Windows Memory leak detection (update to existing article)

Tim Stevens, 2 Mar 2011 CPOL

★★★★★ 5.00 (12 votes)
☆☆☆☆☆

An updated version of the code in David A Jones' article "Memory Leak Detection"

Updated header file MemLeakDetect.h and Updated MemLeakDetect.cpp

David A Jones wrote a very good Code Project article & code in 2004 athttp://www.codeproject.com/KB/cpp/MemLeakDetect.aspx?msg=3102545#xx3102545xx and the following notes suppliment his original article. I can't update that article, so have posted the changed code in this rather long tip!

David is no longer active on Code Project, however several people have contributed updates to the original code. Changes from David's version:

Code now compiles as Unicode and MBCS for both 32 & 64 bit builds.

Tested with Visual Studio 2008, but should compile with VS2010, and possibly with 2005.

Bug which could cause garbage in the debug dump fixed.

Writes log files to user's %TEMP% directory (and clears old ones) in addition to outputting to the debugger output window.

Disabled by default in Release builds (although it is simple to re-enable this).

Changes by:

Doug Rogers (Win64 port)

Igor Jambrek (back-port to Visual Studio 2005 (although Tim may have inadvertently subsequently broken this; he can't test it), file logging)
OfekSH (bugfix)

Tim Stevens (Unicode/MBCS, bugfixes, tidy up).

Credit for the article remains with the original author.

Updated header file *MemLeakDetect.h*

```
/*************************************************************
 Author          : David A. Jones
 File Name       : MemLeakDetect.h
 Date            : July 30, 2004
 Synopsis :
                           A trace memory feature for source code to trace and
                           find memory related bugs.
 Future           :
                                      1) Memory corruption
                                      2) Freeing memory without allocating
                                      3) Freeing memory twice
                                      4) Not Freeing memory at all
                                      5) over running memory boundardies
                   July 2009: Tim Stevens (UNICODE/ANSI 32 bit only, more secure CRT with VS 2008).
                   Feb  2011: Doug Rogers, Igor Jambrek, OfekSH & tim. (Compiles as 64 & 32 bit).
                   Based on http://www.codeproject.com/cpp/MemLeakDetect.asp
****************************************************************/
/*
Compiles clean in Visual Studio 2008 SP1 in 32 & 64 UNICODE and MultiByte builds.
By default, disabled in Release mode, since it relies on the Debug MS
Runtime DLLs, the licence terms of which only allow redistribution in
Release mode. However, if you do want to use it in Release mode, then comment out the
"#ifdef _DEBUG" lines that guard the complete MemLeakDetect.h & .cpp files,
and link against the Debug runtimes
(e.g. /MTd instead of /MT) in Release mode.
Please don't use precompiled headers for this file.
To catch most malloc/free or new/delete leaks, simply add this
block of code (& #define MEMORY_LEAK_CHECK)
at the application level:
#ifdef _DEBUG
         #ifdef MEMORY_LEAK_CHECK
                  #include "MemLeakDetect.h"
                  static CMemLeakDetect memLeakDetect;
         #endif
#endif
A typical leak might be:
         int *pfoo = new int[1000];
Then forgetting to do
         delete [] pfoo;
Then when running under a debugger, if there is a leak, you'll get this kind of
output in the Output pane.
You'll also get files with names like "mldetector-(AppName.exe)_Feb16-2011__21-53-43.log"
written to your %TEMP% directory:
Memory Leak(1)------------------->
Memory Leak <0xBC> bytes(86) occurance(0)
c:\code\ta2svn\sandbox\pjh\software\common\memleakdetect.cpp(201):
0x0044B7C3->CMemLeakDetect::addMemoryTrace()
c:\code\ta2svn\sandbox\pjh\software\common\memleakdetect.cpp(140): 0x0044B4B2->catchMemoryAllocHook()
0x0012D874->_malloc_dbg()
0x0012D874->_malloc_dbg()
0x0012D874->_malloc_dbg()
0x0012D874->malloc()
0x0012D874->??2@YAPAXI@Z()
f:\dd\vctools\crt_bld\self_x86\crt\src\newaop.cpp(7): 0x004B4D1E->operator new[]()
c:\code\ta2svn\sandbox\pjh\software\hw_app\hw_app.cpp(145): 0x00442276->wmain()
```

```
f:\dd\vctools\crt_bld\self_x86\crt\src\crtexe.c(579): 0x004B56C8->__tmainCRTStartup()
f:\dd\vctools\crt_bld\self_x86\crt\src\crtexe.c(399): 0x004B550F->wmainCRTStartup()
0x0012D874->RegisterWaitForInputIdle()
-----------------------------------------------------------
Total 1 Memory Leaks: 86 bytes Total Alocations 276
You can then double-click in the Output pane on the leak ((145) in the example above) and be taken to the source
line
which caused the leak.
*/
#if !defined(MEMLEAKDETECT_H)
#define MEMLEAKDETECT_H
#ifdef _DEBUG
#define _CRTDBG_MAP_ALLOC
#include <map>
#define _CRTBLD
#include <windows.h>
#include <..\crt\src\dbgint.h>
#include <imagehlp.h>
#include <crtdbg.h>
#pragma comment( lib, "imagehlp.lib" )
using namespace std;
// if you want to use the custom stackwalker otherwise
// comment this line out
#define MLD_CUSTOMSTACKWALK                    1
//
#define MLD_MAX_NAME_LENGTH                    256
#define MLD_MAX_TRACEINFO                      63
#define MLD_TRACEINFO_EMPTY                    _T("")
#define MLD_TRACEINFO_NOSYMBOL                 _T("?(?)")
#ifdef  MLD_CUSTOMSTACKWALK
#define MLD_STACKWALKER                                 symStackTrace2
#else
#define MLD_STACKWALKER                                 symStackTrace
#endif
#define AfxTrace MyTrace
#ifndef _WIN64
typedef DWORD ADDR;
typedef PIMAGEHLP_SYMBOL IMAGE_SYM;
typedef IMAGEHLP_LINE IMAGE_LN;
#else
typedef DWORD64 ADDR;
typedef PIMAGEHLP_SYMBOL64 IMAGE_SYM;
typedef IMAGEHLP_LINE64 IMAGE_LN;
#endif
class CMemLeakDetect
{
         public:
                  typedef struct    {
                                    ADDRESS                              addrPC;
```

```
                        ADDRESS                             addrFrame;

                } STACKFRAMEENTRY;
                typedef struct tagAllocBlockInfo
                {
                        //      Added constructor to zero memory - thanks to bugfix from OfekSH.
                                tagAllocBlockInfo() { ZeroMemory(traceinfo, sizeof(traceinfo) ); }
                                void*                               address;
                                size_t                              size;
                                TCHAR                               fileName[MLD_MAX_NAME_LENGTH];
                                DWORD                               lineNumber;
                                DWORD                               occurance;
                                STACKFRAMEENTRY             traceinfo[MLD_MAX_TRACEINFO];
                } AllocBlockInfo;
                //typedef int POSITION;
                typedef map<lpvoid,>                                KEYMAP;
                typedef map<lpvoid,>::iterator      POSITION;
                typedef pair<lpvoid,>                       KEYVALUE;
                class CMapMem
                {
                        public:
                                KEYMAP                      m_Map;
                                POSITION            m_Pos;
                                inline BOOL Lookup(LPVOID pAddr,  AllocBlockInfo& aInfo) {
                                        m_Pos = m_Map.find(pAddr);
                                        //
                                        if (m_Pos == m_Map.end())
                                        {
                                                return FALSE;
                                        }
                                        //
                                        pAddr = m_Pos->first;
                                        aInfo = m_Pos->second;
                                        return TRUE;
                                };
                                inline POSITION end() {
                                        return m_Map.end();
                                };
                                inline void RemoveKey(LPVOID pAddr) {

                                        m_Map.erase(pAddr);
                                };
                                inline void RemoveAll() {
                                        m_Map.clear();
                                };
                                void SetAt(LPVOID pAddr, AllocBlockInfo& aInfo) {
                                        m_Map[pAddr] = aInfo;
                                };
                                inline POSITION GetStartPosition() {
                                        POSITION pos = m_Map.begin();
                                        return pos;
                                };
                                inline void GetNextAssoc(POSITION& pos, LPVOID& rAddr, AllocBlockInfo& 
aInfo) {
                                        rAddr = pos->first;
                                        aInfo = pos->second;
                                        pos++;
                                };
                                void InitHashTable(int preAllocEntries, BOOL flag)     {
                                         preAllocEntries   = NULL;
                                         flag                                = NULL;
                                };
                };
                CMemLeakDetect();
                ~CMemLeakDetect();
                void Init();
                void End();
                void addMemoryTrace(void* addr,  size_t asize,  TCHAR *fname, DWORD lnum);
                void redoMemoryTrace(void* addr,  void* oldaddr, size_t asize,  TCHAR *fname, DWORD lnum);
                void removeMemoryTrace(void* addr, void* realdataptr);
                void cleanupMemoryTrace();
                void dumpMemoryTrace();
                //
                //CMap<lpvoid,> m_AllocatedMemoryList;
                CMapMem                     m_AllocatedMemoryList;
        DWORD memoccurance;
        bool  isLocked;
        //
        private:
                typedef USHORT (WINAPI *CaptureStackBackTraceType)(__in ULONG, __in ULONG, __out PVOID*, 
__out_opt PULONG);
                HMODULE m_k32;
                CaptureStackBackTraceType m_func;
                BOOL initSymInfo(TCHAR* lpUserPath);
                BOOL cleanupSymInfo();
                void symbolPaths( TCHAR* lpszSymbolPaths);
                void symStackTrace(STACKFRAMEENTRY* pStacktrace);
                void symStackTrace2(STACKFRAMEENTRY* pStacktrace);
                BOOL symFunctionInfoFromAddresses(ADDR fnAddress, ADDR stackAddress, TCHAR *lpszSymbol, 
UINT BufSizeTCHARs);
                BOOL symSourceInfoFromAddress(ADDR address, TCHAR* lpszSourceInfo);
                BOOL symModuleNameFromAddress(ADDR address, TCHAR* lpszModule);
                HANDLE                              m_hProcess;
                PIMAGEHLP_SYMBOL   m_pSymbol;
                DWORD                               m_dwsymBufSize;
};
#endif
```

```
#endif
```

Updated *MemLeakDetect.cpp* file:

```
/*************************************************************
 Author            : David A. Jones
 File Name         : MemLeakDetect.h
 Date              : July 30, 2004
 Synopsis
                            A trace memory feature for source code to trace and
                            find memory related bugs.

****************************************************************/
// See MemLeakDetect.h for full history.
// Based on http://www.codeproject.com/cpp/MemLeakDetect.asp
#ifdef _DEBUG
#include <tchar.h>
#include "MemLeakDetect.h"
#include <fstream>
#include <time.h>
#include <psapi.h>                                     // Only needed for GetModuleBaseName().
#pragma comment(lib, "Psapi.lib")    // Only needed for GetModuleBaseName().
//#pragma warning(disable:4312)      // 'type cast' : conversion from 'long' to 'void *' of greater size
//#pragma warning(disable:4313)
//#pragma warning(disable:4267)
#pragma warning(disable:4100)        // Unreferenced formal parameter.
static CMemLeakDetect*      g_pMemTrace                          = NULL;
static _CRT_ALLOC_HOOK      pfnOldCrtAllocHook = NULL;
static int catchMemoryAllocHook(int  allocType,
                                                        void     *userData,
                                                        size_t size,
                                                        int      blockType,
                                                        long     requestNumber,
                    const unsigned char       *filename, // Can't be UNICODE
                                                        int      lineNumber) ;
static int MyTrace(LPCTSTR lpszFormat, ...);

static void DeleteOldTempFiles(const TCHAR dir[], const TCHAR type[], int DaysAge);

static int MyTrace(LPCTSTR lpszFormat, ...)
{
         va_list args;
         va_start( args, lpszFormat);
#ifndef UNICODE
         char buffer[1024];
         vsprintf_s( buffer, lpszFormat, args );
         return _CrtDbgReport(_CRT_WARN,NULL,NULL,NULL,buffer);
#else
```

```
         TCHAR buffer[1024];
         vswprintf_s( buffer, lpszFormat, args );
         char fmtbuf[1024] ;
         WideCharToMultiByte(CP_ACP, 0, buffer, -1,
                  fmtbuf, 1024, NULL, NULL ) ;
         return _CrtDbgReport(_CRT_WARN,NULL,NULL,NULL,fmtbuf);
#endif
}
static int catchMemoryAllocHook(int  allocType,
                                                        void    *userData,
                                                        size_t size,
                                                        int     blockType,
                                                        long    requestNumber,
                    const unsigned char       *filename,  // Can't be UNICODE
                                                        int     lineNumber)
{
         _CrtMemBlockHeader *pCrtHead;
         long prevRequestNumber;
#ifdef UNICODE
         wchar_t Wname[1024] ;
         Wname[0] = L'\0' ;
#endif
         // internal C library internal allocations
         if ( blockType == _CRT_BLOCK )
         {
                  return( TRUE );
         }
         // check if someone has turned off mem tracing
         if  ((( _CRTDBG_ALLOC_MEM_DF & _crtDbgFlag) == 0) &&
                  (( allocType                          == _HOOK_ALLOC)            ||
                           ( allocType                          == _HOOK_REALLOC)))
         {
                  if (pfnOldCrtAllocHook)
                  {
                           pfnOldCrtAllocHook(allocType, userData, size, blockType, requestNumber,
filename, lineNumber);
                  }
                  return TRUE;
         }
         // protect if mem trace is not initialized
         if (g_pMemTrace == NULL)
         {
                  if (pfnOldCrtAllocHook)
                  {
                           pfnOldCrtAllocHook(allocType, userData, size, blockType, requestNumber,
filename, lineNumber);
                  }
                  return TRUE;
         }
```

```
		// protect internal mem trace allocs
		if (g_pMemTrace->isLocked)
		{
			if (pfnOldCrtAllocHook)
			{
				pfnOldCrtAllocHook(allocType, userData, size, blockType, requestNumber, 
filename, lineNumber);
			}
			return( TRUE);
		}
		// lock the function
		g_pMemTrace->isLocked = true;
		//
#ifdef UNICODE
		int len ;
		if (NULL != filename)
		{
			len = (int)strlen((char *)filename) + 1 ;
			MultiByteToWideChar(CP_ACP, 0, (char *)filename, len, Wname, len) ;
		}
		else
			len = 0 ;
#else
		#define Wname (char*)filename
#endif
		if (allocType == _HOOK_ALLOC)
		{
			g_pMemTrace->addMemoryTrace((void *) requestNumber, size, Wname, lineNumber);
		}
		else
		if (allocType == _HOOK_REALLOC)
		{
			if (_CrtIsValidHeapPointer(userData))
			{
				pCrtHead = pHdr(userData);
				prevRequestNumber = pCrtHead->lRequest;
				//
				if (pCrtHead->nBlockUse == _IGNORE_BLOCK)
				{
					if (pfnOldCrtAllocHook)
					{
						pfnOldCrtAllocHook(allocType, userData, size, blockType, 
requestNumber, filename, lineNumber);
					}
					goto END;
				}
				g_pMemTrace->redoMemoryTrace((void *) requestNumber, (void *) prevRequestNumber, 
size, Wname, lineNumber);
			}
```

```
         }
         else
         if (allocType == _HOOK_FREE)
         {
                   if (_CrtIsValidHeapPointer(userData))
                   {
                            pCrtHead = pHdr(userData);
                            requestNumber = pCrtHead->lRequest;
                            //
                            if (pCrtHead->nBlockUse == _IGNORE_BLOCK)
                            {
                                     if (pfnOldCrtAllocHook)
                                     {
                                               pfnOldCrtAllocHook(allocType, userData, size, blockType,
requestNumber, filename, lineNumber);
                                     }
                                     goto END;
                            }
                            g_pMemTrace->removeMemoryTrace((void *) requestNumber, userData);
                   }
         }
END:
         // unlock the function
         g_pMemTrace->isLocked = false;
         return TRUE;
}
void CMemLeakDetect::addMemoryTrace(void* addr,  size_t asize,  TCHAR *fname, DWORD lnum)
{
         AllocBlockInfo ainfo;
         //
         if (m_AllocatedMemoryList.Lookup(addr, ainfo))
         {
                   // already allocated
                   AfxTrace(_T("ERROR!CMemLeakDetect::addMemoryTrace() Address(0x%p) already allocated\n"),
addr);
                   return;
         }
         //
         ainfo.address                = addr;
         ainfo.lineNumber    = lnum;
         ainfo.size                            = asize;
         ainfo.occurance              = memoccurance++;
         MLD_STACKWALKER(&ainfo.traceinfo[0]);
         //
         if (fname)
                   _tcsncpy_s(&ainfo.fileName[0], MLD_MAX_NAME_LENGTH, fname, MLD_MAX_NAME_LENGTH);
         else
           ainfo.fileName[0] = 0;
         //
         m_AllocatedMemoryList.SetAt(addr, ainfo);
};
void CMemLeakDetect::redoMemoryTrace(void* addr,  void* oldaddr, size_t asize,  TCHAR *fname, DWORD lnum)
{
         AllocBlockInfo ainfo;
         if (m_AllocatedMemoryList.Lookup(oldaddr,(AllocBlockInfo &) ainfo))
         {
                  m_AllocatedMemoryList.RemoveKey(oldaddr);
         }
         else
         {
                  AfxTrace(_T("ERROR!CMemLeakDetect::redoMemoryTrace() didnt find Address(0x%08X) to
free\n"), oldaddr);
         }
         //
         ainfo.address               = addr;
         ainfo.lineNumber   = lnum;
         ainfo.size                           = asize;
         ainfo.occurance             = memoccurance++;
         MLD_STACKWALKER(&ainfo.traceinfo[0]);
         //
         if (fname)
                  _tcsncpy_s(&ainfo.fileName[0], MLD_MAX_NAME_LENGTH, fname, MLD_MAX_NAME_LENGTH);
         else
           ainfo.fileName[0] = 0;
         m_AllocatedMemoryList.SetAt(addr, ainfo);
};
void CMemLeakDetect::removeMemoryTrace(void* addr, void* realdataptr)
{
         AllocBlockInfo ainfo;
         //
         if (m_AllocatedMemoryList.Lookup(addr,(AllocBlockInfo &) ainfo))
         {
                  m_AllocatedMemoryList.RemoveKey(addr);
         }
         else
         {
            //freeing unallocated memory
                  AfxTrace(_T("ERROR!CMemLeakDetect::removeMemoryTrace() didnt find Address(0x%08X) to
free\n"), addr);
         }
};
void CMemLeakDetect::cleanupMemoryTrace()
{
         m_AllocatedMemoryList.RemoveAll();
};
void CMemLeakDetect::dumpMemoryTrace()
{
         POSITION                    pos;
         LPVOID                                 addr;
         AllocBlockInfo              ainfo;
         TCHAR                                  buf[MLD_MAX_NAME_LENGTH];
         TCHAR                                  fileName[MLD_MAX_NAME_LENGTH];
         TCHAR                                  symInfo[MLD_MAX_NAME_LENGTH];
         TCHAR                                  srcInfo[MLD_MAX_NAME_LENGTH];
         size_t                                 totalSize                                               = 0;
         int                                             numLeaks                                                 =
         STACKFRAMEENTRY*   p                                                                           = 0;
         ofstream myfile;
#ifdef UNICODE
                  char dest[1024] ;
#endif
         struct tm timeinfo;
         __time64_t long_time;
         _time64(&long_time);
         // Convert to local time.
         _localtime64_s(&timeinfo, &long_time);
         TCHAR TempDir[MAX_PATH];
         TCHAR ProcName[MAX_PATH];
         GetTempPath(MAX_PATH, TempDir);
         ProcName[0] = _T('\0');
         GetModuleBaseName(GetCurrentProcess(), NULL, ProcName, sizeof(ProcName)/sizeof(TCHAR));
         _stprintf_s(fileName, MLD_MAX_NAME_LENGTH, _T("%smldetector-(%s)_"), TempDir, ProcName);
         _tcsftime(buf,MLD_MAX_NAME_LENGTH, _T("%b%d-%Y__%H-%M-%S.log"),&timeinfo);

         _tcscat_s(fileName,MLD_MAX_NAME_LENGTH, buf);

         myfile.open (fileName);

         DeleteOldTempFiles(TempDir, _T("mldetector-(*.log"), 7);
         //
         _tcscpy_s(symInfo, MLD_MAX_NAME_LENGTH, MLD_TRACEINFO_NOSYMBOL);
         _tcscpy_s(srcInfo, MLD_MAX_NAME_LENGTH, MLD_TRACEINFO_NOSYMBOL);
         //
         pos = m_AllocatedMemoryList.GetStartPosition();
         //
         while(pos != m_AllocatedMemoryList.end())
         {
                  numLeaks++;
                  _stprintf_s(buf, MLD_MAX_NAME_LENGTH, _T("Memory Leak(%d)------------------->\n"),
numLeaks);
                  AfxTrace(buf);
#ifdef UNICODE
                  WideCharToMultiByte( CP_ACP, 0, buf, -1, dest, 1024, NULL, NULL );
                  myfile << dest;
#else
                  myfile << buf;
#endif
```

```
                   //
                   m_AllocatedMemoryList.GetNextAssoc(pos, (LPVOID &) addr, (AllocBlockInfo&) ainfo);
                   if (ainfo.fileName[0] != NULL)
                   {
                            _stprintf_s(buf, MLD_MAX_NAME_LENGTH, _T("Memory Leak <0x%p> bytes(%d)
occurance(%d) %s(%d)\n"),
                                               ainfo.address, ainfo.size, ainfo.occurance, ainfo.fileName,
ainfo.lineNumber);
                   }
                   else
                   {
                            _stprintf_s(buf, MLD_MAX_NAME_LENGTH, _T("Memory Leak <0x%p> bytes(%d)
occurance(%d)\n"),
                                               ainfo.address, ainfo.size, ainfo.occurance);
                   }
                   //
                   AfxTrace(buf);
#ifdef UNICODE
                   WideCharToMultiByte( CP_ACP, 0, buf, -1, dest, 1024, NULL, NULL );
                   myfile << dest;
#else
                   myfile << buf;
#endif
                   //
                   p = &ainfo.traceinfo[0];
                   while(p[0].addrPC.Offset)
                   {
                            symFunctionInfoFromAddresses( p[0].addrPC.Offset, p[0].addrFrame.Offset,
symInfo, MLD_MAX_NAME_LENGTH);
                            symSourceInfoFromAddress(     p[0].addrPC.Offset, srcInfo );
                            _stprintf_s(buf, MLD_MAX_NAME_LENGTH, _T("%s->%s()\n"), srcInfo, symInfo);
                            AfxTrace(_T("%s->%s()\n"), srcInfo, symInfo);
#ifdef UNICODE
                   WideCharToMultiByte( CP_ACP, 0, buf, -1, dest, 1024, NULL, NULL );
                   myfile << dest;
#else
                   myfile << buf;
#endif
                            p++;
                   }
                   totalSize += ainfo.size;
          }
          _stprintf_s(buf, MLD_MAX_NAME_LENGTH,
_T("\n-----------------------------------------------------------\n"));
          AfxTrace(buf);
#ifdef UNICODE
                   WideCharToMultiByte( CP_ACP, 0, buf, -1, dest, 1024, NULL, NULL );
                   myfile << dest;
#else
```

```
                   myfile << buf;
#endif
         if(!totalSize)
         {
                   _stprintf_s(buf, MLD_MAX_NAME_LENGTH, _T("No Memory Leaks Detected for %d
Allocations\n\n"), memoccurance);
                   AfxTrace(buf);
#ifdef UNICODE
                   WideCharToMultiByte( CP_ACP, 0, buf, -1, dest, 1024, NULL, NULL );
                   myfile << dest;
#else
                   myfile << buf;
#endif
         }
         else
         {
                   _stprintf_s(buf, MLD_MAX_NAME_LENGTH, _T("Total %d Memory Leaks: %d bytes Total
Alocations %d\n\n"), numLeaks, totalSize, memoccurance);
         }
         AfxTrace(buf);
#ifdef UNICODE
         WideCharToMultiByte( CP_ACP, 0, buf, -1, dest, 1024, NULL, NULL );
         const TCHAR *umb = _T("Unicode");
         myfile << dest;
#else
         myfile << buf;
         const TCHAR *umb = _T("Multibyte");
#endif
#ifdef _WIN64
         const TCHAR *w64 = _T("64 bit");
#else
         const TCHAR *w64 = _T("32 bit");
#endif
#ifdef NDEBUG
         const TCHAR *dbg = _T("release build.");
#else
         const TCHAR *dbg = _T("debug build.");
#endif
         _stprintf_s(TempDir, MAX_PATH, _T("%s %s %s\n"), umb, w64, dbg);
#ifdef UNICODE
         WideCharToMultiByte( CP_ACP, 0, TempDir, -1, dest, 1024, NULL, NULL );
         myfile << dest;
         AfxTrace(TempDir);
#else
         myfile << TempDir;
         AfxTrace(TempDir);
#endif
         myfile.close();
}
```

```
void CMemLeakDetect::Init()
{
         m_func = (CaptureStackBackTraceType)(GetProcAddress( m_k32 = LoadLibrary(_T("kernel32.dll")),
"RtlCaptureStackBackTrace"));
         m_dwsymBufSize              = (MLD_MAX_NAME_LENGTH + sizeof(PIMAGEHLP_SYMBOL));
         m_hProcess                           = GetCurrentProcess();
         m_pSymbol                            = (IMAGE_SYM)GlobalAlloc( GMEM_FIXED, m_dwsymBufSize);
         m_AllocatedMemoryList.InitHashTable(10211, TRUE);
         initSymInfo( NULL );
         isLocked                             = false;
         g_pMemTrace                          = this;
         pfnOldCrtAllocHook = _CrtSetAllocHook( catchMemoryAllocHook );
}
void CMemLeakDetect::End()
{
         isLocked                             = true;
         _CrtSetAllocHook(pfnOldCrtAllocHook);
         dumpMemoryTrace();
         cleanupMemoryTrace();
         cleanupSymInfo();
         GlobalFree(m_pSymbol);
         g_pMemTrace                                   = NULL;
         FreeLibrary(m_k32);
}
CMemLeakDetect::CMemLeakDetect()
{
         Init();
}
CMemLeakDetect::~CMemLeakDetect()
{
         End();
}
// PRIVATE STUFF
void CMemLeakDetect::symbolPaths( TCHAR* lpszSymbolPath)
{
         TCHAR lpszPath[MLD_MAX_NAME_LENGTH];
   // Creating the default path where the dgbhelp.dll is located
   // ".;%_NT_SYMBOL_PATH%;%_NT_ALTERNATE_SYMBOL_PATH%;%SYSTEMROOT%;%SYSTEMROOT%\System32;"
         _tcscpy_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T(".;..\\;..\\..\\"));
         // environment variable _NT_SYMBOL_PATH
         if ( GetEnvironmentVariable(_T("_NT_SYMBOL_PATH"), lpszPath, MLD_MAX_NAME_LENGTH ))
         {
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T(";"));
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, lpszPath );
         }
         // environment variable _NT_ALTERNATE_SYMBOL_PATH
         if ( GetEnvironmentVariable( _T("_NT_ALTERNATE_SYMBOL_PATH"), lpszPath, MLD_MAX_NAME_LENGTH ))
         {
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T(";"));
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, lpszPath );
         }
         // environment variable SYSTEMROOT
         if ( GetEnvironmentVariable( _T("SYSTEMROOT"), lpszPath, MLD_MAX_NAME_LENGTH ) )
         {
            _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T(";"));
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, lpszPath);
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T(";"));
                  // SYSTEMROOT\System32
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, lpszPath );
                  _tcscat_s( lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T("\\System32"));
         }
}
BOOL CMemLeakDetect::cleanupSymInfo()
{
         return SymCleanup( GetCurrentProcess() );
}
// Initializes the symbol files
BOOL CMemLeakDetect::initSymInfo( TCHAR* lpszUserSymbolPath )
{
         TCHAR   lpszSymbolPath[MLD_MAX_NAME_LENGTH];
    DWORD   symOptions = SymGetOptions();
         symOptions |= SYMOPT_LOAD_LINES;
         symOptions &= ~SYMOPT_UNDNAME;
         SymSetOptions( symOptions );
    // Get the search path for the symbol files
         symbolPaths( lpszSymbolPath);
         //
         if (lpszUserSymbolPath)
         {
                  _tcscat_s(lpszSymbolPath, MLD_MAX_NAME_LENGTH, _T(";"));
                  _tcscat_s(lpszSymbolPath, MLD_MAX_NAME_LENGTH, lpszUserSymbolPath);
         }
#ifdef UNICODE
         int len = (int)_tcslen(lpszSymbolPath) + 1 ;
         char dest[1024] ;
         WideCharToMultiByte( CP_ACP, 0, lpszSymbolPath, -1, dest, len, NULL, NULL );
         BOOL bret = SymInitialize( GetCurrentProcess(), dest, TRUE);
#else
         BOOL bret = SymInitialize( GetCurrentProcess(), lpszSymbolPath, TRUE) ;
#endif
         return bret;
}
/*void CMemLeakDetect::symStackTrace(STACKFRAMEENTRY* pStacktrace )
{
         STACKFRAME     callStack;
         BOOL          bResult;
         CONTEXT        context;
         HANDLE                hThread  = GetCurrentThread();
```

```
        // get the context
        memset( &context, NULL, sizeof(context) );
        context.ContextFlags = CONTEXT_FULL;
        if ( !GetThreadContext( hThread, &context ) )
        {
    //   AfxTrace("Call stack info(thread=0x%X) failed.\n", hThread );
           return;
        }
        //initialize the call stack
        memset( &callStack, NULL, sizeof(callStack) );
        callStack.AddrPC.Offset    = context.Eip;
        callStack.AddrStack.Offset = context.Esp;
        callStack.AddrFrame.Offset = context.Ebp;
        callStack.AddrPC.Mode      = AddrModeFlat;
        callStack.AddrStack.Mode   = AddrModeFlat;
        callStack.AddrFrame.Mode   = AddrModeFlat;
        //
        for( DWORD index = 0; index < MLD_MAX_TRACEINFO; index++ )
        {
                bResult = StackWalk(IMAGE_FILE_MACHINE_I386,
                                                        m_hProcess,
                                                        hThread,
                                                        &callStack,
                                                        NULL,
                                                        NULL,
                                                        SymFunctionTableAccess,
                                                        SymGetModuleBase,
                                                        NULL);


                //if ( index == 0 )
                 //  continue;
                if( !bResult || callStack.AddrFrame.Offset == 0 )
                        break;
                //
                pStacktrace[0].addrPC       = callStack.AddrPC;
                pStacktrace[0].addrFrame = callStack.AddrFrame;
                pStacktrace++;
        }
        //clear the last entry
        memset(pStacktrace, NULL, sizeof(STACKFRAMEENTRY));
}*/
//
// This code is still under investigation
// I have to test this code and make sure it is compatible
// with the other stack walker!
//
void CMemLeakDetect::symStackTrace2(STACKFRAMEENTRY* pStacktrace )
{
```

```
         long                        StackIndex                                    = 0;
         ADDR                        block[63];
         memset(block,0,sizeof(block));
         USHORT frames = (m_func)(3,59,(void**)block,NULL);
         for (int i = 0; i < frames ; i++)
         {
                  ADDR                        InstructionPtr = (ADDR)block[i];
                  pStacktrace[StackIndex].addrPC.Offset= InstructionPtr;
                  pStacktrace[StackIndex].addrPC.Segment        = NULL;
                  pStacktrace[StackIndex].addrPC.Mode           = AddrModeFlat;
                  //
                  StackIndex++;
         }
         pStacktrace[StackIndex].addrPC.Offset = 0;
         pStacktrace[StackIndex].addrPC.Segment = 0;
}
BOOL CMemLeakDetect::symFunctionInfoFromAddresses( ADDR fnAddress, ADDR stackAddress, TCHAR *lpszSymbol,

{
         ADDR            dwDisp      = 0;
         ::ZeroMemory(m_pSymbol, m_dwsymBufSize );
         m_pSymbol->SizeOfStruct              = sizeof(IMAGEHLP_LINE64);
         //m_pSymbol->MaxNameLength  = DWORD64 - sizeof(IMAGEHLP_SYMBOL64);
   // Set the default to unknown
         _tcscpy_s( lpszSymbol, MLD_MAX_NAME_LENGTH, MLD_TRACEINFO_NOSYMBOL);
         // Get symbol info for IP
         if ( SymGetSymFromAddr( m_hProcess, (ADDR)fnAddress, &dwDisp, m_pSymbol ) )
         {
#ifdef UNICODE
                  int len = (int)strlen(m_pSymbol->Name) + 1 ;
                  wchar_t dest[1024] ;
                  MultiByteToWideChar(CP_ACP, 0, m_pSymbol->Name, len, dest, len );
                  _tcscpy_s(lpszSymbol, BufSizeTCHARs, dest);
#else
                  _tcscpy_s(lpszSymbol, BufSizeTCHARs, m_pSymbol->Name);
#endif
                  return TRUE;
         }
         //create the symbol using the address because we have no symbol
         _stprintf_s(lpszSymbol, BufSizeTCHARs, _T("0x%08X"), fnAddress);
         return FALSE;
}
BOOL CMemLeakDetect::symSourceInfoFromAddress(ADDR address, TCHAR* lpszSourceInfo)
{
         BOOL           ret = FALSE;
         IMAGE_LN  lineInfo;
         DWORD          dwDisp;
         TCHAR          lpModuleInfo[MLD_MAX_NAME_LENGTH] = MLD_TRACEINFO_EMPTY;
         _tcscpy_s( lpszSourceInfo, MLD_MAX_NAME_LENGTH, MLD_TRACEINFO_NOSYMBOL);
```

```
          memset( &lineInfo, NULL, sizeof( IMAGEHLP_LINE ) );
          lineInfo.SizeOfStruct = sizeof( IMAGEHLP_LINE );
          if ( SymGetLineFromAddr( m_hProcess, address, &dwDisp, &lineInfo ) )
          {
             // Using the "sourcefile(linenumber)" format
#ifdef UNICODE
                    wchar_t dest[1024] ;
                    int len = (int)strlen((char *)lineInfo.FileName) + 1 ;
                    MultiByteToWideChar(CP_ACP, 0, (char *)lineInfo.FileName, len, dest, len) ;
                    _stprintf_s(lpszSourceInfo, MLD_MAX_NAME_LENGTH, _T("%s(%d): 0x%08X"), dest,
lineInfo.LineNumber, address );//    <--- Size of the char thing.
#else
                    _stprintf_s(lpszSourceInfo, MLD_MAX_NAME_LENGTH, _T("%s(%d): 0x%08X"), lineInfo.FileName,
lineInfo.LineNumber, address );//    <--- Size of the char thing.
#endif
                    ret = TRUE;
          }
          else
          {
        // Using the "modulename!address" format
                    symModuleNameFromAddress( address, lpModuleInfo );
                    if ( lpModuleInfo[0] == _T('?') || lpModuleInfo[0] == _T('\0'))
                    {
                              // Using the "address" format
                              _stprintf_s(lpszSourceInfo,MLD_MAX_NAME_LENGTH,  _T("0x%p"), lpModuleInfo,
address );          // Tim ???
                    }
                    else
                    {
                              _stprintf_s(lpszSourceInfo, MLD_MAX_NAME_LENGTH, _T("%sdll! 0x%08X"),
lpModuleInfo, address );
                    }
                    ret = FALSE;
          }
          //
          return ret;
}
BOOL CMemLeakDetect::symModuleNameFromAddress( ADDR address, TCHAR* lpszModule )
{
          BOOL             ret = FALSE;
          IMAGEHLP_MODULE   moduleInfo;
          ::ZeroMemory( &moduleInfo, sizeof(IMAGEHLP_MODULE) );
          moduleInfo.SizeOfStruct = sizeof(IMAGEHLP_MODULE);
          if ( SymGetModuleInfo( m_hProcess, (ADDR)address, &moduleInfo ) )
          {
                    // Note. IMAGEHLP_MODULE::ModuleName seems to be hardcoded as 32 char/wchar_t (VS2008).
#ifdef UNICODE
                    int len = (int)_tcslen(lpszModule) + 1 ;
                    char dest[1024] ;
                    WideCharToMultiByte( CP_ACP, 0, lpszModule, -1, dest, len, NULL, NULL );
                    strcpy_s(moduleInfo.ModuleName, 32, dest);     // bloody ANSI!
#else
                    strcpy_s(moduleInfo.ModuleName, 32, lpszModule);
#endif
                    ret = TRUE;
          }
          else
          {
                    _tcscpy_s( lpszModule, MLD_MAX_NAME_LENGTH, MLD_TRACEINFO_NOSYMBOL);
          }

          return ret;
}
static void DeleteOldTempFiles(const TCHAR dir[], const TCHAR type[], int days)
{
          union tu
          {
                    FILETIME fileTime;
                    ULARGE_INTEGER ul;
          };        // Seems simplest way to do the Win32 time manipulation.
          WIN32_FIND_DATA FindFileData;
          HANDLE hFind = INVALID_HANDLE_VALUE;

          TCHAR curdir[MAX_PATH];
          GetCurrentDirectory(MAX_PATH, curdir);         // Ignoring failure!
          SetCurrentDirectory(dir);

          hFind = FindFirstFile(type, &FindFileData);

          if (hFind != INVALID_HANDLE_VALUE)
          {
                    SYSTEMTIME st;
                    tu ft;

                    GetSystemTime(&st);
                    SystemTimeToFileTime(&st, &ft.fileTime);

                    while (FindNextFile(hFind, &FindFileData) != 0)
                    {
                              if (FILE_ATTRIBUTE_DIRECTORY != FindFileData.dwFileAttributes)
                              {
                                        tu t;
                                        t.fileTime = FindFileData.ftCreationTime;

                                        _int64 delta = (ft.ul.QuadPart - t.ul.QuadPart) / 10000000;      //
Seconds.
                                        int ddays = (int)(delta /= (24 * 3600));
```

```
                //_tprintf (TEXT("Next file name is: %s delta days %d\n"),
FindFileData.cFileName, ddays);
                if (ddays >= days)
                {
                    //_tprintf (TEXT("Next file to delete is: %s delta days %d\n"),
FindFileData.cFileName, ddays);
                    DeleteFile(FindFileData.cFileName);
                }
                //else
                //{
                //    _tprintf (TEXT("Skipping: %s delta days %d\n"),
FindFileData.cFileName, ddays);
                //}
            }
        }
        FindClose(hFind);

    }
    SetCurrentDirectory(curdir);
}
#endif
```

License

About the Author

Tim Stevens

Software Developer (Senior)

United Kingdom

No Biography provided