static void CreateMiniDump(EXCEPTION_POINTERS* pExcPtrs);

    /* Exception handler functions. */

    static LONG WINAPI SehHandler(PEXCEPTION_POINTERS pExceptionPtrs);
    static void __cdecl TerminateHandler();
    static void __cdecl UnexpectedHandler();

    static void __cdecl PureCallHandler();

    static void __cdecl InvalidParameterHandler(const wchar_t* expression,
          const wchar_t* function, const wchar_t* file,
          unsigned int line, uintptr_t pReserved);

    static int __cdecl NewHandler(size_t);

    static void SigabrtHandler(int);
    static void SigfpeHandler(int /*code*/, int subcode);
    static void SigintHandler(int);
    static void SigillHandler(int);
    static void SigsegvHandler(int);
    static void SigtermHandler(int);
};
```

正如你从上面的代码中看到的，CCrashHandler 类有两个用来设置异常处理函数的函数：SetProcessExceptionHandlers()和 SetThreadExceptionHandlers()，分别用来为整个程序与当前线程分别设置设置异常处理函数。这两个函数的实现代码如下：

```
void CCrashHandler::SetProcessExceptionHandlers()
{
    // Install top-level SEH handler
    SetUnhandledExceptionFilter(SehHandler);

    // Catch pure virtual function calls.
    // Because there is one _purecall_handler for the whole process,
    // calling this function immediately impacts all threads. The last
    // caller on any thread sets the handler.
    // http://msdn.microsoft.com/en-us/library/t296ys27.aspx
    _set_purecall_handler(PureCallHandler);

    // Catch new operator memory allocation exceptions
    _set_new_handler(NewHandler);

    // Catch invalid parameter exceptions.
    _set_invalid_parameter_handler(InvalidParameterHandler);

    // Set up C++ signal handlers

    _set_abort_behavior(_CALL_REPORTFAULT, _CALL_REPORTFAULT);

    // Catch an abnormal program termination
    signal(SIGABRT, SigabrtHandler);

    // Catch illegal instruction handler
    signal(SIGINT, SigintHandler);

    // Catch a termination request
    signal(SIGTERM, SigtermHandler);
}

void CCrashHandler::SetThreadExceptionHandlers()
{

    // Catch terminate() calls.
    // In a multithreaded environment, terminate functions are maintained
    // separately for each thread. Each new thread needs to install its own
    // terminate function. Thus, each thread is in charge of its own termination handling.
    // http://msdn.microsoft.com/en-us/library/t6fk7h29.aspx
    set_terminate(TerminateHandler);

    // Catch unexpected() calls.
    // In a multithreaded environment, unexpected functions are maintained
```

```
    // separately for each thread. Each new thread needs to install its own
    // unexpected function. Thus, each thread is in charge of its own unexpected handling.
    // http://msdn.microsoft.com/en-us/library/h46t5b69.aspx
    set_unexpected(UnexpectedHandler);

    // Catch a floating point error
    typedef void (*sigh)(int);
    signal(SIGFPE, (sigh)SigfpeHandler);

    // Catch an illegal instruction
    signal(SIGILL, SigillHandler);

    // Catch illegal storage access errors
    signal(SIGSEGV, SigsegvHandler);

}
```

在类的声明中，你还可以看到用到的几个异常处理函数，比如 SehHandler()，TerminateHandler()等等，这些函数都会在相应的异常产生时被调用。每个处理函数都分别会获取异常信息并且调用生成崩溃时 minidump 的代码，然后通过 TerminateProcess()来终止程序。

GetExceptionPointers()静态函数用来获取异常信息，我已经在上面的“获取异常信息”这一章节中描述过它是如何工作的。

CreateMiniDump()函数是用来产生崩溃时的 minidump 文件的。它需要一个 EXCEPTION_POINTERS 结构的指针。这个函数会调用微软 Debug Help Library（dbghelp.dll）里面的 MiniDumpWriteDump()函数来产生一个 minidump 文件。这个函数的代码如下。

```
// This method creates minidump of the process
void CCrashHandler::CreateMiniDump(EXCEPTION_POINTERS* pExcPtrs)
{
    HMODULE hDbgHelp = NULL;
    HANDLE hFile = NULL;
    MINIDUMP_EXCEPTION_INFORMATION mei;
    MINIDUMP_CALLBACK_INFORMATION mci;

    // Load dbghelp.dll
    hDbgHelp = LoadLibrary(_T("dbghelp.dll"));
    if(hDbgHelp==NULL)
    {
        // Error - couldn't load dbghelp.dll
        return;
    }

    // Create the minidump file
```

```
hFile = CreateFile(
    _T("crashdump.dmp"),
    GENERIC_WRITE,
    0,
    NULL,
    CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL,
    NULL);

if(hFile==INVALID_HANDLE_VALUE)
{
    // Couldn't create file
    return;
}

// Write minidump to the file
mei.ThreadId = GetCurrentThreadId();
mei.ExceptionPointers = pExcPtrs;
mei.ClientPointers = FALSE;
mci.CallbackRoutine = NULL;
mci.CallbackParam = NULL;

typedef BOOL (WINAPI *LPMINIDUMPWRITEDUMP)(
    HANDLE hProcess,
    DWORD ProcessId,
    HANDLE hFile,
    MINIDUMP_TYPE DumpType,
    CONST PMINIDUMP_EXCEPTION_INFORMATION ExceptionParam,
    CONST PMINIDUMP_USER_STREAM_INFORMATION UserEncoderParam,
    CONST PMINIDUMP_CALLBACK_INFORMATION CallbackParam);

LPMINIDUMPWRITEDUMP pfnMiniDumpWriteDump =
    (LPMINIDUMPWRITEDUMP)GetProcAddress(hDbgHelp, "MiniDumpWriteDump");
if(!pfnMiniDumpWriteDump)
{
    // Bad MiniDumpWriteDump function
    return;
}

HANDLE hProcess = GetCurrentProcess();
DWORD dwProcessId = GetCurrentProcessId();

BOOL bWriteDump = pfnMiniDumpWriteDump(
    hProcess,
```

```
        dwProcessId,
        hFile,
        MiniDumpNormal,
        &mei,
        NULL,
        &mci);

    if(!bWriteDump)
    {
        // Error writing dump.
        return;
    }

    // Close file
    CloseHandle(hFile);

    // Unload dbghelp.dll
    FreeLibrary(hDbgHelp);
}
```

参考文献

- Exception Handling in Visual C++
- SetUnhandledExceptionFilter()
- RaiseException()
- GetExceptionInformation()
- GetExceptionCode()
- set_terminate()
- set_unexpected()
- _set_se_translator()
- _set_security_error_handler()
- _set_purecall_handler()
- _set_new_handler()
- _set_invalid_parameter_handler()
- signal()
- raise()
- /EH (Exception Handling Model)
- /fp (Specify Floating-Point Behavior)
- /GS (Buffer Security Check)
- C Run-time Libraries (CRT)
- Under the Hood: New Vectored Exception Handling in Windows XP
- Vectored Exception Handling

- AddVectoredExceptionHandler Function
- RemoveVectoredExceptionHandler Function
- RaiseFailFastException Function

History

- 7 June 2011 - Initial release.
- 10 June 2011 - Added Visual C++ compiler flags and the References section.
- 6 January 2012 - Added ExceptionHandler sample code and added the Using the Code section.

License

About the Author

zexspectrum
Russian Federation
Member

No Biography provided