Introduction

Visual C++ provides built-in memory leak detection, but its capabilities are minimal at best. This memory leak detector was created as a free alternative to the built-in memory leak detector provided with Visual C++. Here are some of Visual Leak Detector's features, none of which exist in the built-in detector:

- Provides a complete stack trace for each leaked block, including source file and line number information when available.
- Detects most, if not all, types of in-process memory leaks including COM-based leaks, and pure Win32/Win64 heap-based leaks.
- Selected modules (DLLs or even the main EXE) can be excluded from leak detection.
- Provides complete data dumps (in hex and ASCII) of leaked blocks.
- Customizable memory leak report: can be saved to a file or sent to the debugger and can include a variable level of detail.

Other after-market leak detectors for Visual C++ are already available. But most of the really popular ones, like Purify and BoundsChecker, are very expensive. A few free alternatives exist, but they're often too intrusive, restrictive, or unreliable. Visual Leak Detector is currently the only freely available memory leak detector for Visual C++ that provides all of the above professional-level features packaged neatly in an easy-to-use library.

Visual Leak Detector is licensed free of charge as a service to the Windows developer community. If you find it to be useful and would like to just say "*Thanks!*", or you think it stinks and would like to say "*This thing sucks!*", please feel free to write review for recent release. Or, if you'd prefer, you can contribute a small donation. Both are very appreciated.

Using Visual Leak Detector

This section briefly describes the basics of using Visual Leak Detector (VLD).

Important! : Before using VLD with any Visual C++ project, you must first add the Visual Leak Detector include and library directories to the Visual C++ include and library directory search paths: *For all compiler versions take care to ensure that no junk characters get added when you add the include and library paths. If you browse to the "Program Files(x86) folder using the dialog box provided by Visual Studio and select it you could end up seeing the "%" replacing the "(".*

And remember to close and open the Visual Studio IDE once you have modified the default include and library paths which the compiler and linker would always look at.

- **Visual C++ 2010/2012/2013**: Go to *View ->Property Manager*, select *Microsoft.Cpp.Win32.user*. Select*VC++ Directories* and then "*Include files*" from the tree. Add the *include* subdirectory from the Visual Leak Detector installation directory. Move it to the bottom of the list. Then select "*Library files*" from the drop-down menu and add the *lib\Win32* subdirectory from the Visual Leak Detector installation directory. Again, move it to the bottom of the list. Repeat for *Microsoft.Cpp.x64.user*, but select *lib\Win64* subdirectory instead.
- **Visual C++ 2005 and 2008**: Go to *Tools -> Options -> Projects* and *Solutions -> VC++ Directories*. Select"*Include files*" from the "*Show Directories For*" drop-down menu. Add the *include* subdirectory from the Visual Leak Detector installation directory. Move it to the bottom of the list. Then select "*Library files*" from the drop-down menu and add the *lib\Win32* subdirectory from the Visual Leak Detector installation directory. Again, move it to the bottom of the list.
- **Visual C++ 2003**: Go to *Project Properties -> C/C++ -> General -> Additional Include Directories* and add the include subdirectory from the Visual Leak Detector installation directory. Move it to the bottom of the list. Then select *Additional Library Directories* and add the *lib\Win32* subdirectory from the Visual Leak Detector installation directory. Again, move it to the bottom of the list.

To use VLD with your project, follow these simple steps:

1. In at least one C/C++ source file from your program, include the *vld.h* header file. It should not matter which file you add the include statement to. It also should not matter in what order the header is included in relation to other headers. The only exception is *stdafx.h* (or any other precompiled header). A precompiled header, such as stdafx.h, must always be the first header included in a source file, so *vld.h*must be included after any precompiled headers.
2. If your program contains one or more DLLs that you would also like to check for memory leaks, then also include *vld.h* in at least one source file from each DLL to be included in leak detection.
3. Build the debug version of your program.

Note: Unlike earlier (pre-1.9) versions of VLD, it is now acceptable to include *vld.h* in every source file, or to include it in a common header that is included by many or all source files. Only one copy of the VLD code will be loaded into the process, regardless of how many source files include *vld.h*.
VLD will detect memory leaks in your program whenever you run the debug version. When you run the program under the Visual C++ debugger, a report of all the memory leaks detected will be displayed in the debugger's output window when your program exits (the report can optionally be saved to a file instead, see *ReportFile*under *Configuration Options*). Double-clicking on a source file's line number in the memory leak report will take you to that file and line in the editor window, allowing easy navigation of the code path leading up to the allocation that resulted in the memory leak.

Note: When you build release versions of your program, VLD will not be linked into the executable. So it is safe to leave *vld.h* included in your source files when doing release builds. Doing so will not result in any performance degradation or any other undesirable overhead.

Configuration Options

There are a several configuration options that control specific aspects of VLD's operation. These configuration options are stored in the vld.ini configuration file. By default, the configuration file should be in the Visual Leak Detector installation directory. However, the configuration file can be copied to the program's working directory, in which case the configuration settings in that copy of vld.ini will apply only when debugging that one program.

VLD

This option acts as a master on/off switch. By default, this option is set to "on". To completely disable Visual Leak Detector at runtime, set this option to "off". When VLD is turned off using this option, it will do nothing but print a message to the debugger indicating that it has been turned off.

AggregateDuplicates

Normally, VLD displays each individual leaked block in detail. Setting this option to "yes" will make VLD aggregate all leaks that share the same size and call stack under a single entry in the memory leak report. Only the first leaked block will be reported in detail. No other identical leaks will be displayed. Instead, a tally showing the total number of leaks matching that size and call stack will be shown. This can be useful if there are only a few sources of leaks, but those few sources are repeatedly leaking a very large number of memory blocks.

ForceIncludeModules

In some rare cases, it may be necessary to include a module in leak detection, but it may not be possible to include vld.h in any of the module's sources. In such cases, this option can be used to force VLD to include those modules in leak detection. List the names of the modules (DLLs) to be forcefully included in leak detection. If you do use this option, it's advisable to also add vld.lib to the list of library modules in the linker options of your project's settings.

Caution: Use this option only when absolutely necessary. In some situations, use of this option may result in unpredictable behavior including false leak reports and/or crashes. It's best to stay away from this option unless you are sure you understand what you are doing.

MaxDataDump

Set this option to an integer value to limit the amount of data displayed in memory block data dumps. When this number of bytes of data have been dumped, the dump will stop. This can be useful if any of the leaked blocks are very large and the debugger's output window becomes too cluttered. You can set this option to 0 (zero) if you want to suppress data dumps altogether.

MaxTraceFrames

By default, VLD will trace the call stack for each allocated block as far back as possible. Each frame traced adds additional overhead (in both CPU time and memory usage) to your debug executable. If you'd like to limit this overhead, you can define this macro to an integer value. The stack trace will stop when it has traced this number of frames. The frame count may include some of the "internal" frames which, by default, are not displayed in the debugger's output window

(see **TraceInternalFrames** below). In some cases there may be about three or four "internal" frames at the beginning of the call stack. Keep this in mind when using this macro, or you may not see the number of frames you expect.

ReportEncoding

When the memory leak report is saved to a file, the report may optionally be Unicode encoded instead of using the default ASCII encoding. This might be useful if the data contained in leaked blocks is likely to consist of Unicode text. Set this option to "unicode" to generate a Unicode encoded report.

ReportFile

Use this option to specify the name and location of the file in which to save the memory leak report when using a file as the report destination, as specified by the **ReportTo** option. If no file is specified here, then VLD will save the report in a file named "memory_leak_report.txt" in the working directory of the program.

ReportTo

The memory leak report may be sent to a file in addition to, or instead of, the debugger. Use this option to specify which type of destination to use. Specify one of "debugger" (the default), "file", or "both".

SelfTest

VLD has the ability to check itself for memory leaks. This feature is always active. Every time you run VLD, in addition to checking your own program for memory leaks, it is also checking itself for leaks. Setting this option to "on" forces VLD to intentionally leak a small amount of memory: a 21-character block filled with the text "Memory Leak Self-Test". This provides a way to test VLD's ability to check itself for memory leaks and verify that this capability is working correctly. This option is usually only useful for debugging VLD itself.

SlowDebuggerDump

If enabled, this option causes Visual Leak Detector to write the memory leak report to the debugger's output window at a slower than normal rate. This option is specifically designed to work around a known issue with some older versions of Visual Studio where some data sent to the output window might be lost if it is sent too quickly. If you notice that some information seems to be missing from the memory leak report, try turning this on.

StackWalkMethod

Selects the method to be used for walking the stack to obtain call stacks for allocated memory blocks. The default "fast" method may not always be able to successfully trace completely through all call stacks. In such cases, the "safe" method may prove to be more reliable in obtaining the full stack trace. The disadvantage with the "safe" method is that it is significantly slower than the "fast" method and will probably result in very noticeable performance degradation of the program being debugged. In most cases it should be okay to leave this option set to "fast". If you experience problems getting VLD to show call stacks, you can try setting this option to "safe".

If you do use the "safe" method, and notice a significant performance decrease, you may want to consider using the **MaxTraceFrames** option to limit the number of frames traced to a relatively small number. This can reduce the amount of time spent tracing the stack by a very large amount.

StartDisabled

Set this option to "yes" to disable memory leak detection initially. This can be useful if you need to be able to selectively enable memory leak detection from runtime, without needing to rebuild the executable; however, this option should be used with caution. Any memory leaks that may occur before memory leak detection is enabled at runtime will go undetected. For example, if the constructor of some global variable allocates memory before execution reaches a subsequent call to VLDEnable, then VLD will not be able to detect if the memory allocated by the global variable is never freed. Refer to the following section on controlling leak detection at runtime for details on using the runtime APIs which can be useful in conjunction with this option.

TraceInternalFrames

This option determines whether or not all frames of the call stack, including frames internal to the heap, are traced. There will always be a number of frames on the call stack which are internal to Visual Leak Detector and C/C++ or Win32 heap APIs that aren't generally useful for determining the cause of a leak. Normally these frames are skipped during the stack trace, which somewhat reduces the time spent tracing and amount of data collected and stored in memory. Including all frames in the stack trace, all the way down into VLD's own code can, however, be useful for debugging VLD itself.

SkipHeapFreeLeaks

Determines whether or not report memory leaks when missing HeapFree calls.

Controlling Leak Detection at Runtime

Using the default configuration, VLD's memory leak detection will be enabled during the entire run of your program. In certain scenarios it may be desirable to selectively disable memory leak detection in certain segments of your code. VLD provides simple APIs for controlling the state of memory leak detection at runtime. To access these APIs, include vld.h in the source file that needs to use them.

VLDDisable
This function disables memory leak detection. After calling this function, memory leak detection will remain disabled until it is explicitly re-enabled via a call to VLDEnable.

```
void VLDDisable (void);
```

Arguments:
This function accepts no arguments.

Return Value:

None (this function always succeeds).

Notes:

This function controls memory leak detection on a per-thread basis. In other words, calling this function disables memory leak detection for only the thread that called the function. Memory leak detection will remain enabled for any other threads in the same process. This insulates the programmer from having to synchronize multiple threads that disable and enable memory leak detection. However, note also that this means that in order to disable memory leak detection process-wide, this function must be called from every thread in the process.

VLDEnable
This function enables memory leak detection if it was previously disabled. After calling this function, memory leak detection will remain enabled unless it is explicitly disabled again via a call to VLDDisable().

```
void VLDEnable (void);
```

Arguments:
This function accepts no arguments.

Return Value:

None (this function always succeeds).

Notes:

This function controls memory leak detection on a per-thread basis. See the remarks for *VLDDisable*regarding multithreading and memory leak detection for details. Those same concepts also apply to this function.

Building Visual Leak Detector from Source

Because Visual Leak Detector is open source, it can be built from source if you want to tweak it to your liking. As of Visual Studio 2008, the source can usually be built out-of-the-box without downloading or installing any other tools. If you are using Visual Studio 2008 (or later), you can skip ahead to Executing Your Built vld.dll.

Users with older versions of Visual Studio should continue reading here and follow the instructions in the next subsection.

For Older Versions of Visual Studio
The most difficult part about building VLD from source is getting your build environment correctly set up. But if you follow these instructions carefully, the process should be fairly painless.

1. VLD depends on the Debug Help Library. This library is part of Debugging Tools for Windows (DTfW). Download and install DTfW in order to install the required headers and libraries. I recommend installing version 6.5 of DTfW, or later. Newer versions tend to work fine, but older versions will probably not work. Be sure to manually select to install the SDK files during the DTfW installation or the headers and libraries will not be installed (they are not always installed with a default installation).
2. Visual C++ will need to be made aware of where it can find the Debug Help Library header and library files. Add the *sdk\inc* and *sdk\lib\i368 (sdk\lib\amd64)* subdirectories from the DTfW installation directory to the include and library search paths in Visual C++. (See the section above on using Visual Leak Detector on instructions for adding to these search paths).
3. VLD also requires a reasonably up-to-date *Microsoft Windows SDK*. It is known to work with the latest SDK (as of this writing) which is the *Microsoft Windows SDK for Windows 7 and .NET Framework 4*. It should also work with earlier SDKs, such as the *Windows XP SP2 SDK*. If in doubt, update your *Microsoft Windows SDK*to the latest version.
4. Again, Visual C++ will need to know where to find the *Microsoft Windows SDK* headers and libraries. Add the Include and Lib subdirectories from the Platform SDK installation directory to the Include and Library search paths, respectively. The *Microsoft Windows SDK* directories should be placed just after the DTfW directories.

To summarize, your Visual C++ include search path should look something like this:

```
C:\Program Files\Debugging Tools for Windows (x86)\sdk\inc
C:\Program Files\Microsoft SDKs\Windows\v7.0A\Include
C:\Program Files\Microsoft Visual Studio x\VC\Include
...
```

And your Visual C++ library search path should look like this:

```
C:\Program Files\Debugging Tools for Windows (x86)\sdk\lib\i386
```

```
C:\Program Files\Microsoft SDKs\Windows\v7.0A\Lib
C:\Program Files\Microsoft Visual Studio x\VC\Lib
...
```

In the above examples, "x" could be "8", "9.0", or "10.0" (or possibly other values) depending on which version of Visual Studio you have installed. Also, the name of your *Microsoft Windows SDK* directory will probably be different from the example depending on which version of the Platform SDK you have installed.

Once you have completed all of the above steps, your build environment should be ready. To build VLD, just open the *vld.sln* solution file and do a full build.

Executing Your Built vld.dll

When actually running the built project, vld.dll will expect to find the Debug Help Library as a private assembly. The private assembly must be located in the same directory as vld.dll (either the Release or Debug directory by default). Otherwise, when VLD is loaded, an error message will pop up indicating that the program failed to initialize, and you will see a message similar to the following in the debugger's output window:

LDR: LdrpWalkImportDescriptor() failed to probe C:\Projects\vld\Release\vld.dll for its manifest, ntstatus 0xc0150002

To ensure that vld.dll finds the required private assembly, you need to copy *dbghelp.dll* and*Microsoft.DTfW.DHL.manifest* to the same directory that vld.dll is in.

Frequently Asked Questions

When I try to compile my program with VLD, it fails and the compiler gives this error: **Cannot open include file: 'vld.h': No such file or directory.**

The compiler can't find the header file that VLD installed. This probably means that VLD's include subdirectory has not been added to the Visual C++ include search path. See the section above about Using Visual Leak Detector for instructions on how to add VLD's directories to the search path.
In the memory leak report, the callstack contains many lines that say **"File and line number unvailable"** or**"Function name unavailable"**.

This may mean that VLD couldn't find the symbol database for your program. The symbol database is ususally in a file named my-program-name.pdb. If this file is not located in the same directory as the program itself, then VLD will probably not find it and can't show any file or function names. Set environment variable **DBGHELP_DBGOUT** to check this.

Known Restrictions

Known restrictions/limitations in this version of VLD include:

- Memory allocations made through calls to functions loaded from a DLL using delayed loading may not be detected.
- Support for programs that use MFC 7.0 or MFC 7.1 is not complete yet. Some memory leaks from such MFC-based programs may not be detected. A possible workaround for this restriction is to try forcefully including the MFC DLLs in memory leak detection, by setting the ForceIncludeModules configuration option to: "mfc70d.dll mfc71d.dll" and explicitly adding vld.lib as an input file on the linker command line (can be added through project settings by adding it to the list of library modules in the linker options). This restriction does not apply to programs that use MFC 4.2, MFC 8.0, MFC 9.0, or MFC 10.0, which are all fully supported.
- Visual Leak Detector may report leaks internal to Visual Leak Detector if the main thread of the process terminates while other threads are still running.
- If more than one copy of the same C Runtime DLL is loaded in the process at the same time, then some leaks may go undetected (note that loading more than one copy of the C Runtime DLL at the same time is probably a bad idea to begin with).

Contributing

I encourage developers who've added their own features, or fixed bugs they've found, to contribute to the project. The full version-controlled source tree is available at this site.

GitHub Mirror: https://github.com/KindDragon/vld

License

Visual Leak Detector is distributed under the terms of the GNU Lesser General Public License (LGPL). This license allows you to use the VLD library with your own programs without restriction. However, if you build a program (or another library) that is based on the VLD source code, or uses parts of the VLD source code in it, then some restrictions will apply. What this means is that you are free to ship and use the distributed version of the VLD DLL with regular commercial programs. But if you create a modified version of VLD, that modified version must remain "free software". See the COPYING.txt file for details.

The Debug Help Library (dbghelp.dll) distributed with this software are not part of Visual Leak Detector and are not covered under the terms of the GNU Lesser General Public License. They are separately copyrighted works of Microsoft Corporation. Microsoft reserves all its rights to its copyrights in the Debug Help Library and Microsoft C Runtime Library. Neither your use of the Visual Leak Detector software, nor your license under the GNU Lesser General Public license grant you any rights to use the Debug Help Library or Microsoft C Runtime Library in **ANY WAY** (for example, redistributing them) that would infringe upon Microsoft Corporation's copyright in the Debug Help Library or Microsoft C Runtime Library.

NO WARRANTY
BECAUSE VISUAL LEAK DETECTOR ("THE SOFTWARE") IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE SOFTWARE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE SOFTWARE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE SOFTWARE AS PERMITTED BY THE LICENSING TERMS SET FORTH ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Additional Developers Wanted

This project is looking for additional developers who have the time, knowledge, and talent, to help make VLD continue to be a useful utility for the Windows developer community. If you feel that you, than you fix bugs or implement new features and create pull request to project.