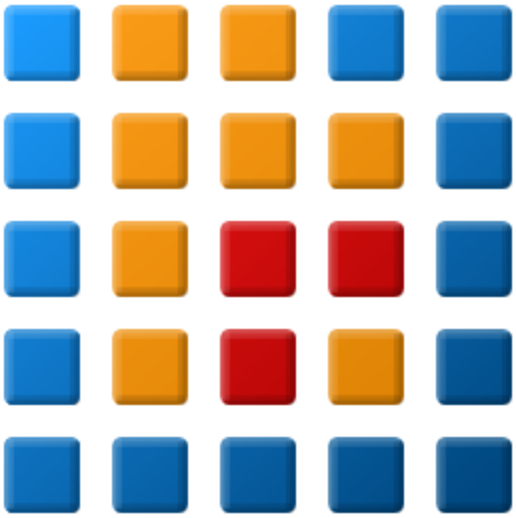


Memory Validator

by

Software Verify

Software Verify

Troubleshoot your software

Memory Validator

Memory leak detection for Windows applications built using .Net, .Net Core, C#, VB.Net, C, C++, Delphi, Fortran 95 and Visual Basic 6.

---

by Software Verify Limited

Welcome to the Memory Validator software tool. Memory Validator is a very powerful memory and handle analysis software tool that can be used for determining the cause of leaking objects, memory hotspots, memory corruption and performing regression tests amongst many other uses.

Memory Validator provides the ability to enable and disable all functionality at a fine-grained level, thus providing you with power and control over the data that Memory Validator monitors.

We have provided two levels of user interface, one simplistic and the other comprehensive and powerful to cater for both novices and experienced software engineers.

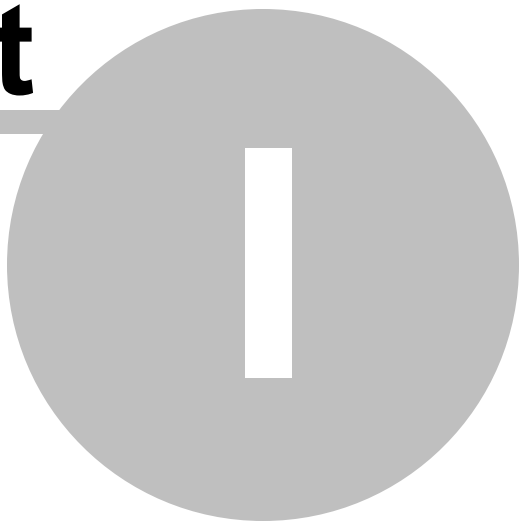
We hope you will find this document useful.

Memory Validator Help

Printed: June 2025 in United Kingdom.

Table of Contents

Part



1 Overview

Hi, welcome to the Memory Validator help manual.

This help manual is available in Compiled HTML Help (Windows Help files), PDF, and online.

Windows Help	https://www.softwareverify.com/documentation/chm/memoryValidator.chm
PDF	https://www.softwareverify.com/documentation/pdfs/memoryValidator.pdf
Online	https://www.softwareverify.com/documentation/html/memoryValidator/index.html

Tutorials for Memory Validator are available at https://www.softwareverify.com/tutorial/memory-validator-tutorial/.

Before reading this manual, it's worth taking a quick look at the notation used.

Read background information

The overview section covers things like:

- the capabilities of Memory Validator
- how it works
- what's supported
- how to purchase

  If you've already purchased, thank you!

Learn about getting started

You can skip the background information, but do make sure you're aware of how to prepare your target program in the getting started section.

Dive right in

The quick start section shows how to launch your application

To find your way around the rest of the features and settings then read about the user interface, or browse the examples.

If you're already feeling confident you can read about some of the advanced features such as regression testing, the API and extending Memory Validator.

Don't miss the powerful and easy-to-use tag tracking feature.

1.1 Notation used in this help

› Instruction › steps
Menu action › steps

Throughout the help you'll find instruction steps like this:

- **Filter...** › shows the session comparison private filters dialog

or

Settings Menu › **Edit Settings...** › **Data Collection** in the list › **Trace Hooks**

This is a shorthand notation for performing consecutive steps in the user interface.

The first example indicates that the action of clicking the **Filter...** button will result in showing the dialog described.

The second example directs you to open the Settings menu (from the menu bar in this case), and then choose the Edit Settings item, and in the dialog that appears, open the Data Collection option via the list and select the Trace Hooks child entry.

Right mouse button menu

Where you see this mouse menu the instruction is to use the right mouse button menu (a.k.a. popup menu or context menu) and select the menu option that follows this symbol.

For example: use **Edit Source Code...**

Interactive images

Shown next to a screenshot or illustration, the hand symbol indicates the picture is interactive and can be clicked on in order to jump directly to the help section most relevant to the part of the image under the cursor.

External Links

You may see this symbol after some links. Those links lead to an external website (shown in your default browser), as opposed to jumping to another section in the help. Naturally, if you have no internet access, these links will be unavailable.

For example: Software Verify Limited

Notes

Warning notes

Notes pertaining to the current topic are indicated by the symbol. Notes may include exceptions to a rule, items to watch out for, or other asides to the main topic.

Notes that include warnings will use the similar symbol, for example where there's a danger of crashing your application. Don't panic though - there aren't many of these!

See also

Where there are other pages in the help that have more detail on the topic at hand, or if there is additional reading that is not already linked within the content, you will find these sections linked after the ➜ symbol.

1.2 Introducing Memory Validator

What is Memory Validator?

Memory Validator is an advanced **memory and resource leak detector** for Windows.

It works with versions of Windows from NT® 4.0 and above, running on the Intel i386 (and compatible) family of processors.

What does it do?

If you think your software might be leaking or even corrupting memory or resources, then Memory Validator will help with that - and more.

Memory Validator is fast and does not require the target program to be recompiled or relinked.

It detects:

- memory leaks
- handle leaks
- memory corruptions
- uninitialized memory
- buffer overruns and underruns

It finds performance problems related to:

- memory allocation and deallocation
- resource allocation and deallocation
- fragmented memory heaps

And it does this by tracking:

- C runtime heaps
- Win32 heaps
- GlobalAlloc heap
- LocalAlloc heap

- VirtualAlloc heaps
- Win32 resource handles
- COM object creation and destruction
- .Net object and .Net handle creation and garbage collection

Not all the above capabilities are enabled 'out of the box' due to extra performance impacts involved. Also, note that not all capabilities can be enabled simultaneously.

➡ What Memory Validator does.

You can learn more about memory management on Windows at the MSDN Dev Center.

The main sections of Memory Validator

The user interface is split via tabs into seven separate sections (+Tutorials), each dedicated to a different task for analysing a problem that is present in the target program.

Here's a summary of those sections, each of which is covered in full in The User Interface section.

Summary	A high level dashboard for everything that you might want to look at. Click on any statistic to see more detailed information.
Memory	Displays any leaked memory or handles, and memory errors, such as double frees, incorrect frees, uninitialized data etc. Contains subtabs for Native and .Net.
Timeline	Shows a visual timeline of total memory, handle allocations and memory allocations.
Statistics	Statistics for types, sizes, allocation location, generations and object ages.
.Net	.Net specific user interfaces for memory snapshots, heap dumps and .net leak analysis.
Analysis	Allocation hotspots, allocation coverage, allocation queries, memory page layout and virtual memory analysis.
Diagnostic	Lists diagnostic information collected by the stub.
Tutorial	Tutorial exercises for people new to the software.

1.3 Why Memory Validator?

Adapts to everyone's workflow

Memory Validator allows you to find otherwise hard to isolate errors, using an intuitive user interface.

Many query options are provided so that when faced with a large amount of data you can search or filter to reduce the amount of data to examine.

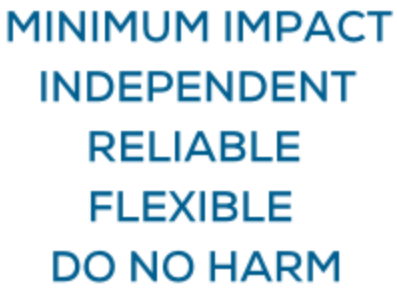
Want to edit the offending source code? Simple - just double click on the code fragment shown and the appropriate source code file will be loaded into Memory Validator's colour coded editor, or into Microsoft® Visual Studio®, or you can choose your preferred awesome editor.

Memory Validator can monitor regression tests, producing HTML and/or XML reports detailing regressions and improvements (compared to a baseline).

You can even reload the saved session at a later date and interact with the collected data. Being HTML or XML output, the results can be used to create reports targeted to the appropriate audience: the management team; quality assurance team; or detailed stack traces for the software engineers.

Designed with principles

Memory Validator has been created with the following principles in mind:



- **must not adversely effect on the program's behaviour**

  Any hooks placed into the target program's code must not affect the registers or the condition code flags of that program. The program must behave in the same way when being inspected by Memory Validator as without.

- **must be reliable and avoid causing the target program to crash**

  Since we can't know exactly which DLLs and other components are present on every computer that Memory Validator is installed on, every hook and group of hooks is configurable so that they can be enabled or disabled, and/or installed or not installed.

  Thus if a new DLL is released in the future that causes problems with certain functions, you can disable the hooks for those functions, and continue using Memory Validator until a fix for the new DLL's behaviour is available.

- **must have as little impact on the target program's performance as possible**

  You can enable and disable as many or as few function hooks as you wish. For example:

  If you are only interested in the CRT for a particular bug, turn all the COM, uninitialized data, buffer and handle checking off.

  If you are only interested in buffer overflows turn everything off except the buffer checking hooks.

If you have handle leaks that you know come from the GDI, turn off all other handle checking code and only check the GDI handles.

If you want to go deeper than that and only check handles from `GetDC()` and `ReleaseDC()` you can do that too!

- **must have a user interface independent of the target program**

  Memory Validator's user interface is independent of the target program. So for example:

  If the target program crashes, the user interface will not crash - you will still have data to work with.

  If the target program is stopped in the debugger, Memory Validator's user interface will continue to work.

  In the unlikely event that the Memory Validator user interface crashes, your target program will not crash.

- **must be flexible**

  We know our users like choices! Where there are multiple ways of presenting the data, the user is given a choice over how that display works, so that not all users have to work with the same settings.

1.4 What do you need to run Memory Validator?

Compilers

The following makes of compiler are supported:

- Microsoft® Visual Studio®
- Borland C++
- Borland Delphi
- Intel
- Metrowerks
- MinGW
- QtCreator
- Fortran (various)

➔ Supported compilers for more details regarding versions and caveats.

User Privileges

Memory Validator can be run as a normal user or an administrator user.

For any tasks that require special privileges to achieve Memory Validator will do these tasks via the svlAdminService that was installed and started when you installed Memory Validator.

You can also run Memory Validator with elevated privileges.

Operating System

Any modern windows machine is suitable to run Memory Validator.

At a minimum, Memory Validator requires Windows NT® 4.0 or better.

Because the `CreateRemoteThread()` Win32 function and named pipes are not available on Windows 95®, Windows 98® and Windows Me®, Memory Validator will not run on these platforms.

Any newer operating systems do not *need* any additional service packs but we generally recommend being up to date where possible.

For older systems, we recommend using the minimum service pack levels below:

- Windows NT 4.0: Service Pack 6
- Windows 2000: Service Pack 2

1.5 Buying Memory Validator and support

The best way to purchase Memory Validator is online from Software Verify Limited - just click the ***Purchasing*** link at the top of the website.

Purchase options

There are options for single or multiple licenses, per-user or floating licenses, and although you can of course purchase it as a single product, you can save significantly by buying Memory Validator as part of a suite of products. All the details are online.

Pre-purchase questions?

If you have any pre-purchase questions not answered in this help manual, or niggling little doubts about something, we can be contacted as below.

email: sales@softwareverify.com *(recommended)*

web: https://www.softwareverify.com

or by old fashioned post:

Software Verify Limited
Suffolk Business Park
Eldo House
Kempson Way
Bury Saint Edmunds
IP32 7AR
United Kingdom

After sales support

If you need support after purchase, check our frequently asked questions and then drop us a line below with as much detail as possible about your problem.

email: support@softwareverify.com

1.6 How does Memory Validator work?

The Stub and the UI - more than the sum of its parts

Memory Validator has two main parts - the *stub* and the *user interface*.

The **stub** is typically injected into the target program and communicates with the Memory Validator user interface.

The stub is injected into the target program using the `CreateProcess()` or `CreateRemoteThread()` Win32 function. Communication between the stub and the user interface is via named pipes. There is no human readable data sent between the two parts of the program. Both the stub and the user interface are multithreaded.

The stub rewrites the *DLL import address table* and the *delay loaded import address tables* to make functions call into the stub's hooks. For functions that are not present in the import address tables, the stub rewrites the function prologue and function epilogue using proprietary techniques.

For COM objects. the stub rewrites the function prologue and function epilogue for the `AddRef()`, `Release()` and `QueryInterface()` functions in each COM object.

The stub can also be linked if required, for example in order to use the API in order to track custom memory heaps.

Memory Validator can be extended by user supplied extension DLLs. These can be used to augment either the stub or the user interface.

1.7 What does Memory Validator do?

Memory Validator provides functionality to detect memory leaks and resource leaks in 32 and 64 bit programs running on Windows NT® 4.0 and above.

Memory leaks

Memory leaks (allocations without a corresponding deallocation) are detected in the following:

Functions

- `new`
- `malloc`
- `calloc`
- `realloc`
- `HeapAlloc`
- `HeapReAlloc`
- `VirtualAlloc`
- `VirtualAllocEx`

- `_expand`
- `LocalAlloc`
- `LocalRealloc`
- `GlobalAlloc`
- `GlobalRealloc`
- `IMalloc`
- `CoTaskMemAlloc`
- `CoTaskMemReAlloc`
- `SysAllocString`

- NetApi function group
- Miscellaneous other memory allocation functions
- Crypt API

COM objects

- COM leaks

User defined

- User defined heaps (requires linking with svlMemoryValidatorStubLib.lib / svlMemoryValidatorStubLib_x64.lib to access this functionality)
- User defined reference counted objects (requires linking as above)
- Reference counting for user managed heaps.
- Memory leaks from user managed heaps.

Potentials

- Potentially unused memory.
- Potentially leaked memory.

Learn more about memory management on Windows at the MSDN Dev Center.

Handle leaks

Handle leaks are detected - typically handles that are not closed.

GDI Handle leaks

GDI Handle leaks are detected - typically handles that are not destroyed, or handles that are still selected into a DC.

Memory usage errors

Memory usage errors are typically the result of passing the wrong pointer to a deallocation or reallocation function, or attempting to deallocate memory more than once.

Examples detected include:

- Incorrect memory freeing - e.g. freeing a pointer not allocated from the appropriate heap
- Allocate with malloc() - then deallocate with delete or delete []
- Allocate with calloc() - then deallocate with delete or delete []
- Reallocate with realloc() or _expand() - then deallocate with delete or delete []
- Allocate with new - then deallocate with free()

- Allocate with new - then reallocate with realloc() or _expand()
- Multiple memory freeing - e.g. freeing the same pointer more than once

Memory corruption

- Heap overruns - writing past the *end* of an allocated heap block
- Heap underruns - writing past the *start* of an allocated block
- Heap corruption - heap getting damaged by heap overruns, heap underruns and wild pointer writes
- Stack overruns - writing past the *end* of the current stack frame
- Stack underruns - writing past the *start* of the current stack frame

Uninitialized data

- Uninitialized data in C++ objects.
- Detection of C++ objects that do not initialise all their member variables.

COM object tracking

Memory Validator can hook `AddRef()`, `Release()` and `QueryInterface()` so that it can track the reference counts for COM objects.

.Net memory and .Net handles

Memory Validator uses the .Net ICorProfiler and ICorProfilerCallback interfaces to monitor .Net applications.

1.8 Supported Compilers

Memory Validator will work with any portable executable (PE) file format and supports .Net, .Net Core, C#, VB.Net, C, C++, Delphi, Fortran 95 and Visual Basic.

All Win32 functions listed in the Hook Reference section will be tracked for any executable.

Microsoft .Net, .Net Core

Both .Net and .Net Core technologies are supported as well all the native compilers listed below.

The following compilers are supported by Memory Validator.

Microsoft http://www.microsoft.com

- **Microsoft Developer Studio 4.0**
- **Microsoft Developer Studio 5.0**
- **Microsoft Developer Studio 6.0**
- **Microsoft Visual Basic 6.0**

- **Microsoft Visual Studio 6.0 -** service pack 3 or later
- **Microsoft Visual Studio 7.0 / .net 2002**
- **Microsoft Visual Studio 7.1 / .net 2003**
- **Microsoft Visual Studio 8.0 / .net 2005**
- **Microsoft Visual Studio 9.0 / .net 2008**
- **Microsoft Visual Studio 10.0 / .net 2010**
- **Microsoft Visual Studio 11.0 / .net 2012**
- **Microsoft Visual Studio 12.0 / .net 2013**
- **Microsoft Visual Studio 14.0 / .net 2015**
- **Microsoft Visual Studio 15.0 / .net 2017**
- **etc...**
- **Microsoft Visual Studio Express -** supported, but see also: building the examples for Visual Studio Express

 Some of the heap identification functions will not work with applications built using Microsoft® Visual Studio® 4.0, 7.0, 7.1, 8.0, 9.0, 10.0. This only affects the virtual memory view in terms of representing a heap as a CRT heap - a minor lack of functionality.

➡ Visual Studio and Visual Basic 6 in the *Getting Started* section.

Intel http://www.intel.com

- **Intel performance compiler -** The Intel compiler uses the Microsoft runtime already installed on your computer rather than supply its own
- **Intel Fortran**

Metrowerks http://www.metrowerks.com

- **Metrowerks CodeWarrior for Windows Version 8.0**
- **Metrowerks CodeWarrior for Windows Version 9.0**

You will need to ensure the debug information is stored as *CodeView* information and not a custom Metrowerks debug format. Metrowerks symbolic information is embedded in the .exe/.dll as CodeView information. Please consult the documentation for CodeWarrior to include debug information (including filenames and line numbers) in the CodeView information.

Embarcadero https://www.embarcadero.com/

This includes compilers formerly produced by Borland.

- **C++ Builder 5.0 to C++ Builder 11**
- **Delphi 6.0 to Delphi 11**
- **Rad Studio**

➡ C++ Builder and Delphi in the *Getting Started* section.

MinGW http://www.mingw.org

- **MinGW** (Minimalist GNU for Windows)

MinGW can create symbols in a variety of formats. If you configure MinGW to produce DWARF symbols, STABS symbols or COFF symbols Memory Validator can read them.

➔ MinGW compiler in the *Getting Started* section.

Qt (Digia, Nokia, Trolltech) http://qt.digia.com

- **QtCreator**

➔ Ensure that debug information is created in DWARF, STABS or COFF formats.

Salford Software http://www.salfordsoftware.co.uk

- **Salford Software FORTRAN95**

Compaq

- **Compaq Visual Fortran 6.6**

The Compaq Visual Fortran product may be compatible with Memory Validator. If you are using Compaq Visual Fortran and wish to use Memory Validator please contact us.

Cherrystone

- **Cherrystone Software Extremely Scalable Allocator (ESA) -** a replacement allocator providing high performance allocation algorithms for use in single threaded applications and multi-threaded applications.

There is no web page for this product.

Other...?

If the compiler you are using is not listed here, please contact support@softwareverify.com for advice. We add compilers as we receive requests for them. The Borland C++, Borland Delphi, Metrowerks CodeWarrior, Salford Software's FORTRAN95 compiler, Intel Fortran support and Cherrystone ESA support were all added at the request of customers.

1.9 User Permissions

This section details the privileges a user requires to successfully run Memory Validator.

Typically, *Administrator* and *Power User* user types will already have the appropriate privileges.

Why do user privileges matter?

Debugging tools such as Memory Validator are intrusive tools - they require specific privileges not normally granted to typical applications.

Memory Validator requires specific privileges to write to the default user profile in the registry.

This is so that when Memory Validator is working with services (or any application run on an account which is not the current user's account) it can read the registry and the correct configuration data.

If the account upon which a service or application is running is *not* the user's account, the fallback position is the DEFAULT account in `HKEY_USERS\.DEFAULT`.

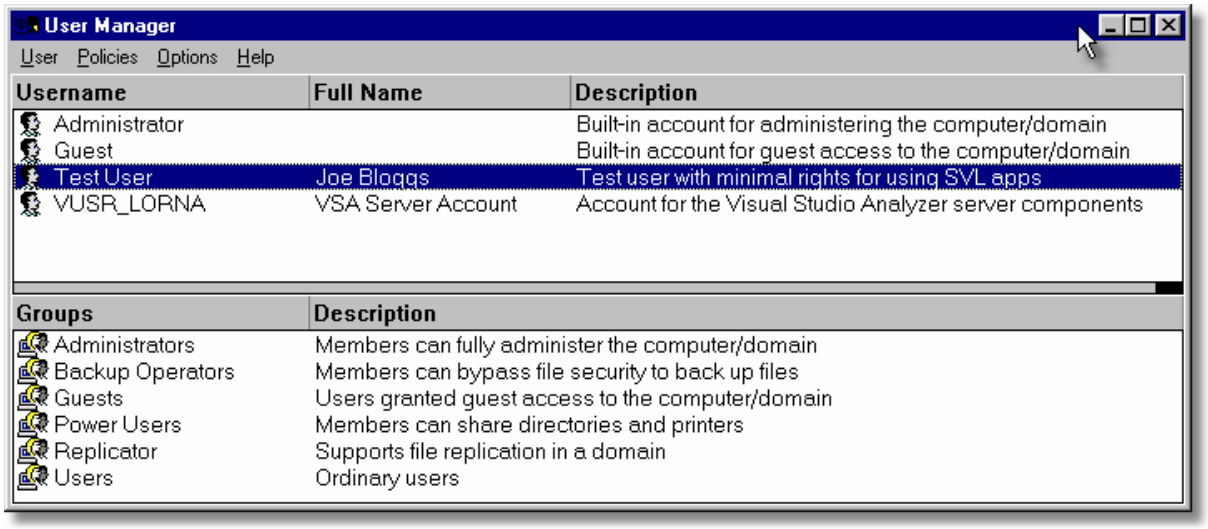
You can enable and disable various warnings using the User Permissions Warnings dialog.

User privileges

Memory Validator requires the following privilege to allow debugging of applications and services:

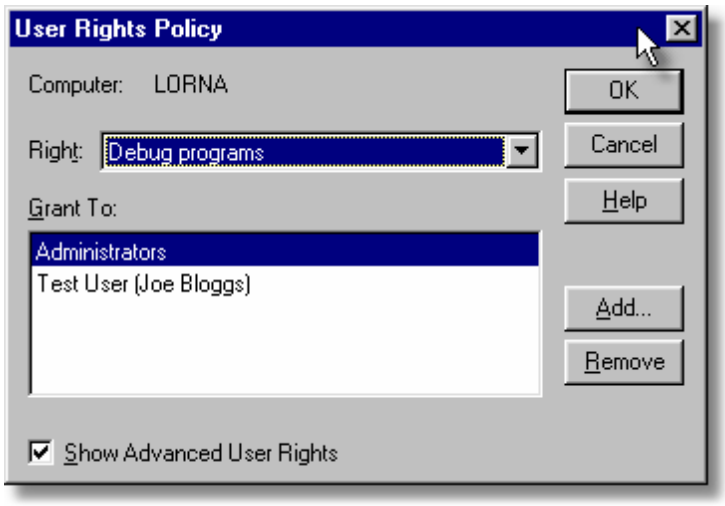
Debug Programs (SE_DEBUG_NAME)

Ordinary users will need to be granted these permissions using the Administrative *User Manager* tool. The example below shows the NT4 User Manager - the Windows 2000 User Manager and Windows XP User Manager will be different but similar in principle.

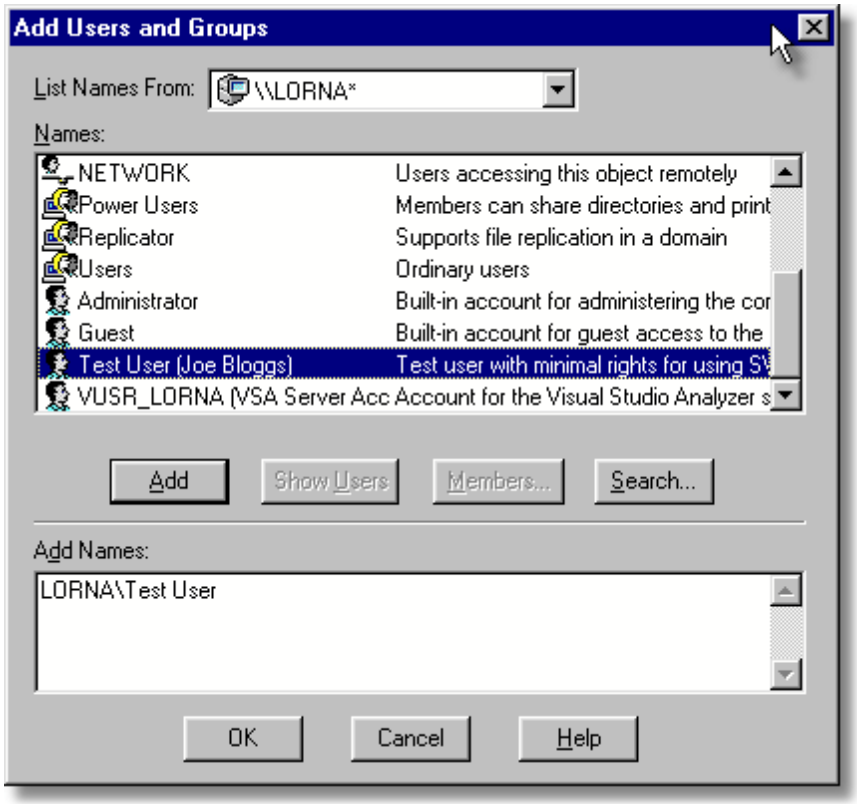


In the User Manager select the user - in this case "Test User".

Choose: **Policies** Menu › **User Rights** › check **Show Advanced User Rights** › select **Debug Programs** in the **Right** combo box



Click **Add...** › **Show Users**



Select **[ComputerName]\Test User** in the top list. Click **Add** › **OK** › **OK** › Close the User Manager.

Registry access privileges

Memory Validator requires read access and write access to:

```
HKEY_CURRENT_USER\Software\SoftwareVerification\MemoryValidator
```

When working with services, Memory Validator requires read access and write access to `HKEY_USERS\.DEFAULT\Software\SoftwareVerification\MemoryValidator`.

If read access and write access to that key is not allowed, Memory Validator will use default settings (thus any user selections will not apply).

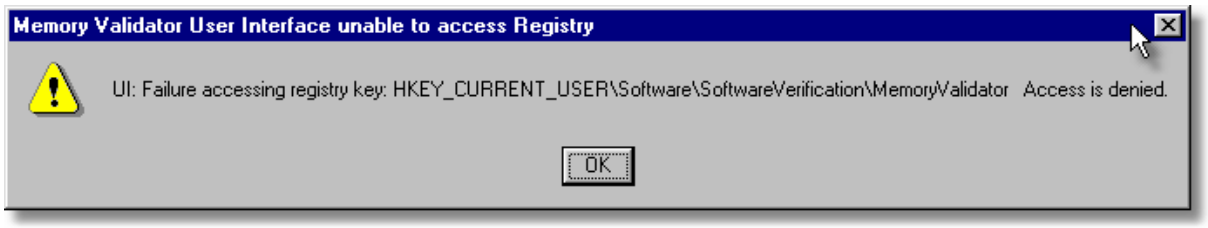
In addition, error messages will be displayed when Memory Validator tries to access this registry key. These error messages can be suppressed if they are not desired.

You can modify the registry access permissions using the **regedt32.exe** tool Security menu. Ask your administrator to modify your registry access permissions if you can't do this yourself.

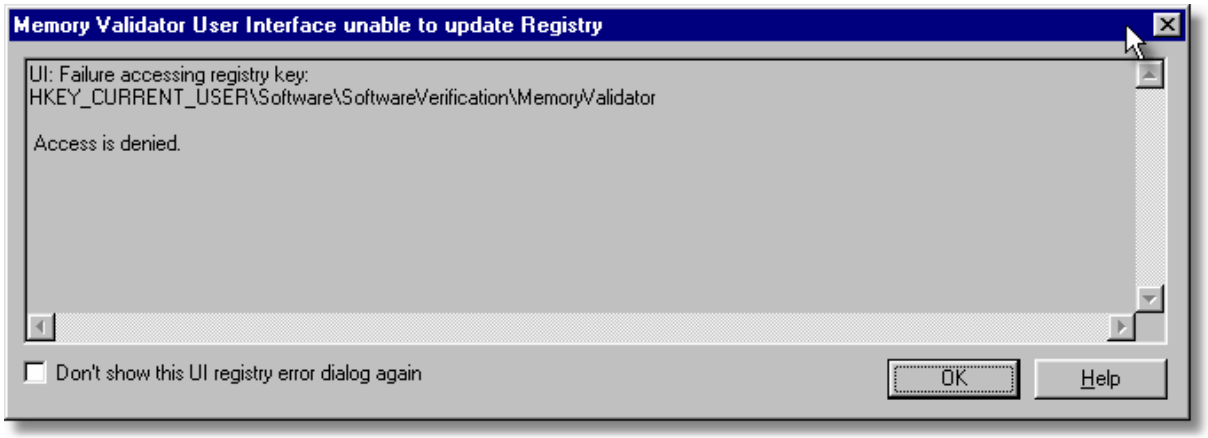
Error notifications

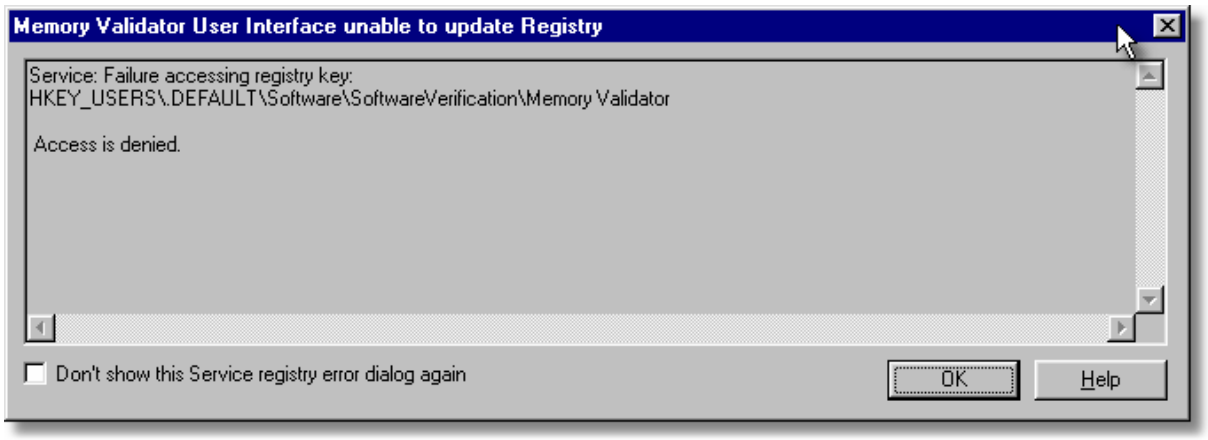
When Memory Validator fails to gain access for read or write to the registry a message box is displayed indicating if the error is for the user interface (UI) or Services. The message indicates the name of the registry key that failed and the failure reason.

This simple message box is displayed during early startup and late closedown of Memory Validator:



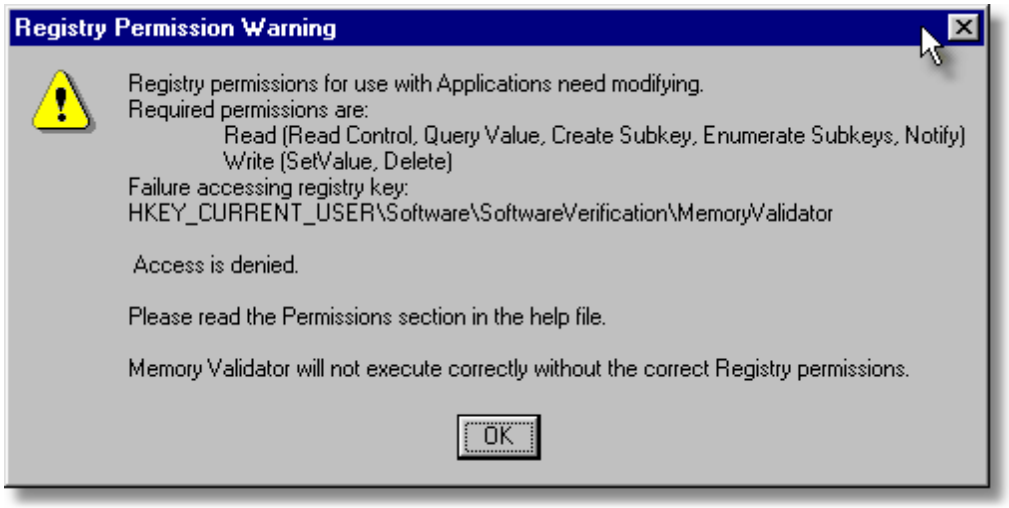
This message box is displayed when Memory Validator is not starting up or closing down:

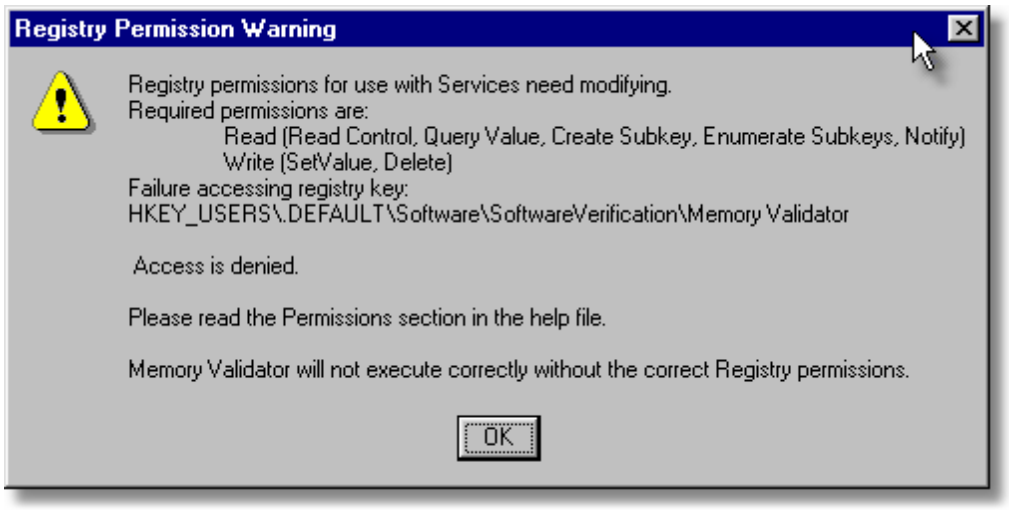




Detailed registry access error messages

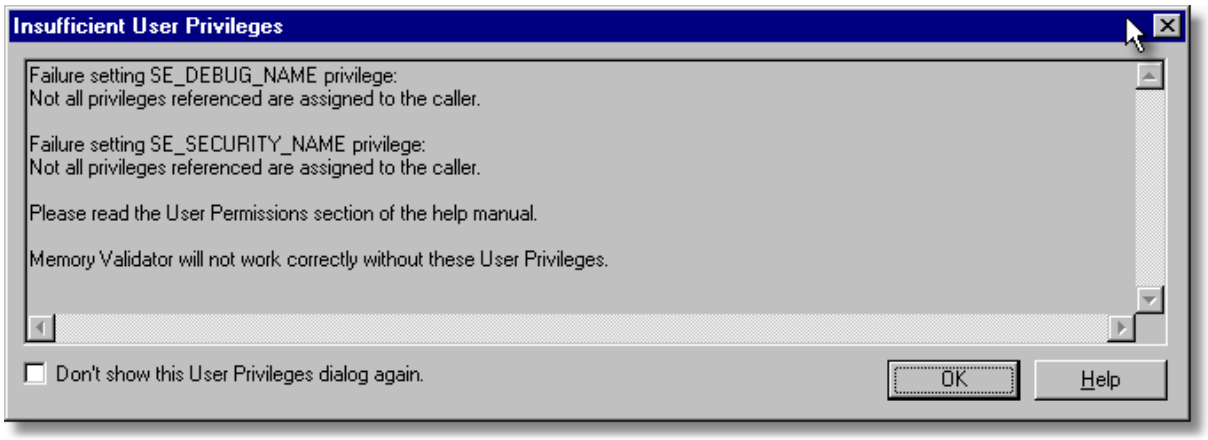
The following detailed registry access error messages are also displayed when failing to gain access to the registry:





Insufficient user privileges

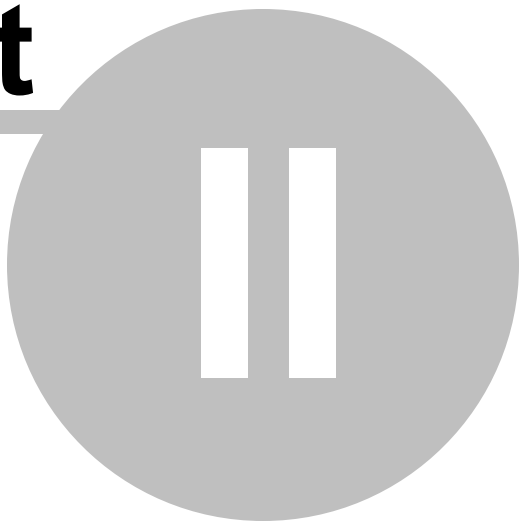
The following dialog is displayed if a user has insufficient privileges to use the software correctly.



Without the **Debug Programs** privilege Memory Validator will not work correctly with *Services*, and may not work correctly with *Applications*.

➡ Creating Power User accounts for Windows XP.

Part



2 Getting Started

For those that wish to 'dive in', this section will make you aware of how to prepare your target program before giving a quick start introduction.

Otherwise skip right on to the next chapter - The User Interface.

Diving in?

If you have never used Memory Validator before you have probably purchased Memory Validator because you're already aware of a memory leak or performance problem that you have in your application. As such you may want to 'dive in' and find memory leaks immediately.

However, if you choose to read this manual first you'll find out more about Memory Validator and how to leverage it to its full advantage.

For new users of Memory Validator, a configuration wizard appears the first time you run the application. This ends with a brief overview video.

We also recommend that you watch the tutorials online - it's an easy way to explore the functionality available.

2.1 Before you start

This section details the requirements necessary to monitor C runtime (CRT) heap allocations, followed by sections specific to enabling debugging information for Visual Studio, C++ Builder, Delphi, MinGW, and other compilers.

If you are using Memory Validator to monitor C runtime (CRT) heap allocations, read on, and check you have the necessary run time library setup.

Once you're confident you have the required environment (CRT and debug info settings), skip to Quick Start.

CRT heap allocations

The gist of the rest of this section is that Microsoft compiler users should use a dynamically linked CRT library, or you may use a statically linked CRT *provided* you have a matching MAP file.

➡ Supported compilers and linkers

Dynamic CRT linking

To track C runtime heap allocations, your program and/or dlls must be linked to the dynamic CRT in one of the following:

- MSVCRT.DLL
- MSVCRTD.DLL
- the equivalent depending on the version of Visual Studio e.g. msvcr70/71/80/90/100/110.dll
- an alternative library for other compilers, as outlined in the Memory Hooks reference

Static CRT linking

If you are linked statically, to one of the following CRTs:

- LIBC.LIB - single-threaded
- LIBCD.LIB - single-threaded debug
- LIBCMT.LIB - multi-threaded
- LIBCMTD.LIB - multi-threaded debug

then the CRT heap allocation tracking will function but performance will be slower than for the dynamically linked case.

However, in addition, you *must* generate MAP files for each EXE/DLL that you link statically to the C runtime. The MAP file should have the same name as the EXE/DLL but with the extension ".map", e.g. "example.dll" should have a corresponding MAP file "example.map".

- What are MAP files and why are they needed?MAP files are typically plain text files that indicate the relative offsets of functions for a given version of a compiled binary. They contain information about your program's global symbols, source file and source line numbers. In the event of statically linked CRT libraries, Memory Validator falls back to using the MAP file to locate the functions. You can create the MAP file when linking your executable or DLL by setting certain options for the linker.

- Why is dynamic linking better?The reason for this is that Memory Validator uses the Import Address Table to provide hooks into MSVCRT.DLL. When you link statically the Import Address Table is not present and cannot be hooked. Most large applications are dynamically linked (based on our testing), but many small projects are often built statically. These statically linked projects need to be built using the dynamic library for best monitoring performance.

 If using **Borland** compilers you *can* use Memory Validator to monitor memory allocations in a statically linked program. You'll be presented with the usual CRT warning dialog informing you that the use of the dynamic memory allocator could not be detected, but that can be ignored.

Which CRT am I using?

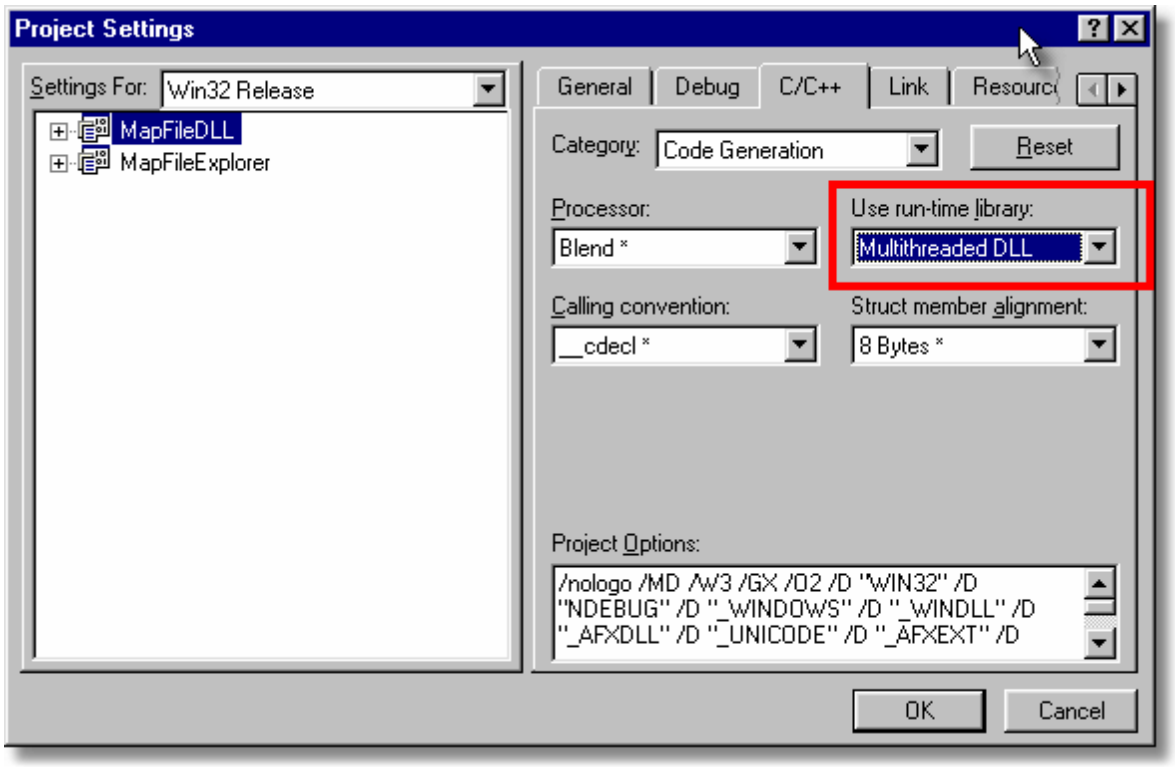
You can inspect (and change) the static/dynamic linking nature of your program:

Microsoft Developer Studio

Choose **Project** menu › **Settings...** › **C++** tab on the dialog box › Using the **Category** combo box choose **Code Generation**.

If you are using the dynamic multi-threaded MSVCRT.DLL the following will be shown in the **Use run-time library** combo box (highlighted in red below).

Alternatively, if using the *debug* MSVCRTD.DLL, the same box will show *Debug Multithreaded DLL*.

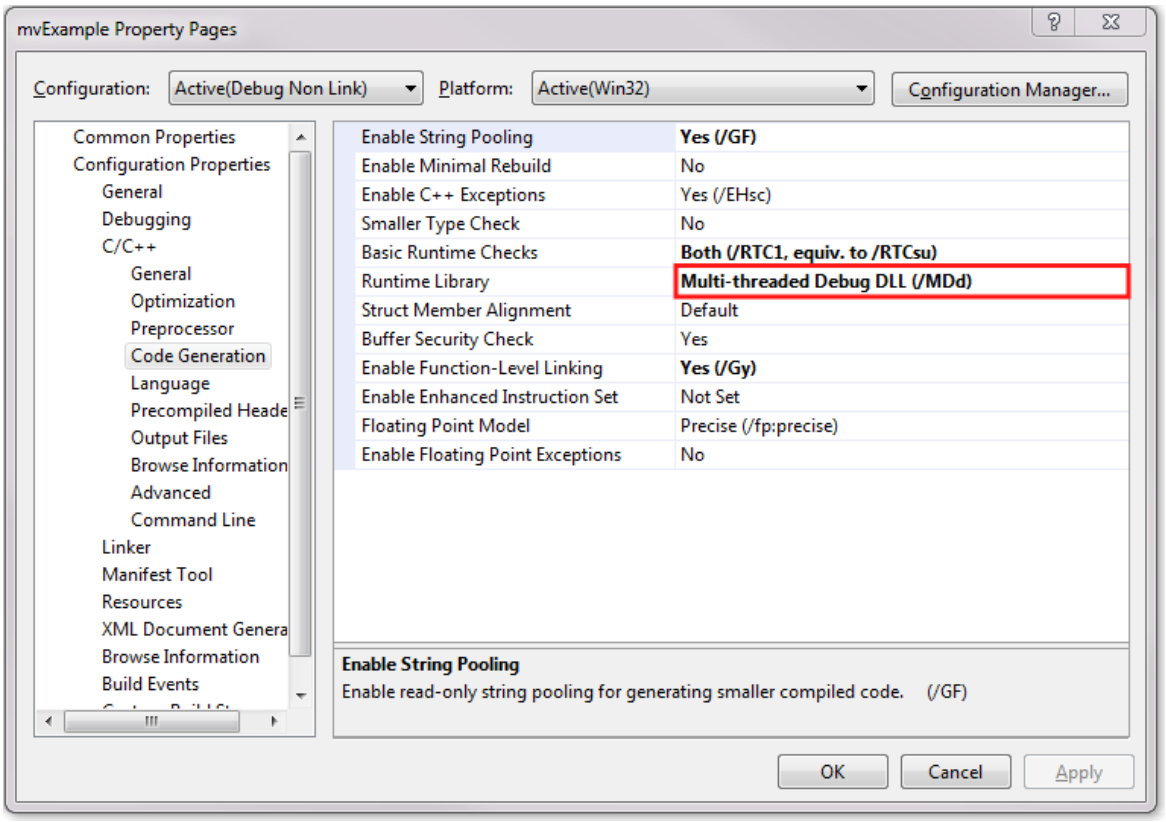


Visual Studio e.g. VS9.0 (2008)

Choose **Project** menu › **[ProgramName] Properties...** › **Configuration Properties** in the list › **C++** › **Code Generation**

If you are using the debug dynamic multi-threaded MSVCRTD.DLL the following will be shown in the **Runtime Library** option (highlighted in red below).

Alternatively, if using the non debug MSVCRT.DLL, the same box will show *Multi-threaded DLL (/MD)*.



CRT Compiler options

The compiler options corresponding to the above choices are:

For *dynamic* multi-threaded CRT linking:

/MD	MSVCRT.DLL
/MDd	MSVCRTD.DLL debug

For *static* CRT linking:

/ML	LIBC.LIB - single threaded
/MLd	LICD.LIB - single threaded debug
/MT	LIBCMT.LIB - multi-threaded
/MTd	LIBCMTD.LIB - multi-threaded debug

Changing from static to dynamic CRT linking

To use dynamic linking you need to either select one of the selections shown in the pictures above, or use the /MD or /MDd compiler flags instead of /ML, /MLd, /MT, /MTd.

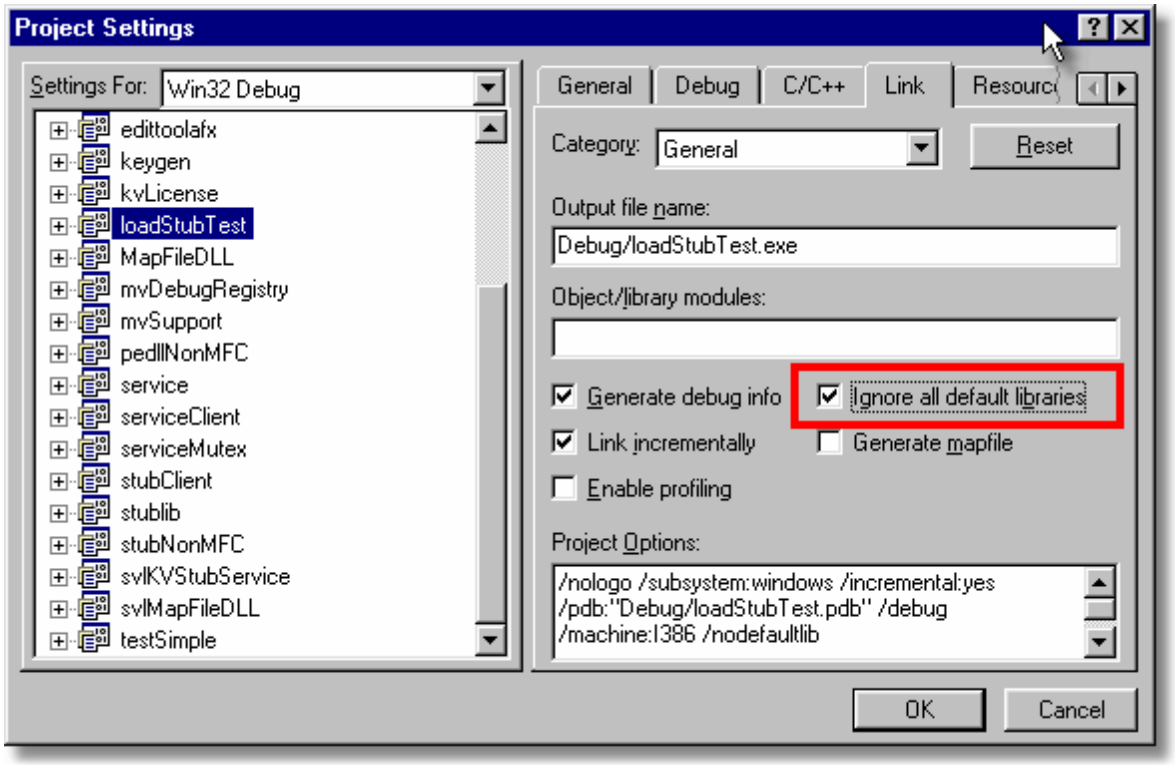
The following shows which option you should use when changing compiler options:

static		dynamic
/ML	›	/MD
/MT	›	/MD
/MLd	›	/MDd
/MTd	›	/MDd

If you are linked statically, you only need to make this change once and rebuild the application. If you are linked dynamically you do not need to rebuild your application.

When rebuilding your DLL(s), you may get warning messages which refer to libraries used by the static linking which you are no longer using. This is caused by some default settings in your project. You can prevent these by disabling the linking of default libraries. You will then need to specify any libraries you require.

In **Developer Studio**, the image below shows the checkbox to select to prevent linking to the default libraries.



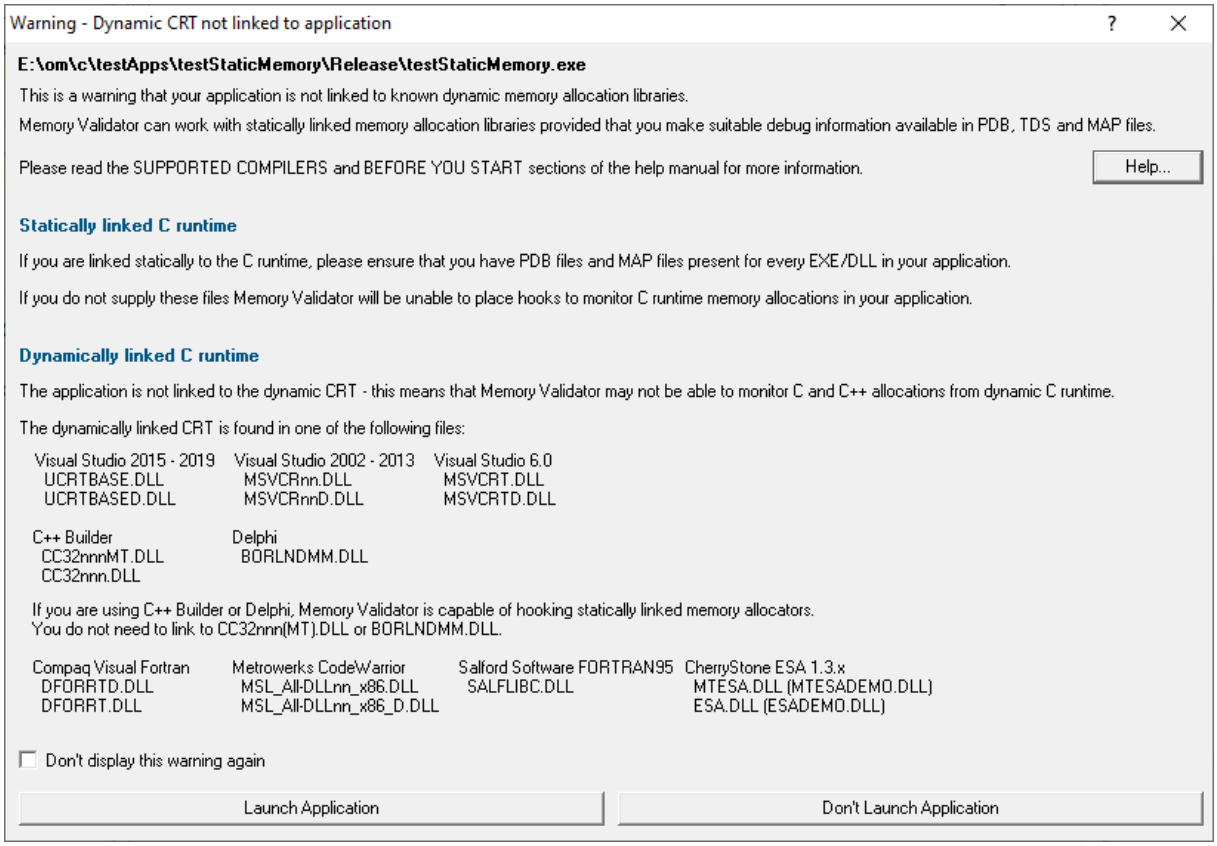
In **Visual Studio 9.0** to show the same option:

Choose **Project** menu › **[ProgramName] Properties...** › **Configuration Properties** in the list › **Linker** › **Input** › **Ignore All Default Libraries**

CRT Warning Dialog

When Memory Validator launches an application it checks to determine if the application is using the dynamic C runtime library (or equivalent).

If the dynamic CRT cannot be detected, a warning dialog is displayed to the user:



Don't panic! It looks worrying, but it's not! It is just a warning. You can still launch your application with Memory Validator.

Many applications do not use the dynamic CRT in the main .EXE, but do use it in the DLLs used by the application. In this case, Memory Validator will monitor CRT allocations in the DLLs, and will monitor non-CRT allocations (handles and other memory types) in the .EXE and the DLLs.

Lucky users of Borland compilers can build their applications statically linked or dynamically linked - either will work.

2.2 Enabling Debugging

To get the best from our tools you will need to enable debugging information for your compiler and your linker.

Detailed instructions are available for these IDEs / compilers:

- Visual Studio
- Visual Basic 6
- C++ Builder
- Delphi

- MingW, gcc, g++
- Dev C++
- Salford Software FORTRAN 95
- Metrowerks Code Warrior

Debug Information Formats

Thread Validator can understand debugging information in the following formats:

- Microsoft Program Database (PDB)
- Turbo Debugger Symbols (TDS)
- COFF
- DWARF
- STABS

The Intel Performance Compiler and Intel Fortran compilers produce symbols in Microsoft's PDB format.

2.3 Quick Start

If you're an admin level user, using Microsoft compilers on a modern OS and already know that you create debug info in your debug and release product, then you're more than likely good to go and dive in! Otherwise, we recommend reading these topics before starting:

- What do you need to run Memory Validator?
- Supported Compilers
- User Permissions

Default collection of data

If you have just installed Memory Validator the default settings:

- *will* enable collection of data about the C runtime heap, Win32 heaps and handles
- *will not* collect data about uninitialized memory, buffer overruns and buffer underruns

Testing on the Example Program

You can test drive the capabilities of Memory Validator using the example program supplied with Memory Validator - **nativeExample.exe**. This program deliberately leaks memory and provides many menu entries that create even more errors, for all memory heap types, handles, uninitialized data, buffer overruns, buffer underruns and demonstrates the API. (You can disable use of the API by undefining a macro in the source code).

Ensure you have debugging information

Your application needs to be compiled to produce debugging information and linked to make that debugging information. Details are available for enabling debugging information with Visual Studio, C++ Builder, Delphi, MinGW, and other compilers.

If you have no debugging information but you do have a Microsoft format MAP file available the MAP file must contain line number information by using the /MAPINFO:LINES linker directive.

Launching

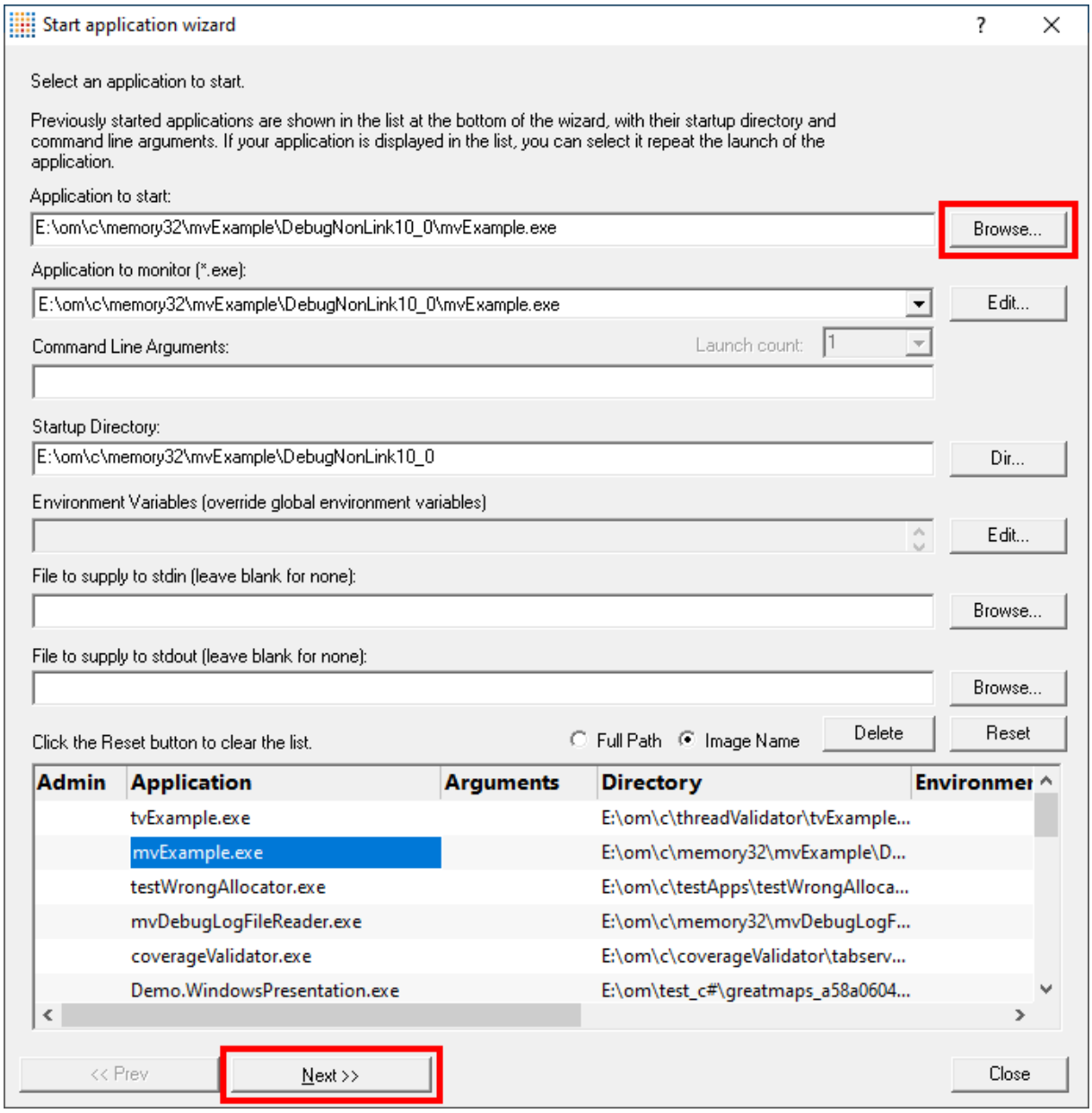
To start your program click on the launch icon on the session toolbar.

What you see next depends on the user interface mode (wizard or dialog style).

The Launch Wizard...

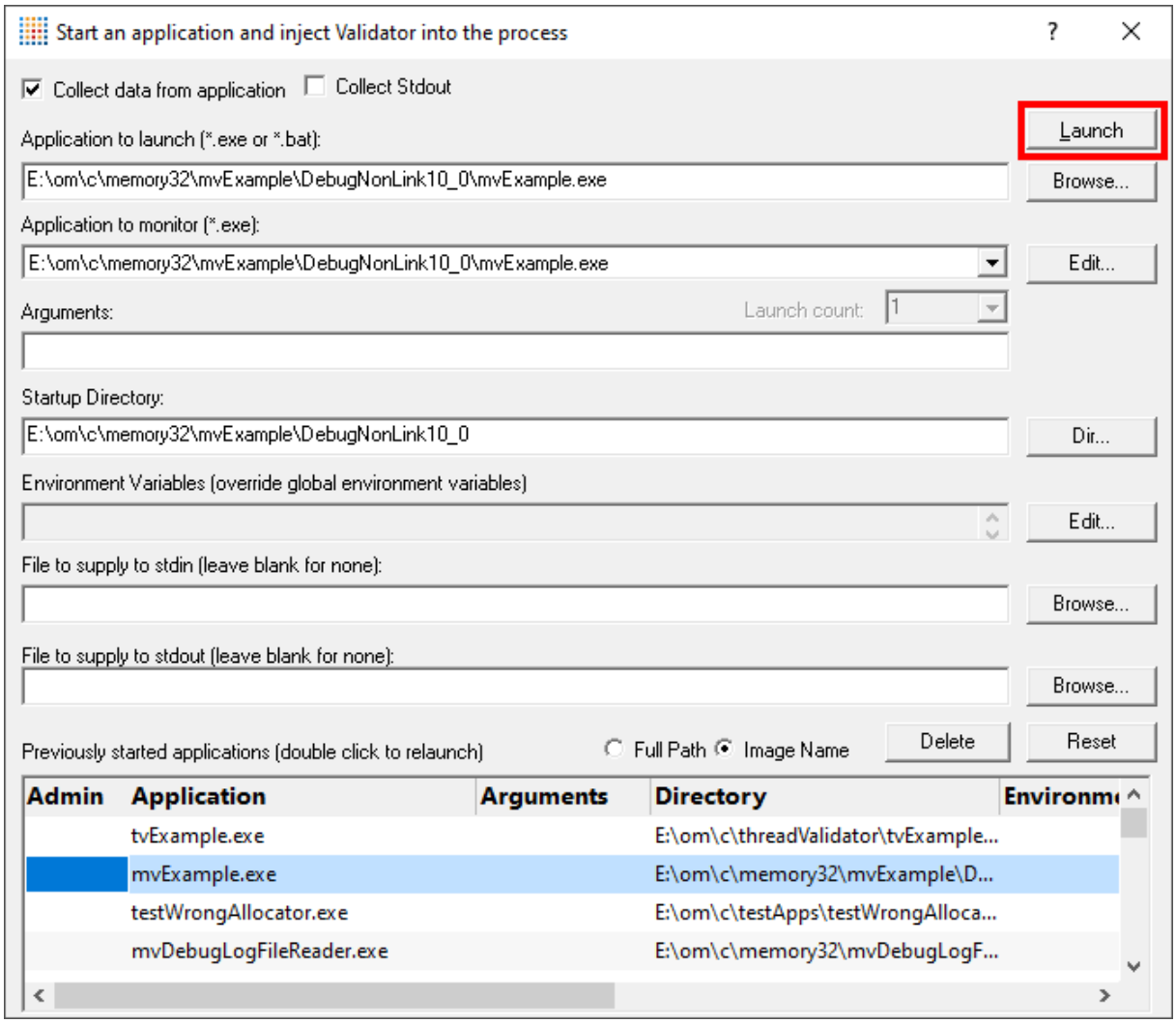
If you have just installed the software you will be shown the launch wizard:

Click **Browse...** to choose a program to launch › **Next** › **Next** › **Start Application...**



...or the Launch Dialog

If you have switched to Dialog mode you will be shown the launch dialog:



During launch

Memory Validator will start and inject the stub into the target program. Progress during this phase is displayed in the header of the main window.

Once correctly installed in the target program the stub will establish communications with Memory Validator and data can be collected and viewed until the target program exits.

Click on the **Refresh** button on the left of the Memory Tab, or click on the refresh icon on the toolbar to get a snapshot view of the errors.

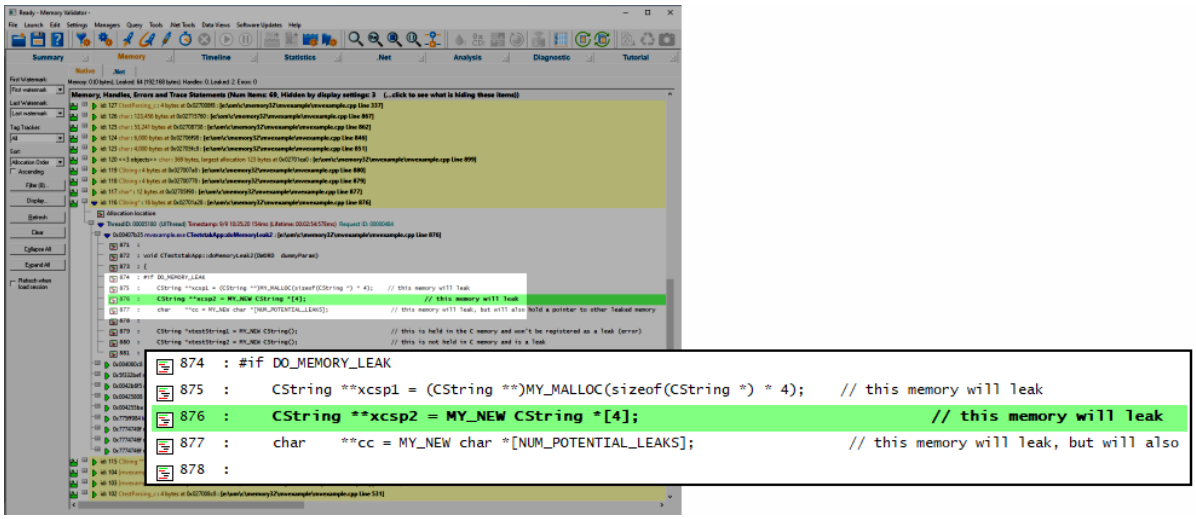
After exit - examining the output

When the target program exits, Memory Validator calculates which data items are leaked. The data collection icons on the session toolbar are disabled, and the launch icons are enabled again.

Click on the **Refresh** button on the left of the Memory Tab, or click on the Refresh icon on the toolbar.

Memory Validator will display data on any outstanding allocations of memory and/or handles. Leaked memory will be shown in yellow, although the colours can be changed from the colour settings tab.

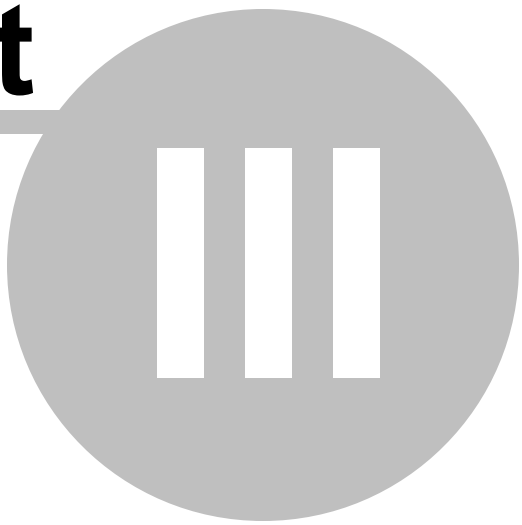
The picture shown below shows the memory tab displaying some memory leaks. One of the leaked objects has had its stack trace expanded showing the source code line responsible for the leak.



Ending the session

Even though the target program has exited, the session is still active and can be examined or saved until the session is closed via the **File** menu › **Close Session**.

Part



3 The User Interface

The part of memory Validator that you get to see is the user interface. Behind the scenes, the stub installs and controls the data hooks in the target program and interacts with the user interface.

This section describes the various functions of the user interface so that you can get the most from using Memory Validator.

Typical workflow

Typical usage of Memory Validator is very simple:

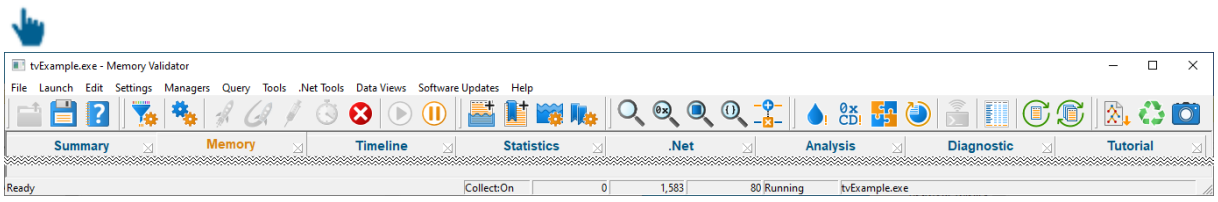
- Start the target program
- Test whatever needs testing on the program
- Close the program
- See what memory leaks and other errors Memory Validator reports

However, there is much more to Memory Validator than this simple workflow. For example, whilst your program is running, you can display data and gain insight into a bug you are looking at in the debugger. You can find which objects are pointing *at* a given object, or which objects are being pointed to *from* a given object. Such capabilities are very useful for finding bad pointers, dangling pointers and so on.

The user interface

The user interface consists of the menus, toolbars, status bar and the main display.

You can read on to find out about all the features, or click parts of the images below to jump directly to the four main section of interest.

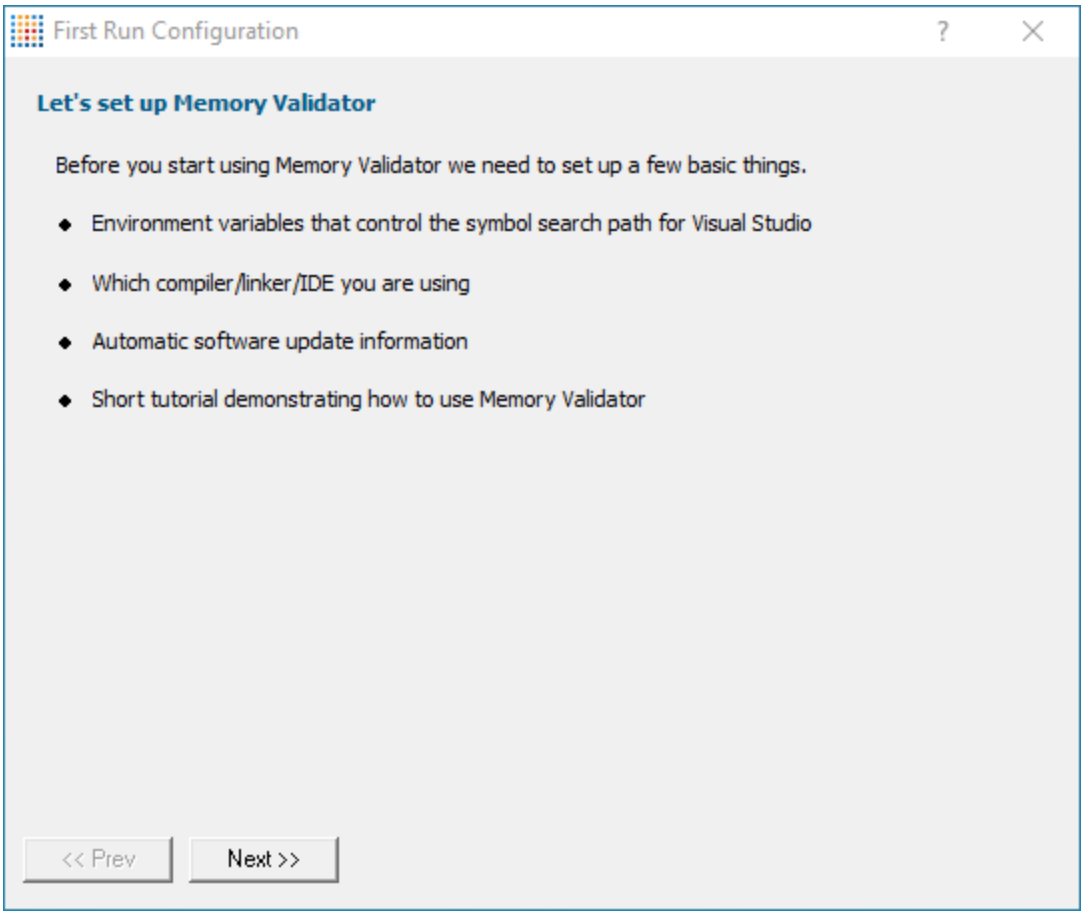


3.1 First run configuration

First run configuration

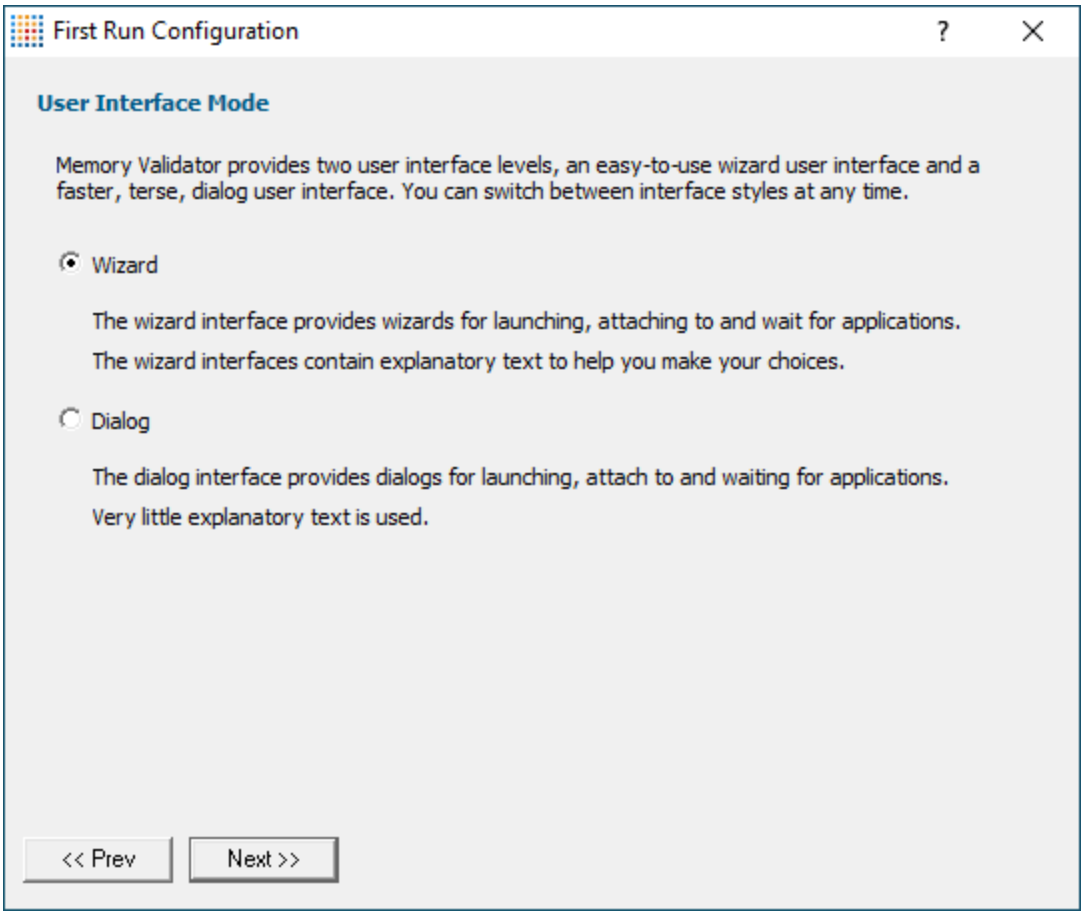
For new users of Memory Validator, a configuration wizard appears the first time you run the application.

The wizard collects a few details about environment, tools, update requirements (for non-evaluation users) and ends with a short video tutorial.



User interface mode

After the introductory page, the wizard presents options for configuring the how the launch, inject and wait dialogs present information to you.



- **Wizard mode** › guides you through the tasks in a linear fashion
- **Dialog mode** › all options are contained in a single dialog

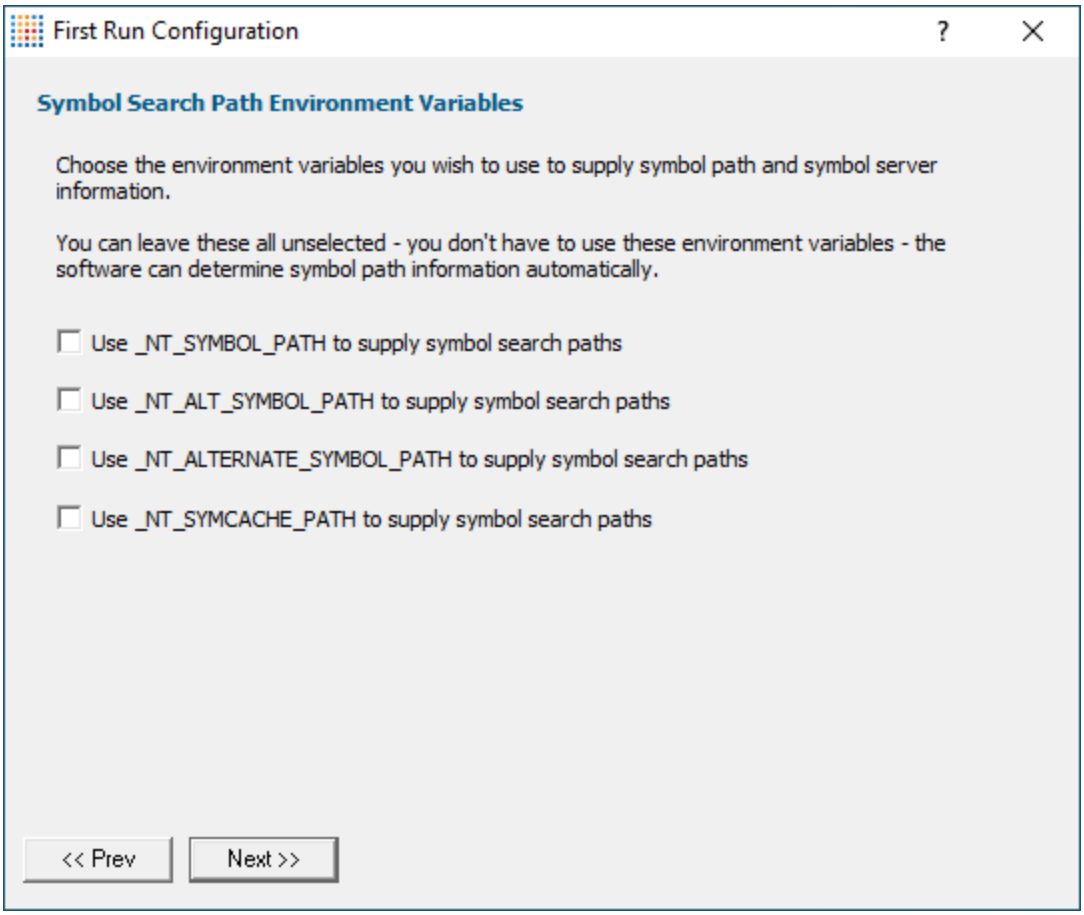
  Experienced users will find this mode quicker to use

These settings can be changed at any time via the User Interface Mode option on the Settings menu.

Symbol search path environment variables

After the introductory page, the wizard presents options for using environment variables for symbol search paths when finding PDB symbols.

You don't *have* to choose any of these options as Memory Validator will try to automatically determine symbol path information.



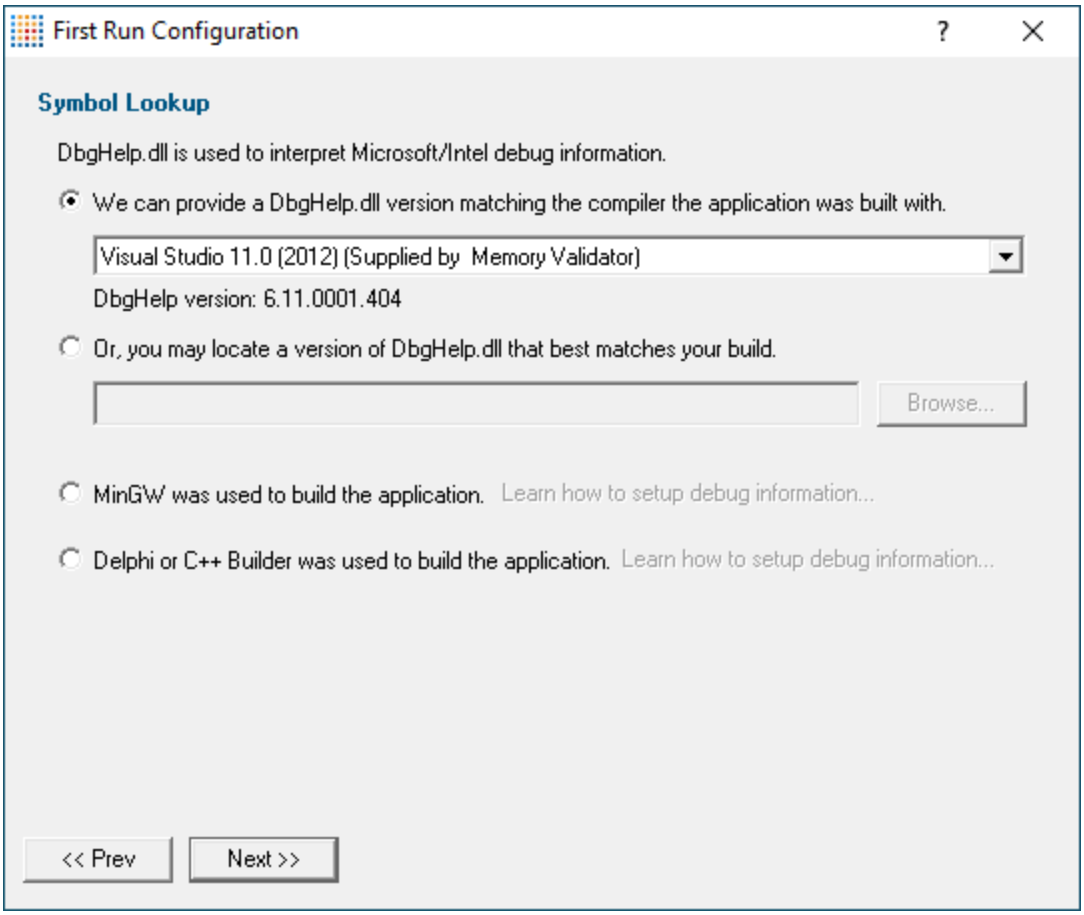
These environment settings can be changed at any time via the Configure Symbol Handling Environment Variables on the Symbol Server page of the Settings Dialog.

Symbol lookup

The next page of the wizard allows you to specify which IDE, Compiler or Linker you're using.

This is important as it affects how symbol lookup is performed. Visual Studio has various quirks in its history of symbol handling and we have to work around that.

The default settings are shown below, although the Visual Studio version may vary.

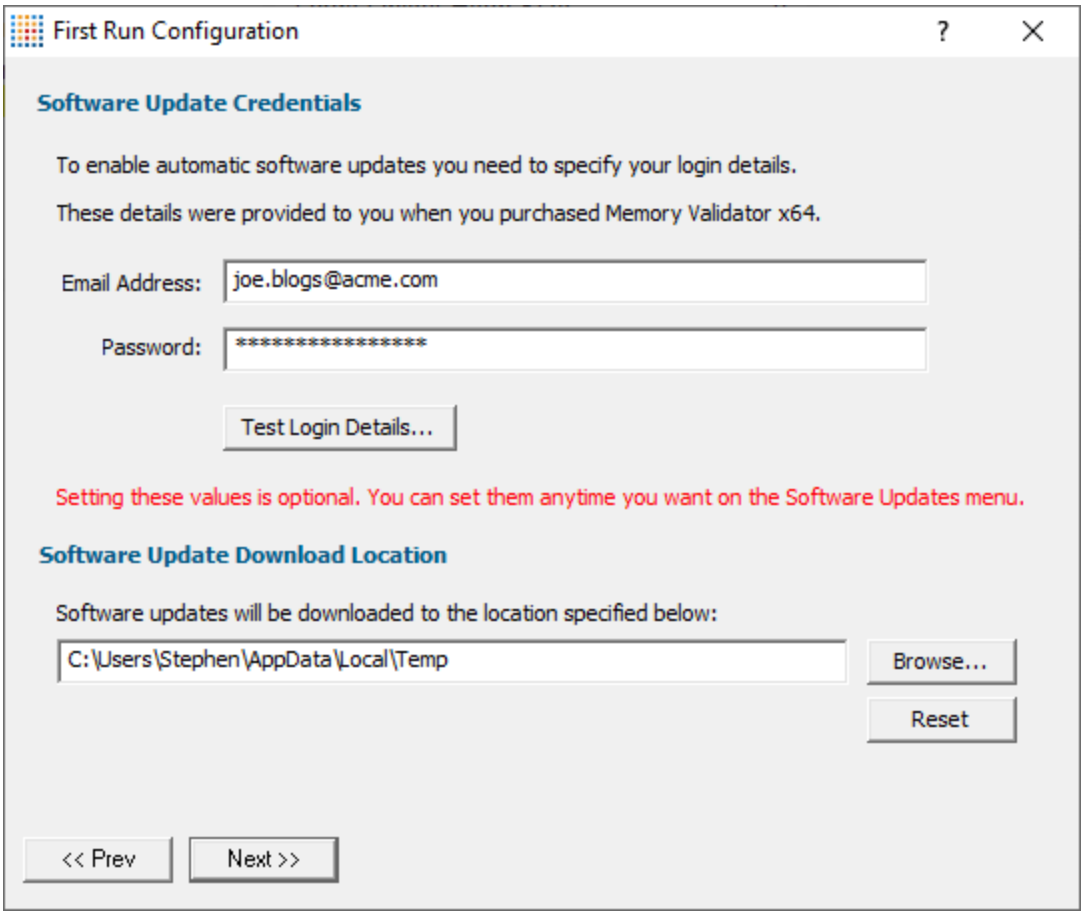


These lookup settings can be changed at any time on the Symbol Lookup page of the Settings Dialog.

Software update credentials

The software updates page of the wizard is only shown to non-evaluation users.

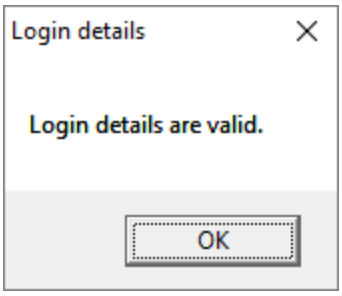
You can configure your software update credentials within the application and where updates are downloaded to.



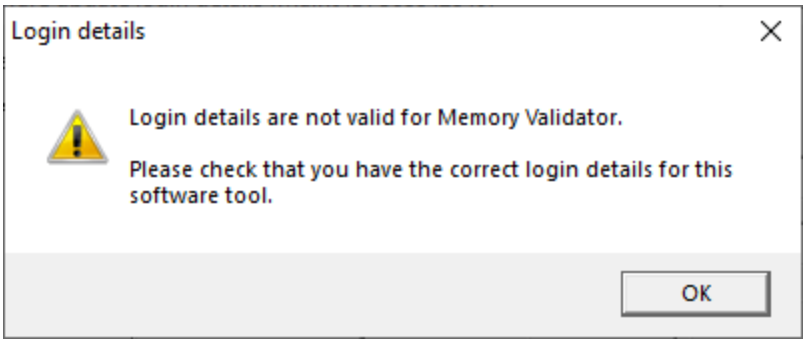
You can test the login details to ensure they are valid:

- **Test login details** › click to check your entered details are valid (requires an internet connection)

  Valid details will be confirmed:



  Invalid details may mean you entered credentials for another application in the Validator suite, or they could have been entered incorrectly.



You should have received the correct credentials when you purchased Memory Validator.

If you experience problems, check with your system administrator or contact Software Verify.

These update credentials can be changed at any time from the Software Updates menu.

Software update download location

It's important to be able to specify where software updates are downloaded to because of potential security risks that may arise from allowing the `TMP` directory to be executable.

We use the `TMP` directory as a default, but if you're not comfortable with that you can specify your preferred download directory. This allows you to set permissions for `TMP` to deny execute privileges if you wish.

An invalid directory, e.g. one that does not exist, will show text in red and will not be accepted until a valid folder is entered.

- **Reset** › reverts the download location to the user's `TMP` directory

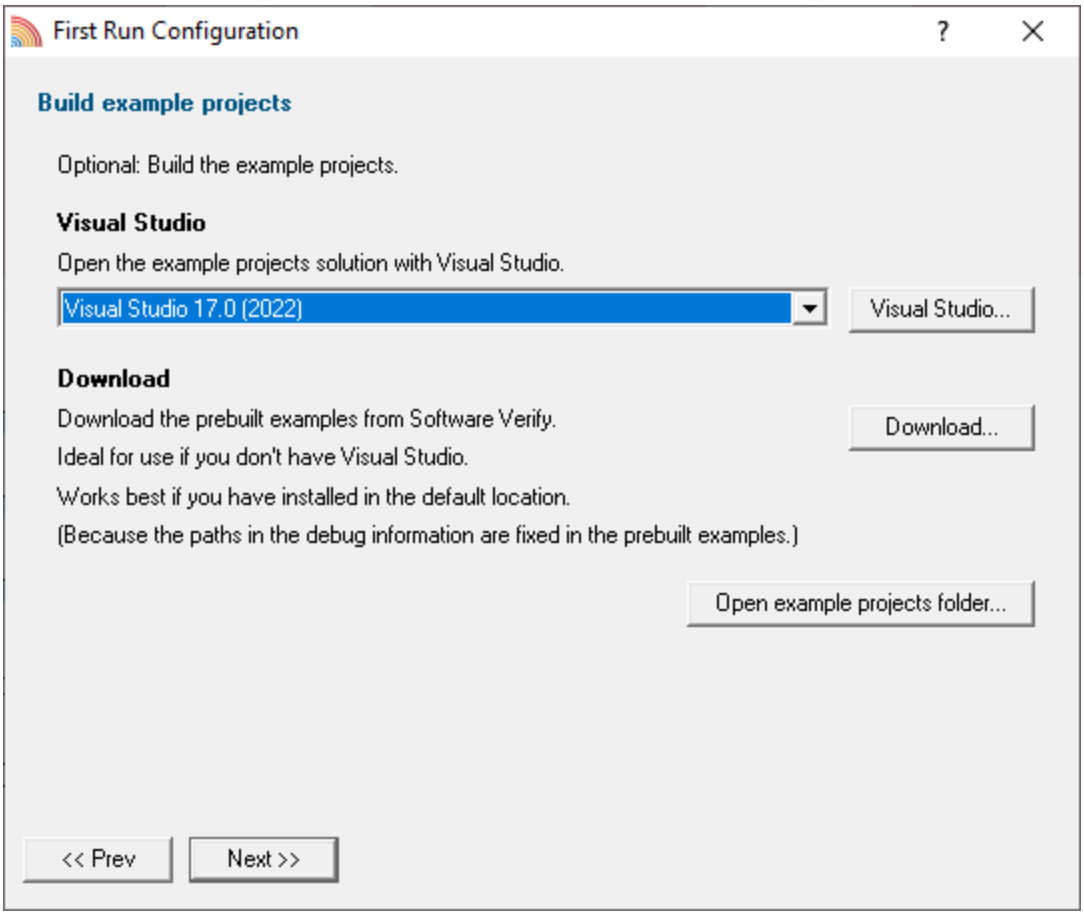
  The default location is `c:\users\[username]\AppData\Local\Temp`

The download location can be changed at any time from the Software Updates menu.

Build example projects

The next page of the wizard allows you to build the example projects that ship with Memory Validator.

The example projects demonstrate various application types containing bug you may wish to investigate with Memory Validator.



- **Visual Studio...** › opens the example projects solution with the version of Visual Studio selected
- **Download...** › downloads a prebuilt version of the example projects, unzips them and installs them in the examples folder in the Memory Validator installation directory

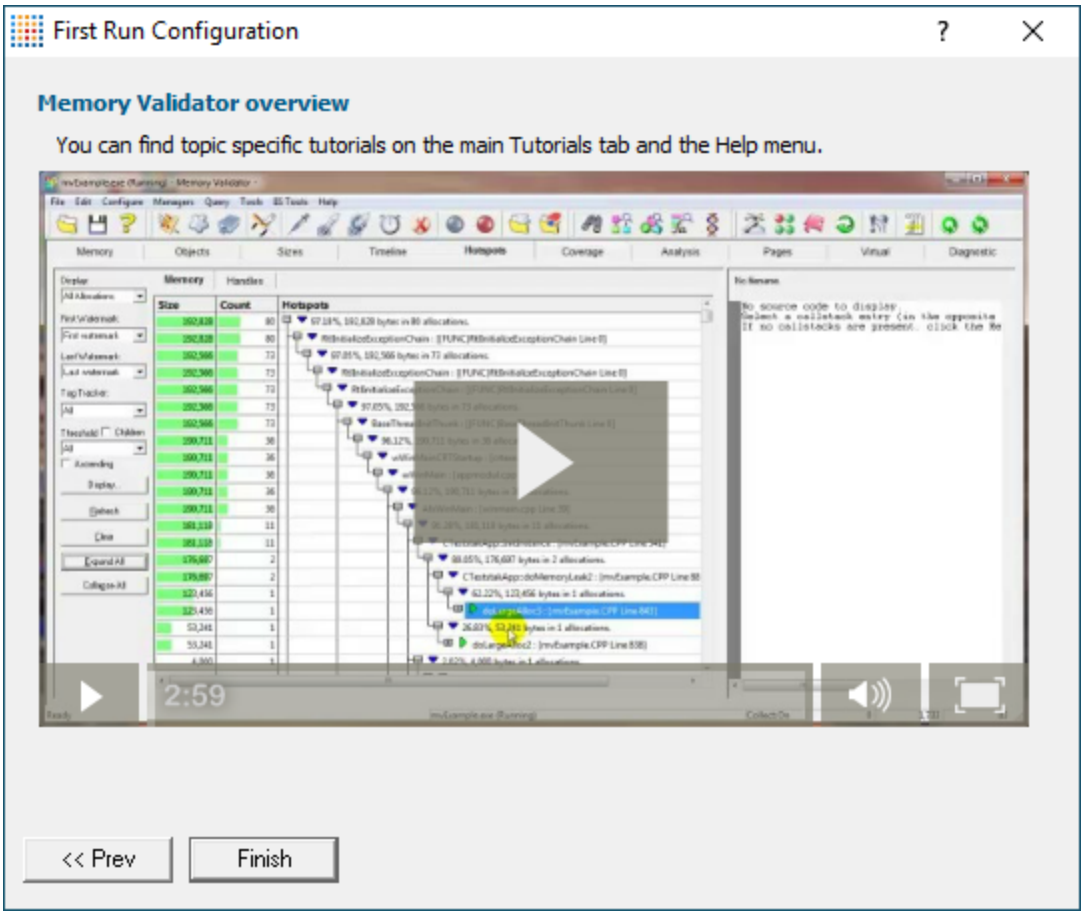
  If you choose this option and you have not installed Memory Validator in the default location (assuming a 64 bit OS) the source file paths in the debug information will be incorrect - you will need to use the File Locations settings to inform Memory Validator of the correct location(s).

- **Open example projects folder...** › opens the folder that contains all example projects

Video overview of application

The final page of the wizard presents a short video overview of Memory Validator.

The video has audio



More help is available via the Tutorials tab and the Help menu.

The video is also available on the product website. Visit https://www.softwareverify.com/products.php and find the product link for Memory Validator.

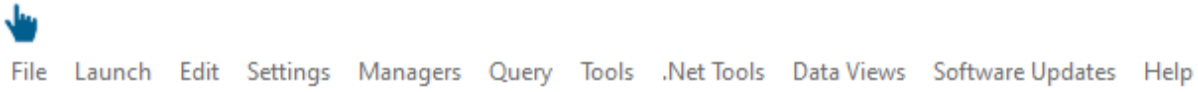
- **Finish** › closes the First Run Configuration dialog leaving the application ready to use

3.2 Menu Reference

The menus provide access to all the major features in Memory Validator. The most common ones are also directly accessible via the toolbars.

The next few pages just provide a convenient collection of links to the detailed help pages on each topic

Click on an item in the picture below to jump to the menu's page:



3.2.1 File menu

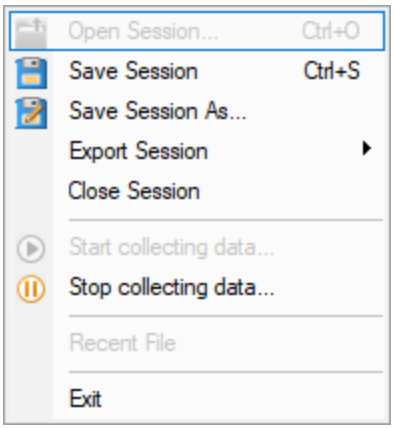
The **File** menu allows you to:

- open, close and save sessions
- manage the launching of an application
- control the collection of data
- exit the application

Most of these actions are also available via the standard or session toolbars.

Near the bottom of the menu, a list of recently used file names allows you to easily reload a previously saved session.

Click on an item in the picture below to find out more:



The last two items are not linked to topics. Exit is self explanatory, and above that is a list of recently opened files.

3.2.2 Launch Menu

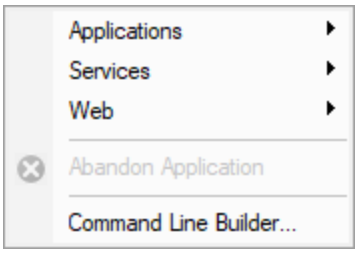
The **Launch** menu allows you to:

- start applications and restart applications
- inject into running applications
- wait for applications to start then attach to them
- monitor services and ISAPI extensions
- stop monitoring an application

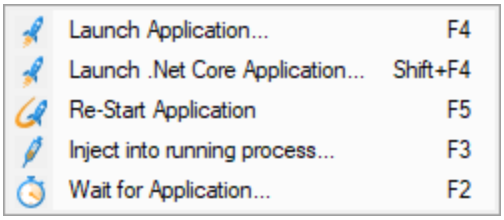
Most of these actions are also available via the standard or session toolbars.

These actions are grouped into submenus according to whether they involve applications or services.

Click on an item in the pictures below to find out more:



Applications



Services

Web

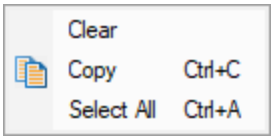
In addition to the function key short cuts shown above, you can redisplay the previously chosen launch dialog by using Ctrl + F4

3.2.3 Edit menu

Selections and the clipboard

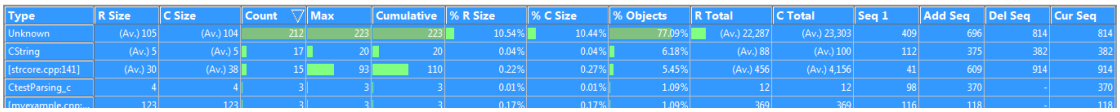
The **Edit** menu options can be used to clear all selected items in a table or tree, copy selected items (and relevant data where applicable) to the clipboard, or select all the items available.

Selected data is formatted into one line per row with a single space used to separate column data.



Select All will include the header row as well as the data, and **Copy** will include the column titles.

For example Select All on the Types tab might show:



Type	R Size	C Size	Count	Max	Cumulative	% R Size	% C Size	% Objects	R Total	C Total	Seq 1	Add Seq	Del Seq	Cur Seq
Unknown	(Av.) 105	(Av.) 104	212	223	223	10.54%	10.44%	77.09%	(Av.) 22,287	(Av.) 23,303	409	696	814	814
CString	(Av.) 5	(Av.) 5	17	20	20	0.04%	0.04%	6.18%	(Av.) 88	(Av.) 100	112	375	382	382
[strcore.cpp:141]	(Av.) 30	(Av.) 38	15	93	110	0.22%	0.27%	5.45%	(Av.) 456	(Av.) 4,156	41	609	914	914
CtestParsing_c	4	4	3	3	3	0.01%	0.01%	1.09%	12	12	98	370	-	370
[myexample.cpp:...	123	123	3	3	3	0.17%	0.17%	1.09%	369	369	116	118	-	118

This would result in the following being copied to clipboard

```
Type R Size C Size Count Max Cumulative % R Size % C Size % Objects R Total C Total Seq 1 A
Unknown (Av.) 105 (Av.) 104 212 223 223 10.54% 10.44% 77.09% (Av.) 22,287 (Av.) 23,303 409
CString (Av.) 5 (Av.) 5 17 20 20 0.04% 0.04% 6.18% (Av.) 88 (Av.) 100 112 375 382 382
[strcore.cpp:141] (Av.) 30 (Av.) 38 15 93 110 0.22% 0.27% 5.45% (Av.) 456 (Av.) 4,156 41 60
CtestParsing_c 4 4 3 3 3 0.01% 0.01% 1.09% 12 12 98 370 - 370
...
```

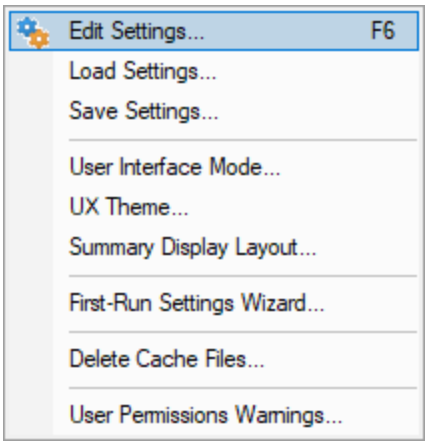
3.2.4 Settings menu

The **Settings** menu allows you to:

- choose the user interface mode (wizards or dialogs)
- change settings for global data and how it is displayed
- modify other application settings

Global settings are also accessible via the toolbar.

Click on an item in the menu below to find out more:



Missing menu options?

If you're looking for the Ordinal to Function mapping, it's now accessed via the Symbols Misc page of the global settings dialog.

The management of datatype and enumeration definitions is now done in the Datatypes and Enumerations page of the global settings dialog.

3.2.5 Managers menu

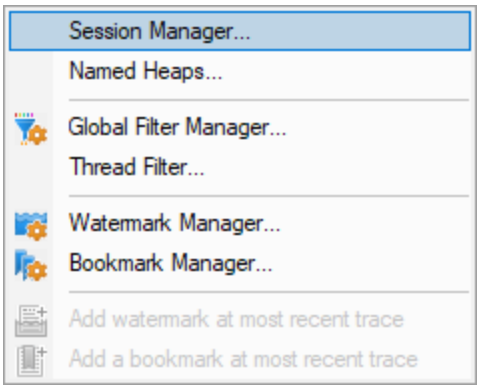
Managers

The **Managers** menu provides a handful of powerful tools to manage or inspect data collected by Memory Validator.

The tools include:

- session management
- global, session, local and thread filters
- named heaps
- watermarks and bookmarks

Click on an item in the picture below to find out more:

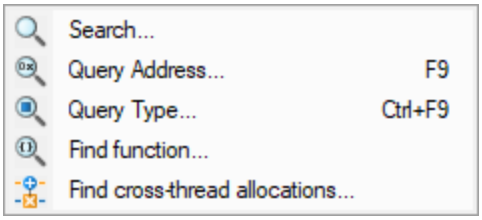


3.2.6 Query menu

Query

The query and search tools help you find memory and handle allocations using different criteria and are all found on the.

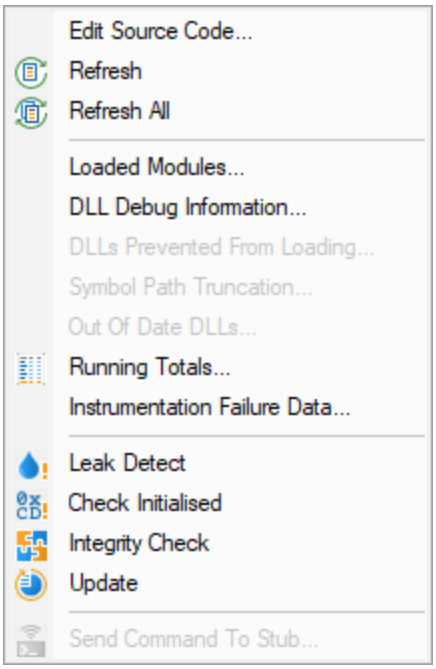
Click on an item in the picture below to find out more:



3.2.7 Tools menu

Tools

Click on an item in the picture below to find out more:



3.2.8 .Net Tools menu

.Net Tools

The query and search .net tools help you inspect the .net memory heap and are all found on the.

Click on an item in the picture below to find out more:

3.2.9 Data Views menu

Data Views

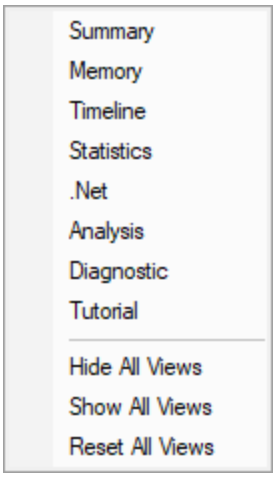
The Data Views provides easy control of which tabs are displayed in the main view.

Selecting any of the items shows the relevant tab (if it's not visible already), and makes it the current selected tab.

- **Hide All Views** › hides all tabs except the one that's currently visible

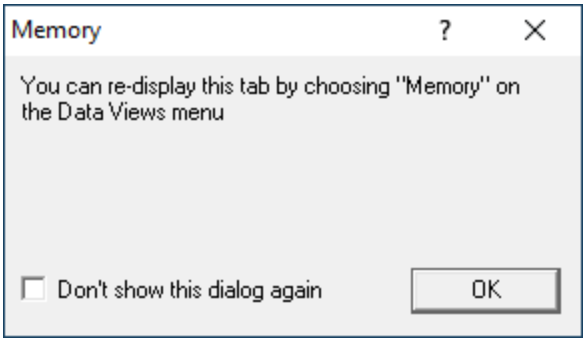
- **Show All Views** › shows all the listed tabs, and in that order
- **Reset All Views** › shows only the most popular tabs, so excludes Coverage, Analysis, Pages, and Virtual

  This is the default setting when you first use the software



When you hide a tab (by clicking the cross on the right of the tab header), you'll initially be reminded of where to go to show it again.

You can choose not to keep seeing this reminder.

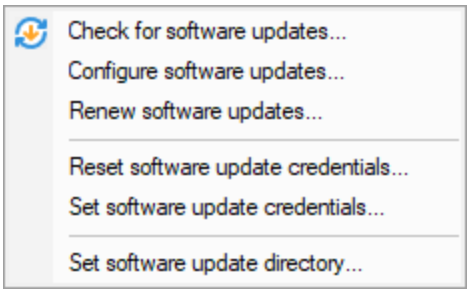


If you hide the Analysis tab, but make use of options to view data in the Analysis tab, it will be shown automatically. The Find menu option on the Memory view is an example of this.

Hidden views are remembered between sessions.

3.2.10 Software Updates menu

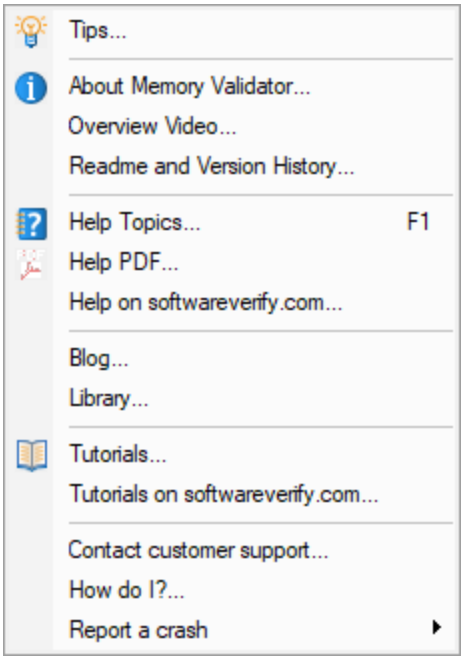
All the items in this menu are covered in the Software Updates topic.



The Software Updates menu is not present in evaluation versions of the software.

3.2.11 Help menu

Click on an item in the picture below to find out more in the Help topic:



If you're looking for more help, check out the Frequently Asked Questions too!

3.3 Toolbar Reference

This reference section lists the various toolbars in Memory Validator, linking to the appropriate section of the help manual.

The items are listed in left to right order.

Click on any part of the pictures below to jump straight to the topic:

Standard toolbar

- Load session
- Save session
- Help

Session toolbar

- Filter manager
- Settings
- Launch application using the launch chooser
- Relaunch the previously launched application
- Inject into application
- Wait for application to start
- Stop application
- Enable collecting data
- Disable collecting data

Watermarks

- Watermark manager
- Bookmark manager
- Add watermark at most recent trace
- Add bookmark at most recent trace

Query

- Search
- Query address
- Query object
- Find function
- Find cross-thread allocations

Tools

- Leak Detect
- Check initialised
- Integrity check
- Update
- Send command to stub extension DLL
- Running totals display
- Refresh view
- Refresh all views

.Net Tools

- Heap Dump
- Garbage Collect
- Snapshot

3.4 The status bar

Elements of the status bar

The status bar has three main sections, from left to right:

- the message line
- program information
- data collection statistics

The message line

Most of the time, you'll just see this:

When you hover the mouse over a toolbar button or a menu item for a short time, a message appears in the status bar describing the button's action.

Data collection statistics

The data statistics counts give a crude indicator of how data is being collected by the stub and sent to Memory Validator.

The collection data has four items

- Status indicating whether collection is currently on or off
- the number of data items received from stub waiting to be processed
- the number of data items processed
- the number of memory or resource allocations and miscellaneous data items not yet resolved

In the example below, collection is on, with 60 items pending processing, 4943 having been processed and 149 items waiting to be deallocated:

The boxes stay gray when the values are static, but will be coloured for a few seconds when the value changes

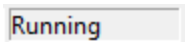
The value increased
The value decreased

Once data collection is off, no more data will arrive in the pending processing queue.

Profiling status

The status of the flow trace indicates what is currently happening.

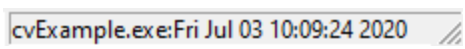
- **Ready**. Waiting to start a run, or a run has finished and is waiting for you to analyze the data.
- **Starting**. Starting a run (hooks being installed etc).
- **Running**. Target executable is hooked and running.
- **Terminating**. Target executable has entered ExitProcess but has not yet finished executing.
- **Post Processing**. Target executable has finished executing. There is data that still needs to be processed.



Target program name

This displays **No active session** when there is no session running, terminating or loaded.

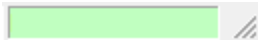
When a session is running, terminated or loaded, this displays the name of the target program followed by a timestamp.



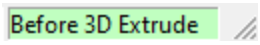
Watermark / Bookmark

This display is empty when there is no session running, terminating or loaded.

It is also empty when a session is running and no watermark or bookmark has been set via the native API.



When a session is running, terminated or loaded, if a watermark or bookmark has been set via the native API, the name of the most recent watermark or bookmark is displayed in this field.



3.5 Keyboard Shortcuts

Keyboard shortcuts

The following shortcuts are available:

Ctrl + A Select All

Ctrl + C Copy

Ctrl + O Open session

Ctrl + S Save session

F1 Help (contextual for current view or dialog)

F2 Wait for application

F3 Inject into process

F4 Start application (Native / .Net)

Shift + F4 Start application (.Net Core)

F5 Restart application

F6 Monitor a Service

F7 Monitor IIS and ISAPI

F8 Monitor IIS and ASP.Net

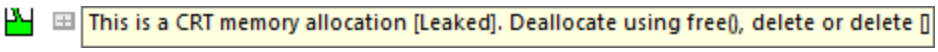
F9 Monitor Web Development Server and ASP.Net

Ctrl + F4 Redisplay the previously chosen launch dialog.

3.6 Icons

Some of the displays show icons on the left border to indicate the type of data associated with that line.

Explanatory tooltips are shown over the icons in the Memory and Analysis views:



Common icon shapes

There are a few recurring shapes you will find in the icons:

allocation
deallocation
reallocation

List of icons used in the main displays

The full suite of icons are shown below in groups, and with a very brief explanation.

General

option enabled
option disabled
line corresponding to allocation/reallocation/deallocation
source code

CRT Heaps

malloc(), calloc(), new
free(), delete
realloc()

Win32 Heaps

HeapAlloc()
HeapFree()
HeapRealloc()

Handles

handle creation
handle destruction

GlobalAlloc

GlobalAlloc()
GlobalFree()
GlobalRealloc()

LocalAlloc


LocalAlloc()

LocalFree()

LocalRealloc()

CoTaskMemAlloc


CoTaskMemAlloc()

CoTaskMemFree()

CoTaskMemRealloc()

IMallocSpy


IMallocSpy allocation

IMallocSpy deallocation

IMallocSpy reallocation

SysAllocString (BSTR)


SysAllocString()

SysFreeString() (and ::VariantClear() used to free BSTRs)

SysReAllocString()

Misc Allocations


miscellaneous allocation

miscellaneous deallocation

miscellaneous reallocation

NetApiXXX


NetApiBufferAllocate() or NetApiXXX() function allocating memory

NetApiBufferFree deallocation

NetApiBufferReallocate reallocation

CryptoAPI


CryptoAPI allocation

CryptoAPI deallocation

CryptoAPI reallocation

COM


COM object created

COM QueryInterface()

COM AddRef()

COM Release()

VirtualAlloc


VirtualAlloc()

VirtualFree()

TRACE and OutputDebugString


trace message, or OutputDebugString() message

trace from an exception

trace from an ASSERT

Errors


bad allocation

bad deallocation

bad reallocation

double free(), or double delete

mismatch between delete and free(), eg used delete when should have used free(), or vice versa

mismatch deleting array, eg used delete [] when should have used delete, vice versa

memory underrun

 memory overrun
 free memory write
 free memory read
 uninitialized memory
 memory error

User Allocations from API

 user allocation (mvUserCustomAlloc())
 user deallocation (mvUserCustomFree())
 user reallocation (mvUserCustomReAlloc())
 user increase reference count (mvUserCustomRefCountIncrement())
 user decrease reference count (mvUserCustomRefCountDecrement())

Coverage

The following symbols are used by the coverage user interface to describe the types of allocations

 malloc()
 calloc()
 realloc()
 _expand()
 free()
 operator new
 operator delete
 handle creation
 handle destruction

Other

 watermark placed by a user, or by mvSetWatermark() API function
 memory leak used by filter managers to show the filtering of an allocation of some

3.7 The main display

Windows

The main display of Memory Validator consists of tabbed windows. Not all the tabs may be visible - see the Data Views menu to show any hidden tabs.

Each window allows the data collected to be viewed, inspected and queried in different and complimentary ways.

Typically usage might be to use one window to analyse a problem that is present in the target program, but to then use another view to gain different insights.

 Click on an item in the picture below to find out more about each of the tabbed windows, or use the list further below:

Local window settings

Some of the windows have local settings - ie settings that affect only that window, rather than the application as a whole, and help on these are also linked below:

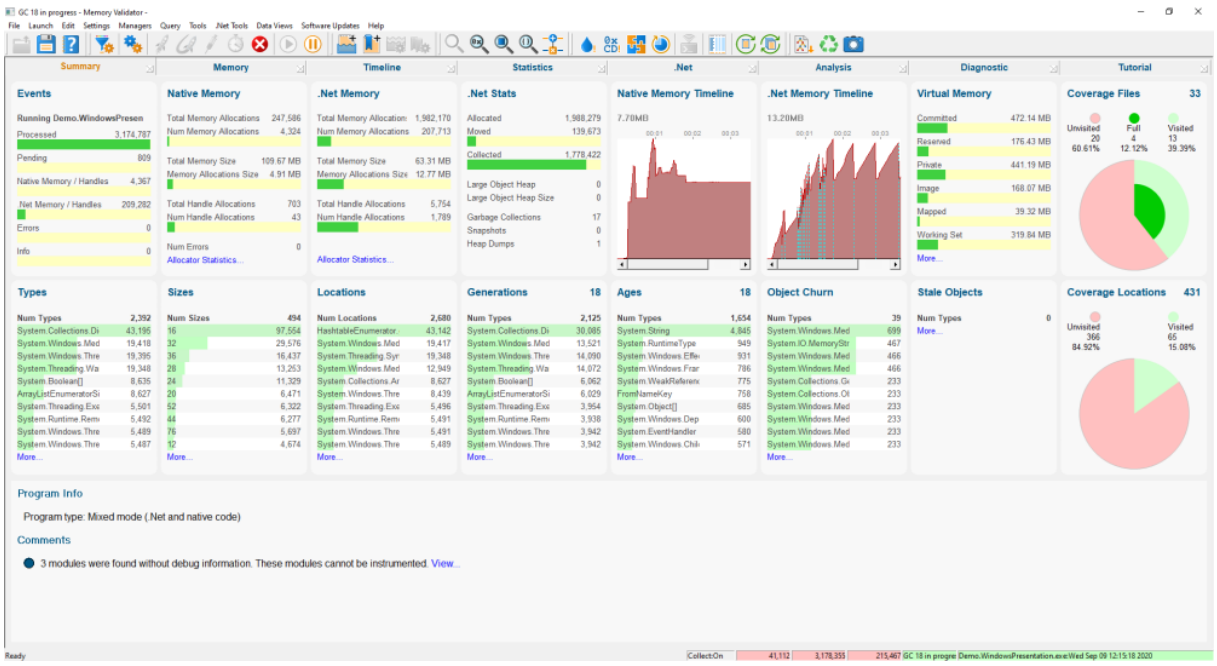
Window	Settings
Memory	display
Types	
Sizes	
Timeline	
Hotspots	display
Coverage	
Analysis	display
Pages	
Virtual	
Diagnostic	

Icons

Most windows use icons to indicate different types of data. You can browse the icons that you'll see.

3.7.1 Summary

The **summary** tab provides a high level overview of all memory and handle activity in the program.



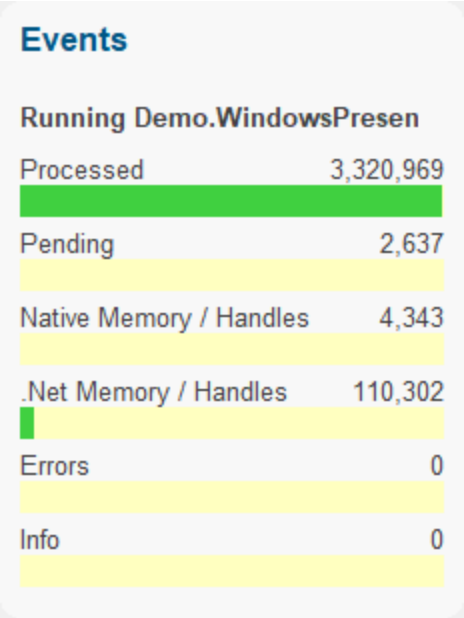
The display is divided into many panels that contain bar graphs, dials or graphs. Some of the panels also contain hyperlinks.

Clicking any bar graph, graph or dial will take you to a more detailed version of the same data.

Hyperlinks usually open a new data display or take you to a more detailed version of the same data.

Bar graphs are usually shown in green/yellow, except when there are error conditions for that statistic, when they are shown in red.

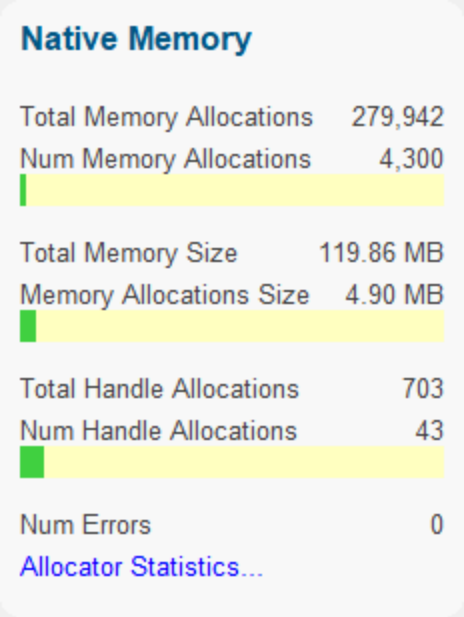
Events



The Events panel gives you an overview of the profiling status, incoming data stream and a high level summary of any errors.

Additional information is in the status bar.

Native Memory

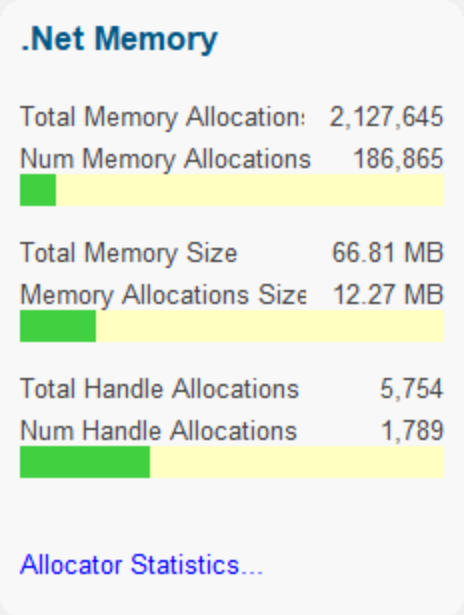


The Native Memory panel gives you an overview of all native memory allocations and native handle allocations.

More detailed information is available on the Memory tab.

The Allocator Statistics hyperlink will display the Running Totals dialog

.Net Memory

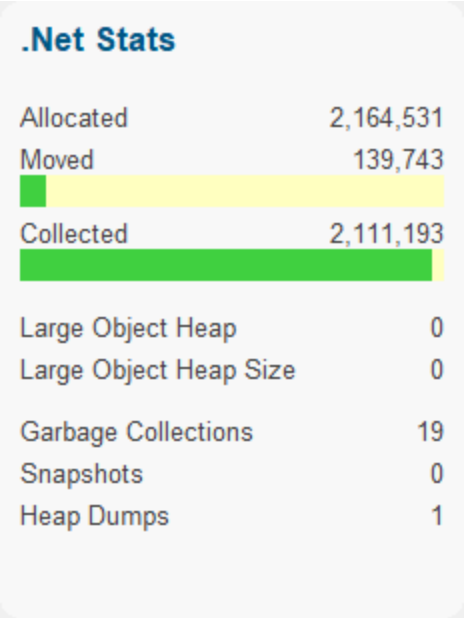


The .Net Memory panel gives you an overview of all .Net memory allocations and .Net handle allocations.

More detailed information is available on the Memory tab.

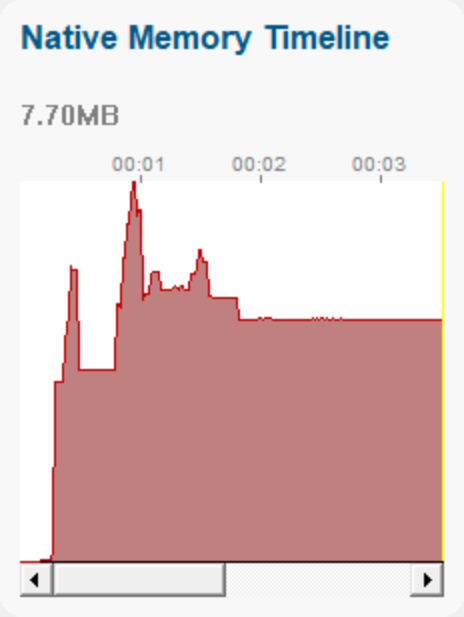
The Allocator Statistics hyperlink will display the Running Totals dialog

.Net Stats



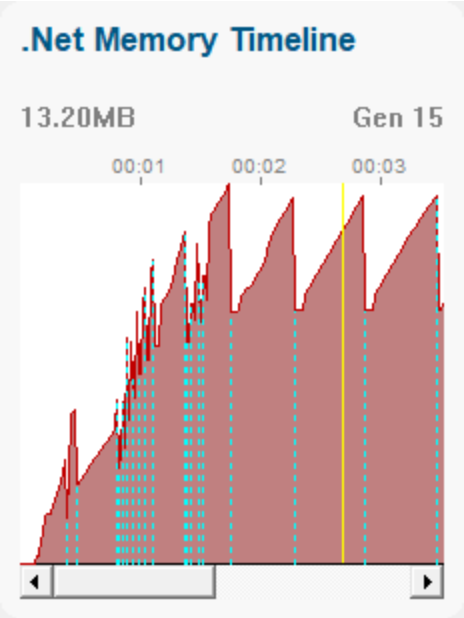
The .Net Stats panel gives you an overview of .Net allocations, garbage collections, snapshots and heap dumps.

Native Memory Timeline



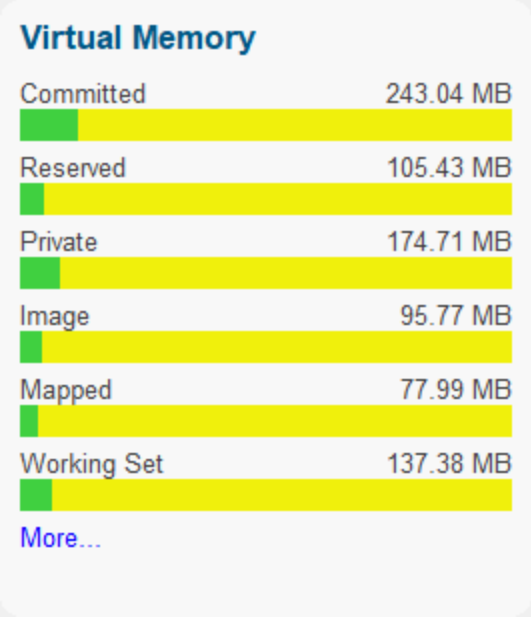
A graphical representation of native memory allocation behaviour. A more detailed version is available on the Timeline tab.

.Net Memory Timeline



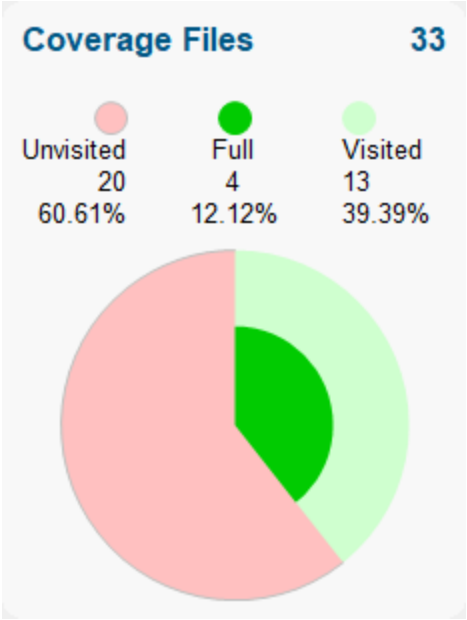
A graphical representation of .Net memory allocation behaviour. A more detailed version is available on the Timeline tab.

Virtual Memory



An overview of virtual memory statistics. A more detailed version is available on the Virtual tab.

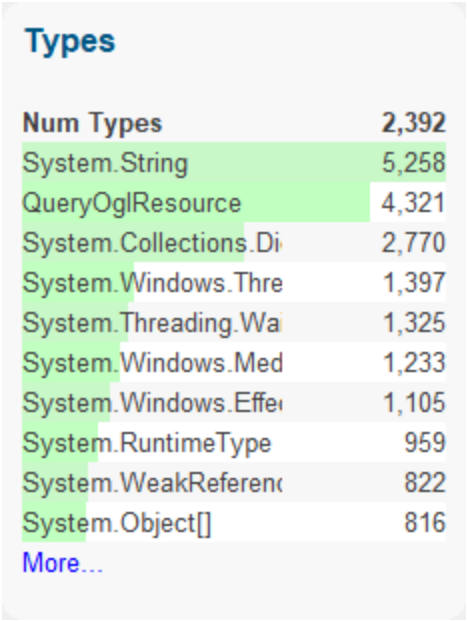
Coverage Files



An overview of coverage of memory allocation locations by filename. A more detailed version is available on the Coverage tab.

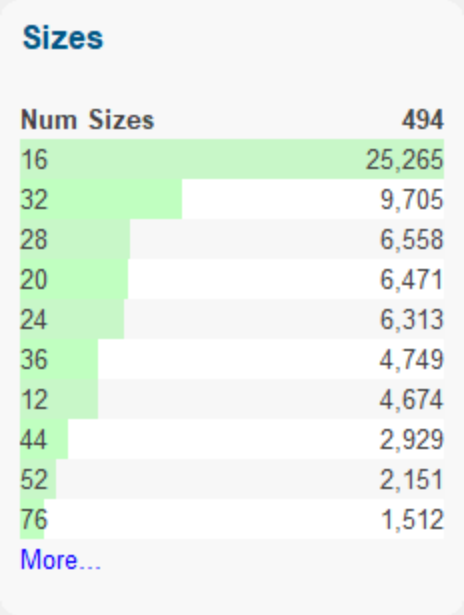
The Coverage Files tile will only be shown if there is memory coverage data to display (Memory Coverage needs to be enabled).

Types



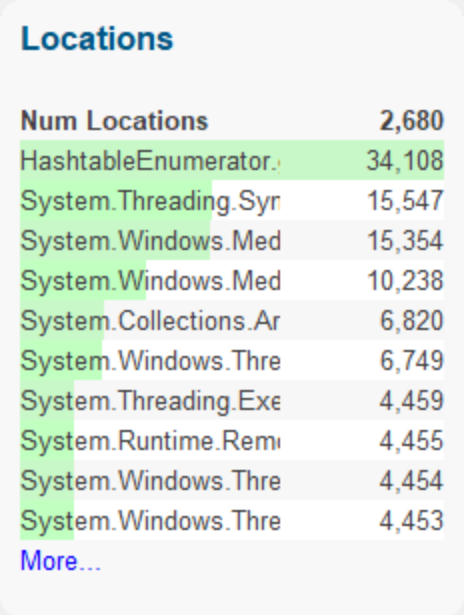
An overview of all the memory allocation types used by the target program. A more detailed version is available on the Types tab.

Sizes



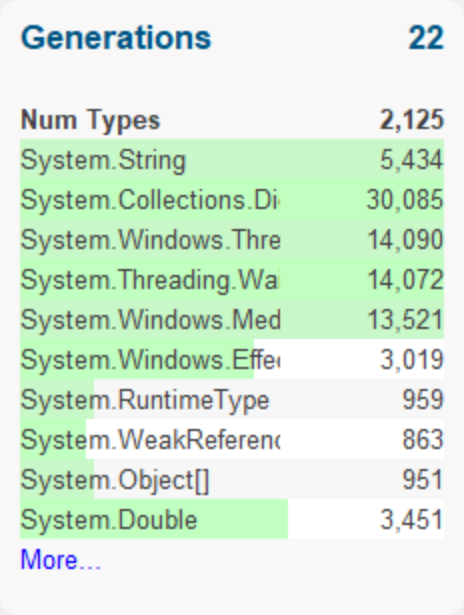
An overview of all the memory allocation sizes used by the target program. A more detailed version is available on the Sizes tab.

Locations



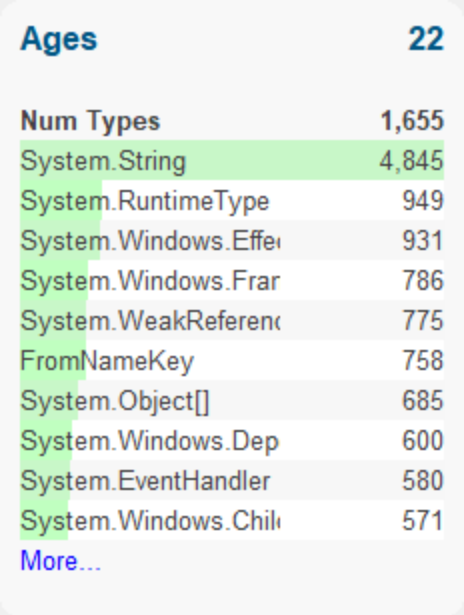
An overview of all the memory allocation locations used by the target program. A more detailed version is available on the Locations tab.

Generations



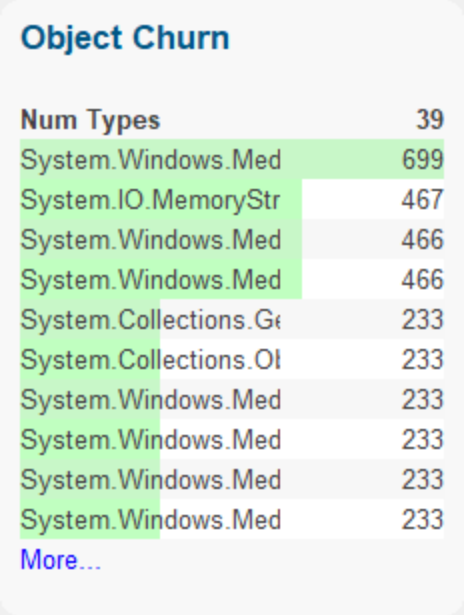
An overview of the generations for the types used by the target program for the most recent .Net generation. A more detailed version is available on the Generations tab.

Ages



An overview of the ages of the types used by the target program for the oldest aged .Net objects in the target program. A more detailed version is available on the Ages tab.

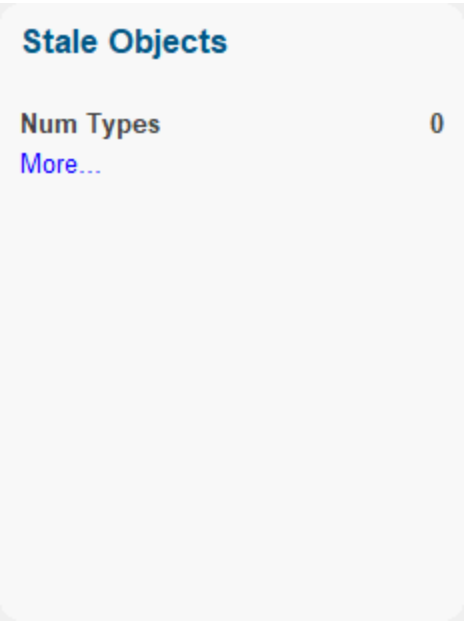
Objects Churn



An overview of the object churn for the types used by the target program. These may indicate leaking objects.

A more detailed version is available on the Object Churn sub-tab of the Ages tab.

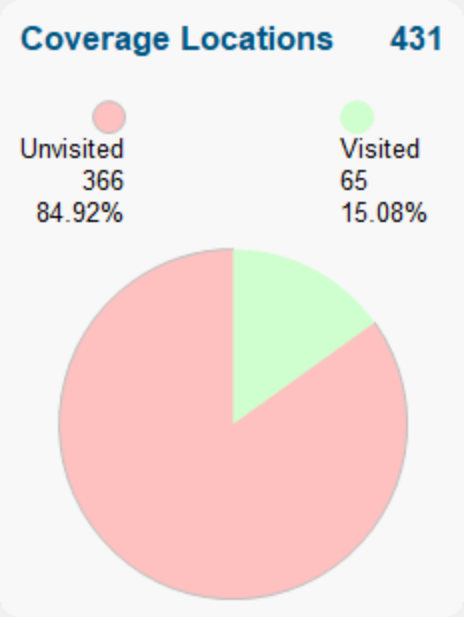
Stale Objects



An overview of the stale object types used by the target program. These may indicate leaking objects.

A more detailed version is available on the Stale Objects sub-tab of the Ages tab.

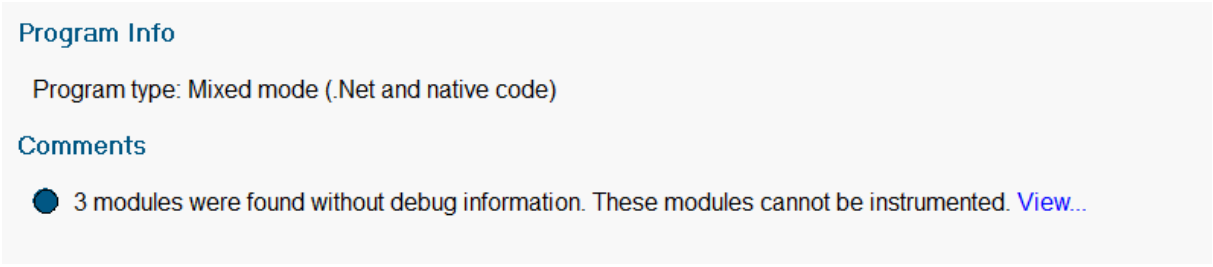
Coverage Locations



An overview of coverage of memory allocation locations. A more detailed version is available on the Coverage tab.

The Coverage Locations tile will only be shown if there is memory coverage data to display (Memory Coverage needs to be enabled).

Instrumentation Status



An overview of profiling settings and status information that will aid your understanding of the current profiling results.

3.7.2 Memory

The **Memory** tab provides tabs for native and .Net memory inspection.

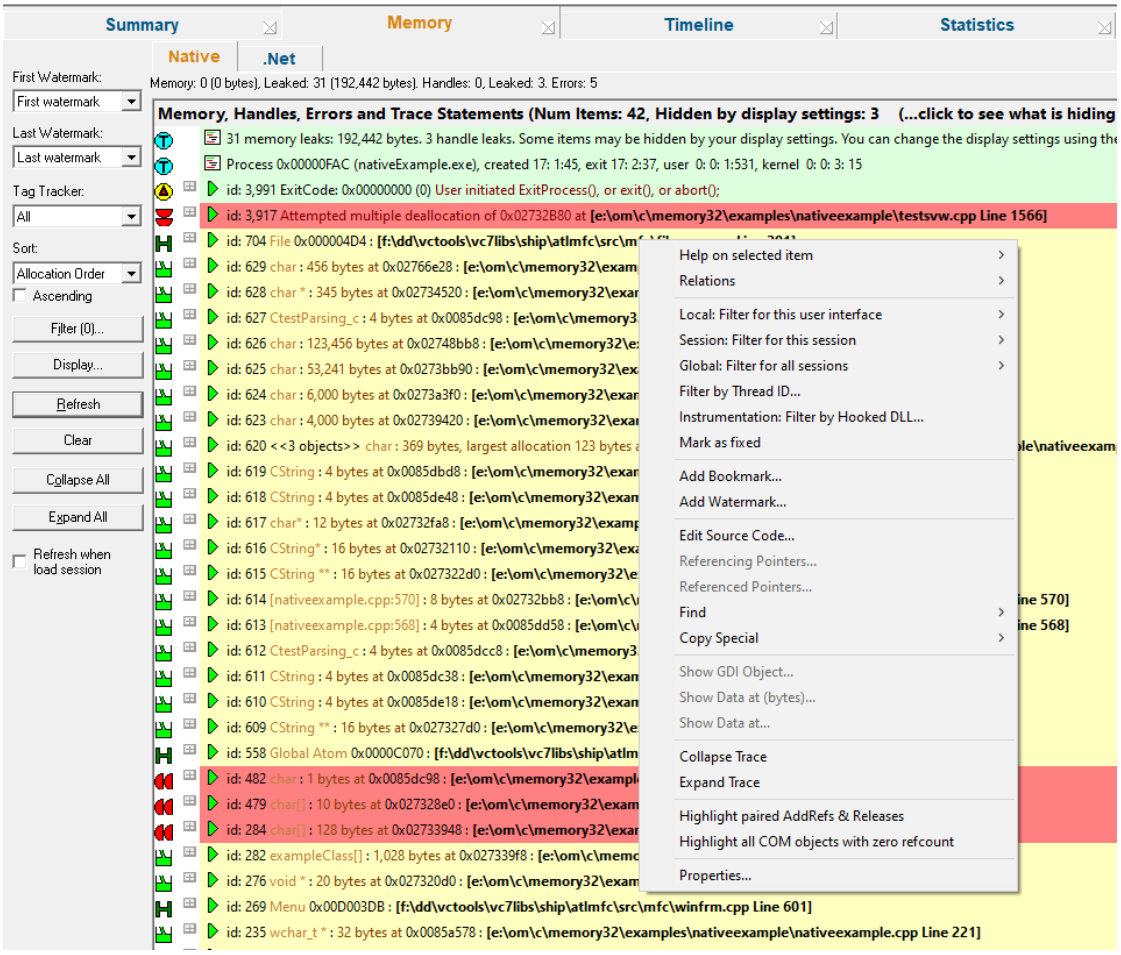
Click on an item in the picture below to find out more about each of the tabbed windows, or use the list further below:

3.7.2.1 Memory and handle leaks

The **Native Memory** tab displays native memory and native handle allocations that are still waiting to be deallocated.

This memory view also displays leaked memory, handles and memory errors, such as double frees, incorrect frees, uninitialized data etc.

Read on, or click a part of the image below to jump straight to the help for that area (the icons link to a different page).



Status Information

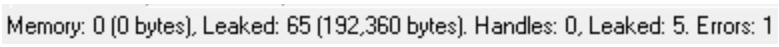
The very top line of the memory tab gives a brief summary of:

- memory allocated and number of blocks
- memory leaked and number of blocks
- handles allocated and leaked
- the number of errors

The status line is only updated when the Refresh button is pressed or when data is added to the display automatically.

More up-to-date information can be found on the Running Totals Dialog or the Types view.

This example shows a status line after the target application has exited (hence 0 bytes for *current* memory):

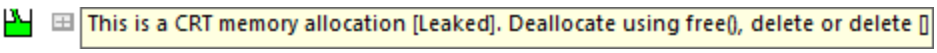


Collected data

The main display shows the *live* collected data during or after the application running. Memory that has been allocated and then freed is not shown.

 Initially no data will be displayed until the data is manually refreshed or the automatically updated with significant errors as they happen.

Each line has an icon at the left, indicating its type, and has an explanatory tooltip:

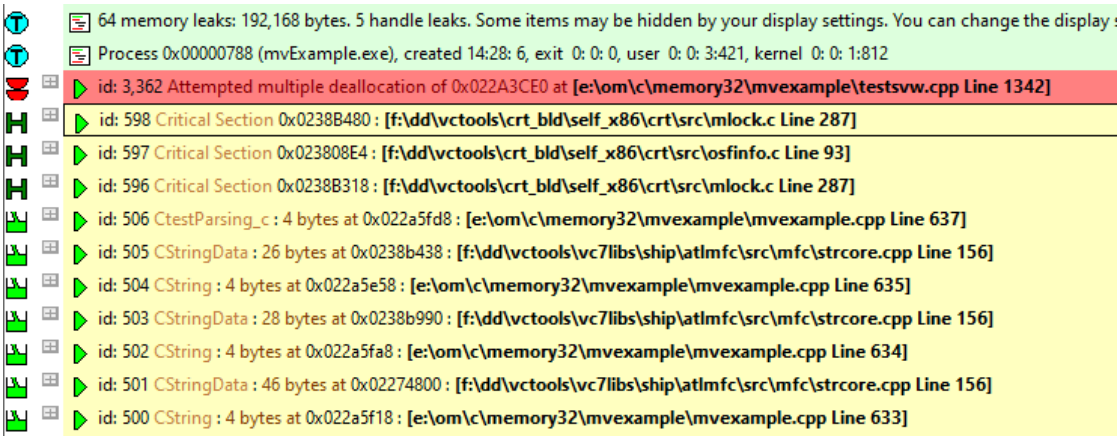


The text on each line indicates:

- datatype (if known)
- size
- allocation address/handle value
- source file and line number (if available) where the allocation occurred
- optional event sequence id at the beginning of the line

The background colour for each line indicates the status of the data - eg, leaked, damaged, or uninitialized.

This example shows memory allocations and handles in yellow are leaked, while the red line shows a serious occurrence of damaged memory.



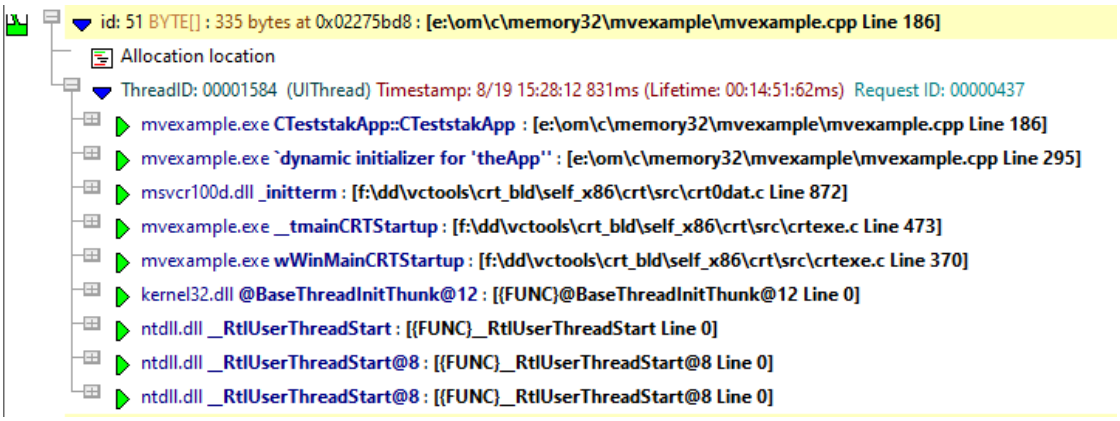
Examining a data item

Each item can be expanded with the button (and then collapsed with the) to show more detailed information:

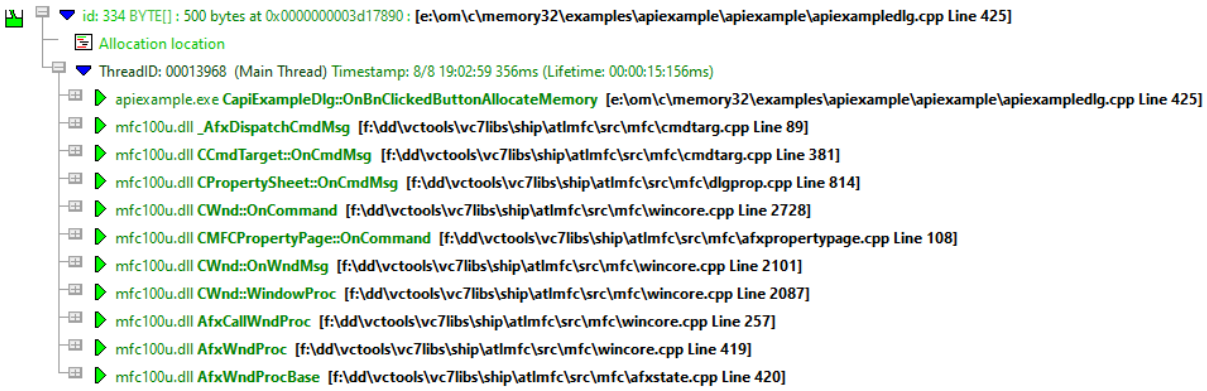
- brief description of the item's purpose (allocation location, reallocation location, etc)
- thread id, and the name if assigned
- timestamp
- lifetime of the item
- allocation request ID (if there is one)
- the callstack for the item

Each line of the callstack shows

- instruction address
- module name
- undecorated C++ function name
- source file and line number (if available) for the function



For allocations that Memory Validator has seen allocated with an identical callstack to a previous allocation that has been deallocated, that callstack and the top line entry are coloured green. This is a hint that allocations at this callstack location have been seen to be deallocated and that this location may not leak memory (although conditional logic may mean that it does).

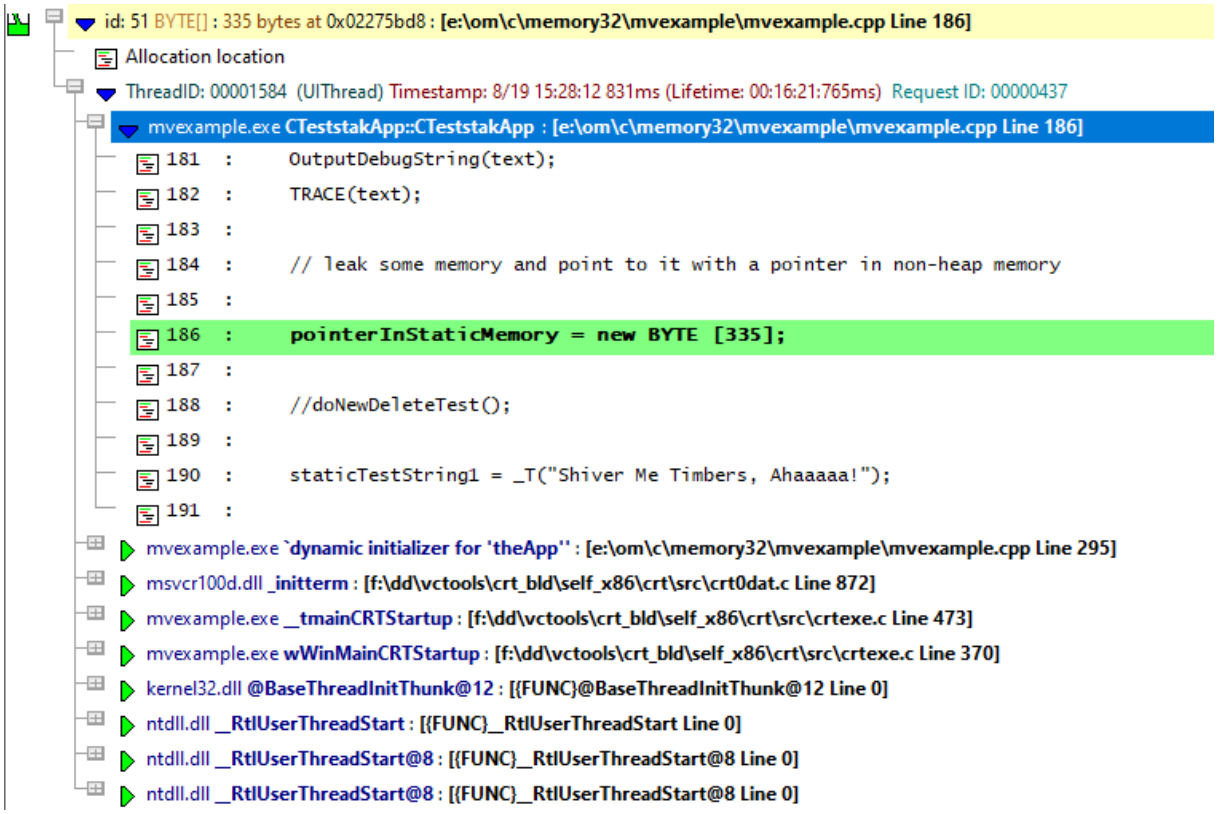


Examining the callstack and code

One or more parts of the callstack can be expanded or collapsed using the ⊞ or ⊟ to show the source code around the relevant line in the associated file.

If the source code can't be found, or the file location is invalid you'll be prompted for the file.

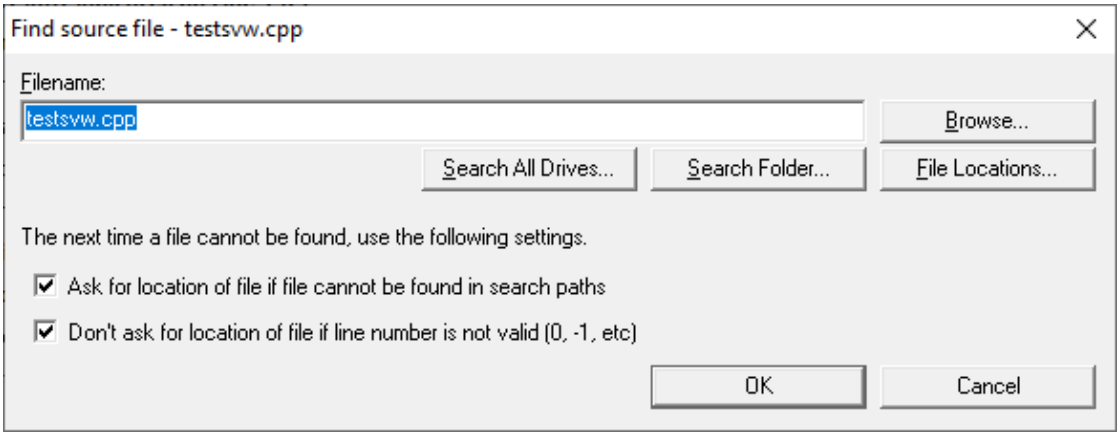
The line on which the allocation occurred is highlighted, e.g. green in this example:



To edit the source code, double click on any part of the lines of source code displayed or use **Edit Source Code...**

Source file not found automatically?

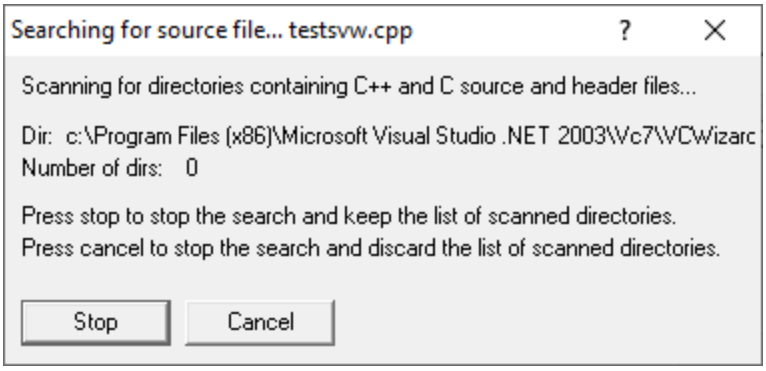
If the source file isn't found automatically, you'll be prompted to provide the location manually with the Find Source File dialog



You can scan, search or browse for the source location depending on how much of an idea you have of the location:

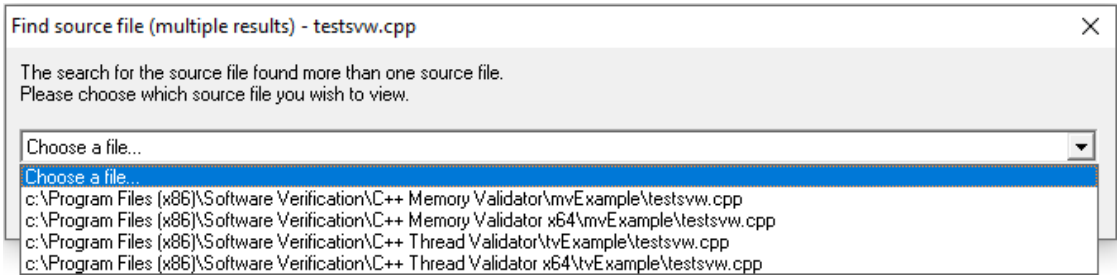
- **Browse...** › uses an explorer to search manually
- **Search All Drives...** › does a full scan of your computer, showing the Searching For Source Files dialog

  You can stop the search at any time



  If a file is found, the filename is entered at the top of the Find Source File dialog.

  If multiple results are found, pick the best one from the results dialog that appears:



- **Search Folder...** › prompts for a folder, and scans that using the same Searching For Source Files dialog as above

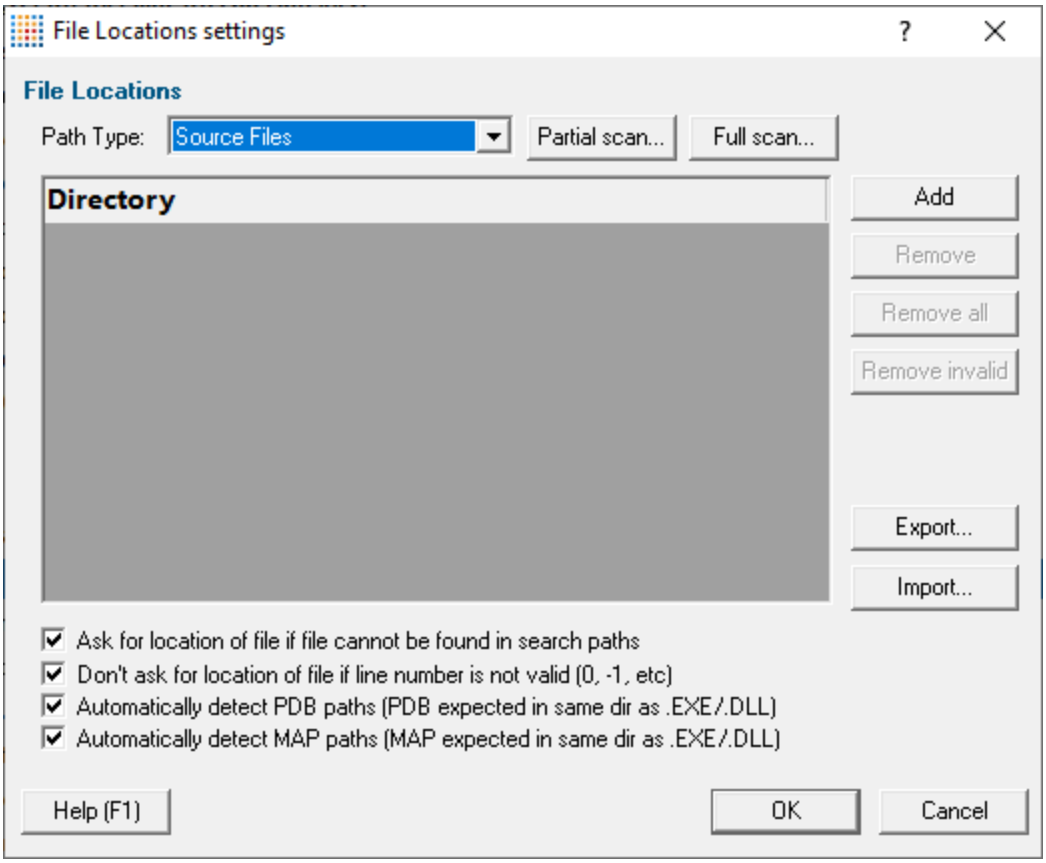
  If multiple results are found, pick the best one from the results dialog (above)

Rather than repeatedly searching manually for locations, it's recommended to modify the automatic source file search paths:

- **File Locations...** › shows the File Locations Settings dialog so you can change the automatic search paths

  Changing the search paths to include additional source locations means you'll get prompted less.

  The file locations settings dialog is identical to the File Locations page of the global settings dialog.



If you don't want to be prompted with this dialog, then uncheck the first option below

- **Ask for location of file if file cannot be found in search paths** › shows this dialog each time you try to open a source file where the location is unknown
- **Don't ask for location of file if line number is not valid** › stops this dialog from showing when line numbers are invalid, e.g. zero or negative

  The default is *not* to ask in this case.

Navigation shortcuts

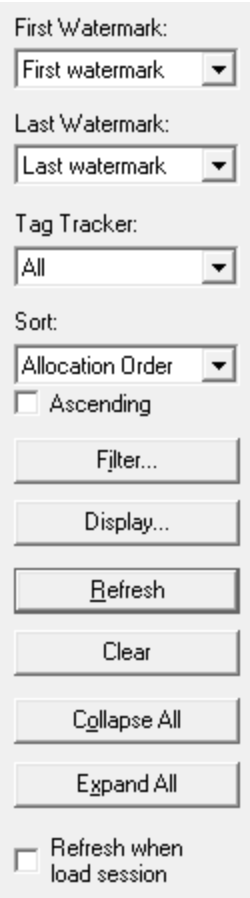
Two navigation keyboard shortcuts are provided. They collapse the current entry, move to the next or previous entry and then expand that entry to show the full callstack.

To move to the previous top level entry press Ctrl <

To move to the next top level entry press Ctrl >

Memory tab options

The following controls are displayed to the left of the data area



Watermarks

The amount of data in the main display can be reduced by filters and watermarks.

Here you can choose two watermarks allowing only the data between them to be displayed.

- **First Watermark** › Choose a watermark from the list › **Last Watermark** › Choose another watermark › **Refresh** › updates the data shown in the display

  There are two permanent default watermarks, called *First watermark* (before anything else) and *Last watermark* (after anything else).

  Attempting to choose a first watermark later than the last watermark, or vice-versa will result in the alternate watermark automatically updating.

Tag Tracker

Data tracking allows you to associate collected allocation data with an id or tag.

If using data tracking, you can then choose here which of your tags you want allocations to be displayed for.

- **Tag Tracker** › Choose a tag from the list › **Refresh** › updates the data shown in the display

  There are two permanent tags: *All* which filters nothing (the default), and *None* which shows everything that is untagged.

➡ This type of tag tracker and watermark selection is also used in the Tag Tracker and Watermark view of the Types and Sizes tabs, as well as the Hotspots and Analysis tabs.

Sorting

The data on the display can be ordered using the following attributes and in an ascending or descending direction - just pick an attribute and click **Refresh.**

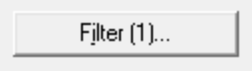
- **Allocation Order** › the order that events are recorded by Memory Validator - the event sequence id
- **Num Allocations** › the number of allocations at this unique callstack
- **Size** › the allocation size
- **Total Size** › the total number of bytes allocated at this unique callstack
- **Object Type** › the allocation object type (or handle type)
- **Tag Tracker** › the allocation tag tracker (above)
- **Filename** › the filename where the allocation occurred
- **DLL** › the name of the DLL where the allocation occurred
- **Address** › the allocation address or handle value

- **Ascending** › **Refresh** › when ticked shows the biggest, newest, etc. last, otherwise first if unticked

Local filters and settings

- **Filter...** › shows the local filters dialog for the memory tab

The filter button also indicates the *number* of local filters, although not all of these may be enabled



- **Display...** › shows the Memory Tab Display Settings dialog to set the types of data and messages displayed in the data view

Updating the display

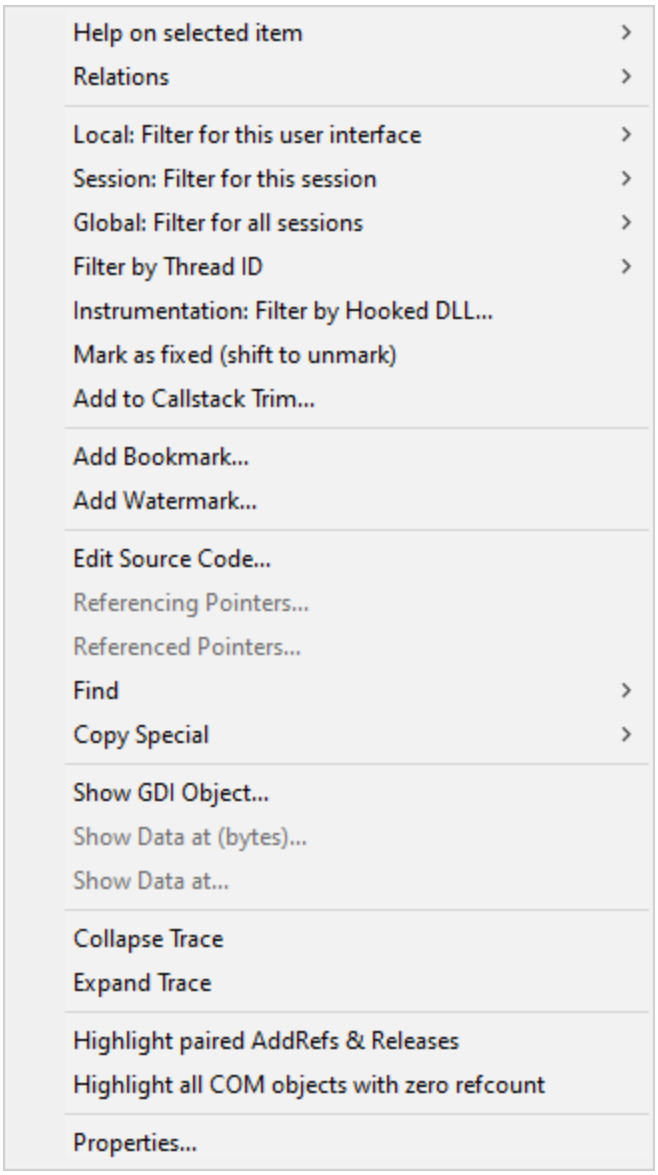
- **Refresh** › updates the display - as does the button on the Tools menu and toolbar
- **Clear** › removes all data from the display

- **Collapse All** › collapse all data items, whilst remembering any source code views that were open
- **Expand All** › expand all data items including any previously expanded sections of source code
- **Refresh when load session** › when ticked updates the display immediately when a session is loaded from a file

Memory view popup menu

The following popup menu is available over the data area

Click on any part of the menu to jump straight to the topic below:



Menu option: data item summary

- **Help on selected item** > the sub-menu shows a simple one line description of the type of data that has been selected:

Menu option: relations

The relations menu has a large sub-menu with many different options for choosing a set of related data to display in the upper analysis window.

Think of this as a sub-query on the working data - like searching for friends of friends on a social network!

Given an entry in the upper window, available relations are as follows, with *allocations* generally meaning any allocation, reallocation or deallocation

• **Same address** • **Same size** • **Smaller** • **Larger**	Finds any other allocations on the same memory address, for example previous allocations or frees Allocations on any memory objects of identical size or on smaller or larger objects
• **Same handle**	Finds any other allocations on the same resource handle
• **Same location, same callstack** **different callstack** **all callstacks**	Finds other allocations made at the source code location: via the same callstack different callstacks or any callstack
• **Same function** • **Same source file** • **Same DLL**	All allocations from the same function... or the same file or the same DLL
• **Class allocations**	All allocations, reallocations or deallocations from the same C++ class
• **Relations to 'this'**	Finds various other events relating to the selected object: **Allocator of this** - only for reallocated objects **Reallocation of this** **Reallocation of this address at same address** **Reallocation of this address at different address** **Deallocation of this** **Allocations, reallocations, deallocations** **Referenced by this** - these two need the target application to still be running **Referencing this**

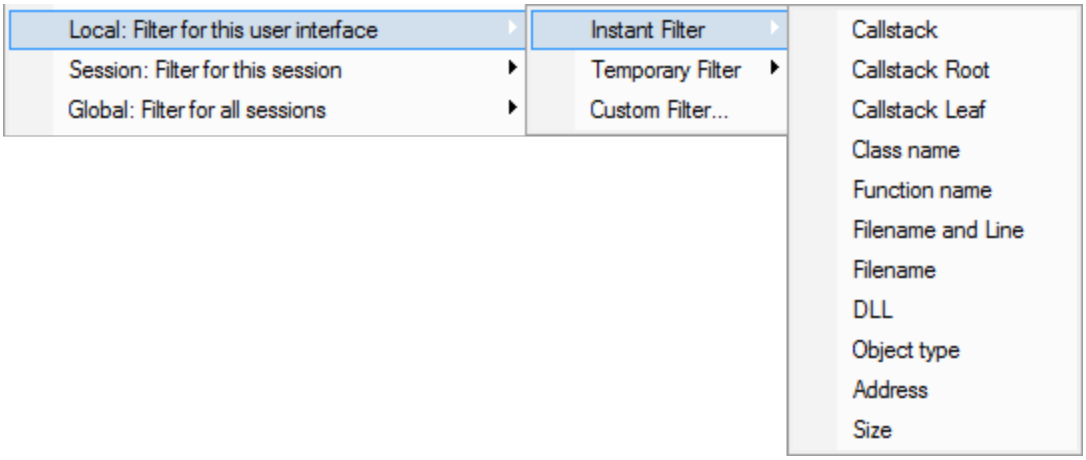
• **Same address** • **Same size** • **Smaller** • **Larger**	Finds any other allocations on the same memory address, for example previous allocations or frees Allocations on any memory objects of identical size or on smaller or larger objects
• **Allocations within**	For memory allocations, finds all other allocations within a range of 32 bytes up to 4Kb of this one
• **Allocations prior** • **Allocations after**	For memory allocations, finds the previous 5, 10 or 20 allocation events or the next 5, 10 or 20 events
• **Errors**	Shows any known damaged memory allocation information relating to this entry

Menu option: filters

The next three options provide a powerful way of filtering the data to exclude all other items similar in some way to the one you clicked on.

- **Local: Filter for this user interface**
- **Session: Filter for this session**
- **Global: Filter for all sessions**

Each of these options have the same sub-menus:



- **Instant Filter** › creates an instant filter based on the selected data item
- **Temporary Filter** › as for instant filter, but will not be saved along with any other filters

The final sub-menu above shows a variety of predefined filters, each of which creates a filter matching that attribute of the selected item.

The term *callstack root* refers to the function at the top of the callstack, while all other options refer to the lowest item in the callstack - the callstack *leaf*.

So for example:

- **Callstack** › creates a filter that excludes data items matching the entire callstack of the selected item
- **Callstack Root** or **Leaf** › excludes items matching the address for the top or bottom of the callstack

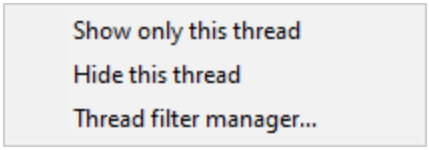
  Examples of matching callstack root and leaf are shown in the Filter Definition topic.
- **Object Type**, **Size** or **Address** › excludes all items with the same allocation type, size or address

- **Custom Filter** › brings up the Define Filter dialog, letting you customise your own filter in more detail

  The filter definition will be pre-populated with the callstack and other datatype information to use in your filter.

Menu option: filter by thread id

The Filter by Thread Id submenu has the following options:

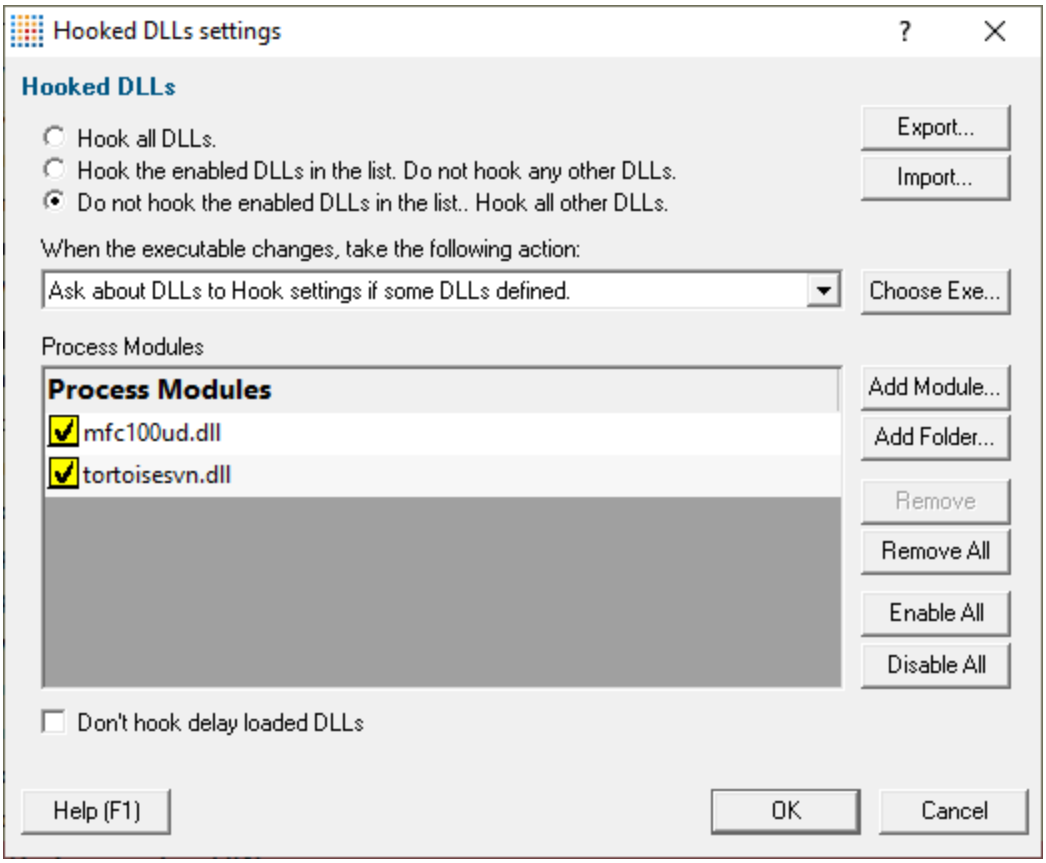


- **Show only this thread** › the thread filter is set to show only the selected thread
- **Hide this thread** › the thread filter is set to hide the selected thread
- **Thread filter manager...** › the thread filter manager is displayed

Menu option: filter by hooked DLL

You can filter the next run by DLL, either excluding that DLL, or including that DLL, in the list of DLLs that will be monitored.

The Hooked DLLs settings dialog is displayed.

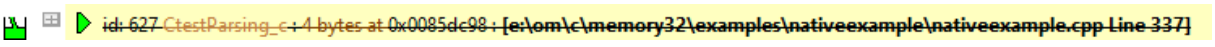


Menu option: mark as fixed

- **Mark as fixed** › marks the selected item as "I have fixed this"

  To remove the marking from the event press the shift key at the same time you choose **Mark as fixed** on the menu.

Items that have been marked as fixed are shown with a line struck through them.



This allows you to easily identify items that you've worked on and items that have yet to be worked on.

Menu option: add to callstack trim

- **Add to Callstack Trim...** › displays the Callstack Trim settings with the selected function added to the list of trimmed functions

Menu option: bookmarks and watermarks

Bookmarks allow you to find a data item easily at a later date, while watermarks are used above to show only those items between two points in time

- **Add Bookmark...** › adds a bookmark for the selected item
- **Add Watermark...** › adds a watermark for the selected item

Menu option: editing source code

- **Edit Source Code...** › opens the default or preferred editor to edit the source code

Menu option: pointers in and out

- **Referencing Pointers...** › shows all pointers in the target program that *point to* the selected item
- **Referenced Pointers...** › shows all pointers in the target program *identified by* the selected item

Referencing and referenced pointers are shown in the same references dialog as when finding addresses.

These options are only available while the target application is running

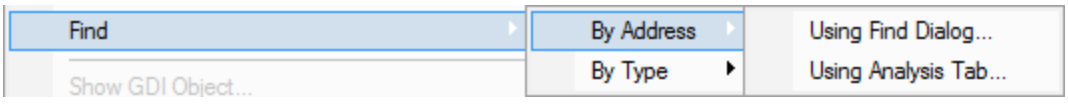
Menu option: find

The find option and sub-menus below allow you to use the address or the object type of the allocation to say things like:

> Show me all the data items that have the same allocation type as the selected item

or

> Add to the Analysis tab all the data items where the allocation address is the same as the selected item

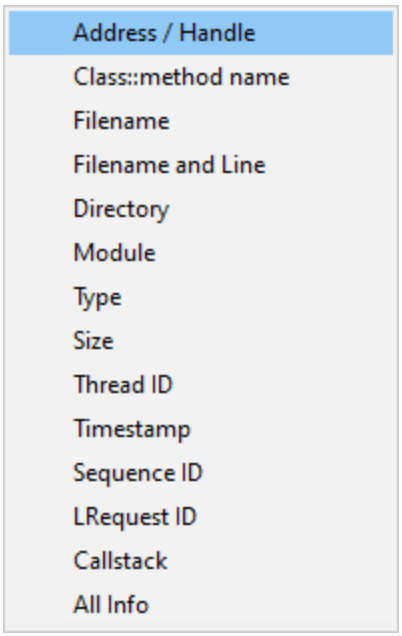


The results can be displayed in one of three places:

- the Address Query Dialog if you choose **By Address** › **Using Find Dialog...**
- the Object Query Dialog if you choose **By Type** › **Using Find Dialog...**
- the Analysis tab when choosing **Using Analysis Tab** for both Address and Type

Menu option: copy special

The copy special sub-menu lets you copy to the clipboard any of the following attributes, (or all the information):



This can be very useful if you need to use the address, file name or other attribute of an allocation elsewhere in Memory Validator; an external application, or just to share with a colleague.

Possible areas in Memory Validator that you might need such data include:

- memory search
- address query
- object type query
- filter definition
- leak detection

...and other similar tools.

Here's an example of copying **All Info** to the clipboard:

```
c:\program files (x86)\software verification\c++ memory validator\examples\nativeexample\te
nativeExample.exe
Thread ID: 284736
11/23 12:28:33 267ms (Lifetime:00:06:25:666ms)
Sequence: 854
nativeExample.exe  CTeststakView::OnTestDoublefreeofmemory : [c:\program files (x86)\softwa
mfc90ud.dll  _AfxDispatchCmdMsg : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\cmdtarg.cpp Li
mfc90ud.dll  CCmdTarget::OnCmdMsg : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\cmdtarg.cpp
mfc90ud.dll  CView::OnCmdMsg : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\viewcore.cpp Line
mfc90ud.dll  CFrameWnd::OnCmdMsg : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\winfrm.cpp Li
mfc90ud.dll  CWnd::OnCommand : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\wincore.cpp Line
mfc90ud.dll  CFrameWnd::OnCommand : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\winfrm.cpp L
mfc90ud.dll  CWnd::OnWndMsg : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\wincore.cpp Line 1
mfc90ud.dll  CWnd::WindowProc : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\wincore.cpp Line
mfc90ud.dll  AfxCallWndProc : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\wincore.cpp Line 2
mfc90ud.dll  AfxWndProc : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\wincore.cpp Line 402]
mfc90ud.dll  AfxWndProcBase : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\afxstate.cpp Line
USER32.dll  gapfnScSendMessage : [{FUNC}gapfnScSendMessage Line 0]
USER32.dll  GetThreadDesktop : [{FUNC}GetThreadDesktop Line 0]
USER32.dll  CharPrevW : [{FUNC}CharPrevW Line 0]
USER32.dll  DispatchMessageW : [{FUNC}DispatchMessageW Line 0]
mfc90ud.dll  AfxInternalPumpMessage : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\thrdcore.c
mfc90ud.dll  CWinThread::PumpMessage : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\thrdcore.
mfc90ud.dll  CWinThread::Run : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\thrdcore.cpp Line
mfc90ud.dll  CWinApp::Run : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\appcore.cpp Line 864
mfc90ud.dll  AfxWinMain : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\winmain.cpp Line 47]
nativeExample.exe  wWinMain : [f:\dd\vctools\vc7libs\ship\atlmfc\src\mfc\appmodul.cpp Line
nativeExample.exe  __tmainCRTStartup : [f:\dd\vctools\crt_bld\self_x86\crt\src\crtexe.c Lin
nativeExample.exe  wWinMainCRTStartup : [f:\dd\vctools\crt_bld\self_x86\crt\src\crtexe.c Li
```

The same menu option is also on the Analysis menu

Menu option: showing GDI objects

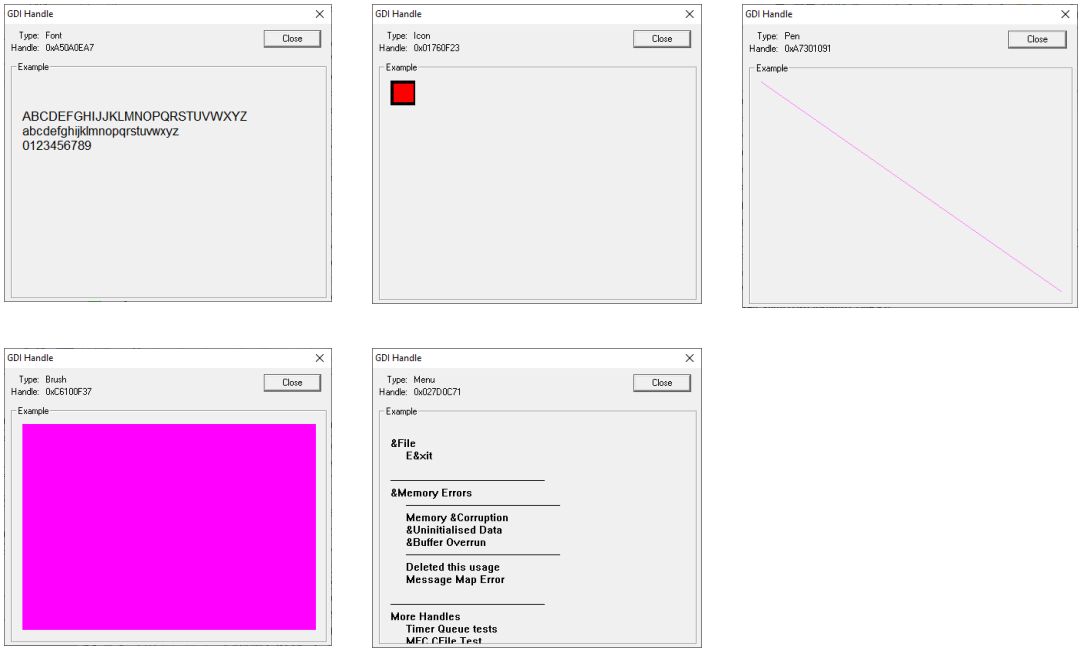
- **Show GDI Object...** › shows a graphical representation of the GDI object that is selected

A GDI Handle dialog will be displayed showing the handle type, the value and appearance.

This option is only valid for the GDI objects listed below, and *only while your application is running* as the data will not be accessible afterwards.

- Bitmap
- Brush
- Cursor
- Font
- Icon
- ImageList
- Palette
- Pen
- Region
- Menu
- Window

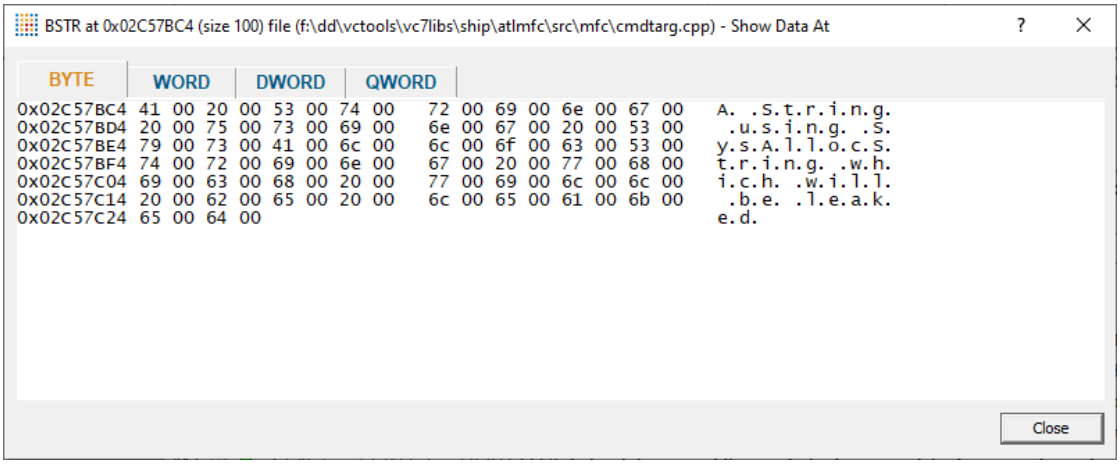
These examples below are from the example application and show a font, icon, pen, brush and menu:



Menu option: show memory byte data...

- **Show Data at (bytes)...** › shows the byte format data at the memory allocation identified by the selected item

The example below shows the memory for a BSTR string shows the address of each 16 byte row, 16 bytes of data and the ASCII text conversion on the right:



The Options menu for this window lets you update the content and choose whether to colourise data (e.g. uninitialized memory), or to display the memory content as bytes, words or dwords.

- **Refresh** › redisplay updated data
- **Colour** › highlight bytes:
  uninitialized data signatures (0xCD)
  C runtime heap data guards (0xFD)
  deleted memory (0xDD)
  Win32 heap values (uninitialized: 0xBAADF00D, deleted: 0xDEADBEEF)
- **Bytes** › displays data as a series of bytes with ascii text representation
- **WORDS** › displays data as WORDs with unicode text representation
- **DWORDS** › displays the data as a series of DWORDs (no text equivalent)
- **Exit** › closes the data window

Showing byte data is only possible whilst the target application is still running

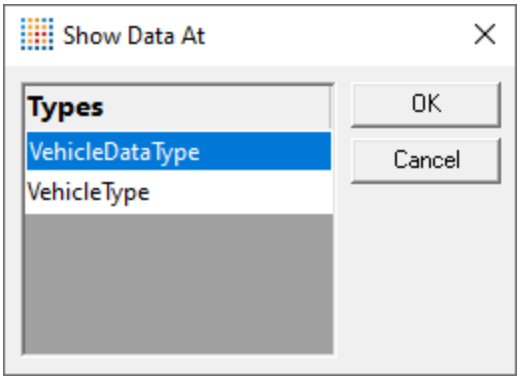
Menu option: show data at...

This option is dependent upon having first defined some datatypes or enumerations in the global settings dialog.

These example allocation below uses the data structures from the section on defining a new datatype.

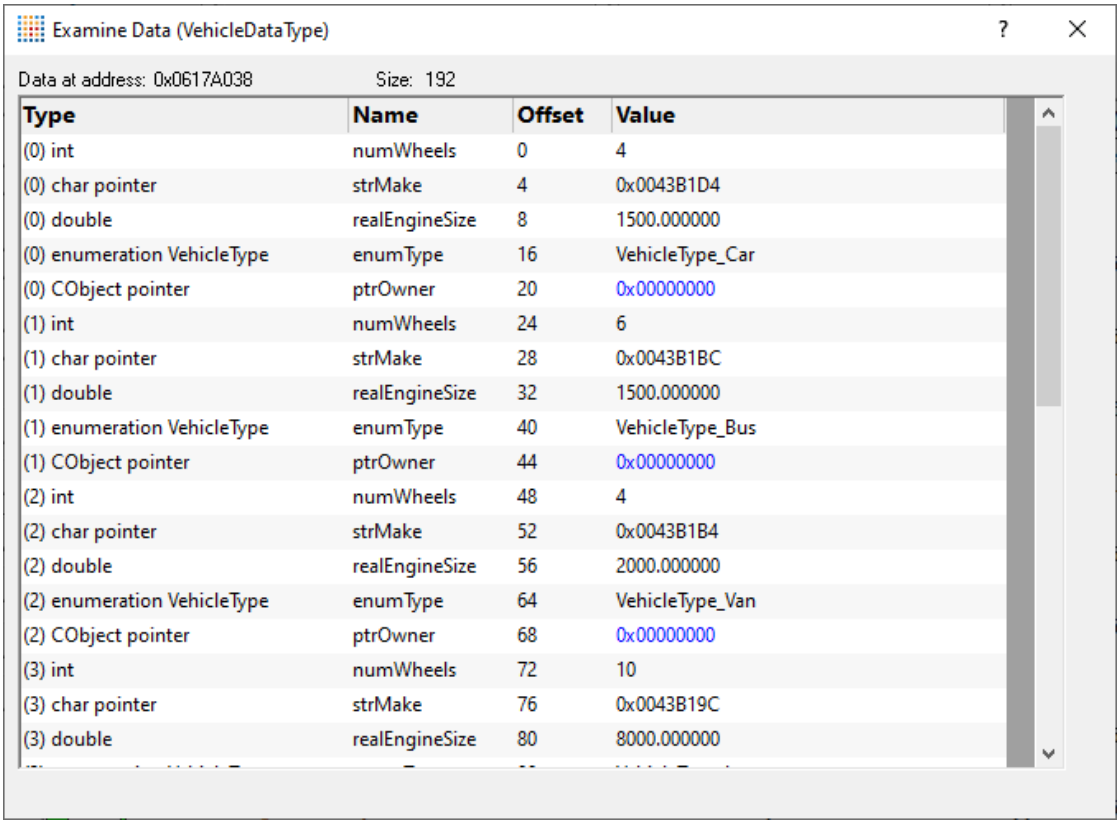
```
775 :  VehicleDataType* vehicles = new VehicleDataType[4];
776 :
777 :  vehicles[0].enumType = VehicleType_Car;
778 :  vehicles[1].enumType = VehicleType_Bus;
779 :  vehicles[2].enumType = VehicleType_Van;
780 :  vehicles[3].enumType = VehicleType_Lorry;
```

- **Show Data at...** › shows the Show Data At dialog below



Choose a datatype to use for interpreting the memory contents at the allocation address and click **OK**.

The allocated memory will be displayed showing values in memory for each data member:



Type	Name	Offset	Value
(0) int	numWheels	0	4
(0) char pointer	strMake	4	0x0043B1D4
(0) double	realEngineSize	8	1500.000000
(0) enumeration VehicleType	enumType	16	VehicleType_Car
(0) CObject pointer	ptrOwner	20	0x00000000
(1) int	numWheels	24	6
(1) char pointer	strMake	28	0x0043B1BC
(1) double	realEngineSize	32	1500.000000
(1) enumeration VehicleType	enumType	40	VehicleType_Bus
(1) CObject pointer	ptrOwner	44	0x00000000
(2) int	numWheels	48	4
(2) char pointer	strMake	52	0x0043B1B4
(2) double	realEngineSize	56	2000.000000
(2) enumeration VehicleType	enumType	64	VehicleType_Van
(2) CObject pointer	ptrOwner	68	0x00000000
(3) int	numWheels	72	10
(3) char pointer	strMake	76	0x0043B19C
(3) double	realEngineSize	80	8000.000000

Some values are colour coded to highlight them. For example the NULL pointers in the above image.

Values that are colour coded are values related to NULL, uninitialised memory, deleted memory and damaged memory (buffer overruns and underruns).

Menu options: collapse / expand trace

- **Collapse** or **Expand Trace** › simply shows and hides data item information, the same as using the ⊞ or ⊟ buttons

Menu options: AddRefs & Releases, showing related COM objects

When tracking leaked COM objects it is useful if you can quickly and easily identify operations on the same COM objects.

- **Highlight paired AddRefs & Releases** › highlights *all* paired AddRef and Release calls

In general, clicking on any AddRef or Release item will display items relating to the same object in the selected object colour.

Menu options: Zero refcount COM objects

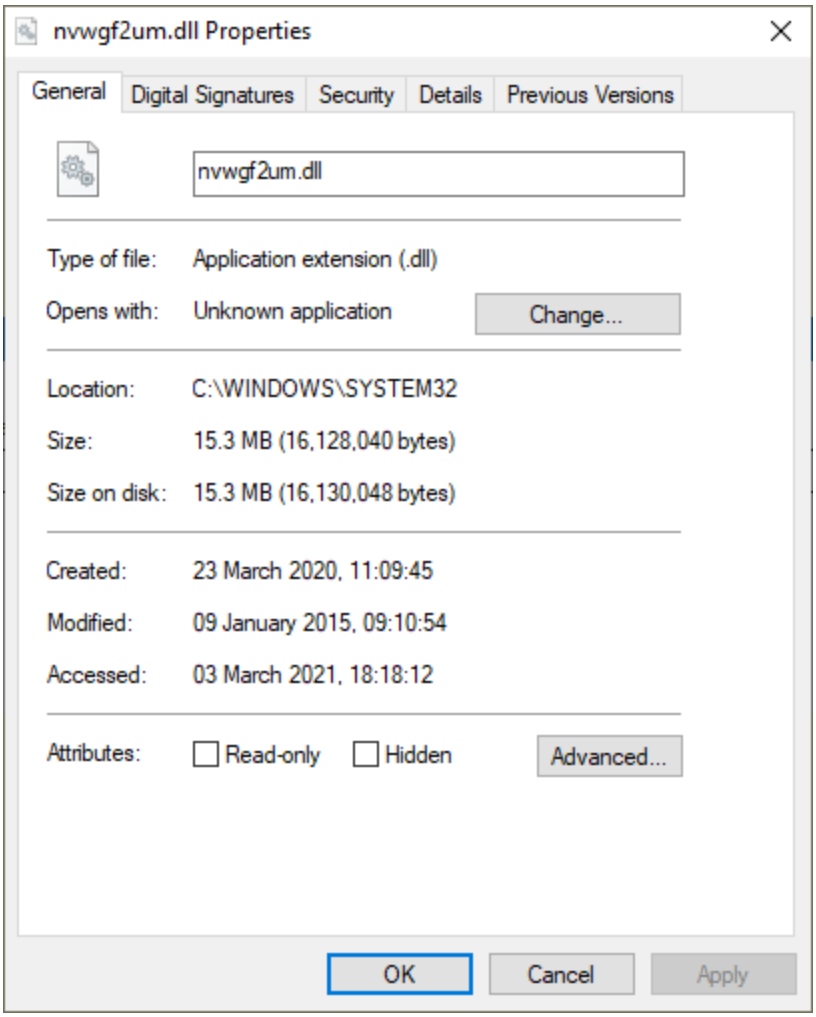
- **Highlight all COM objects with zero refcount** › highlights items in the display that are zero refcount COM objects

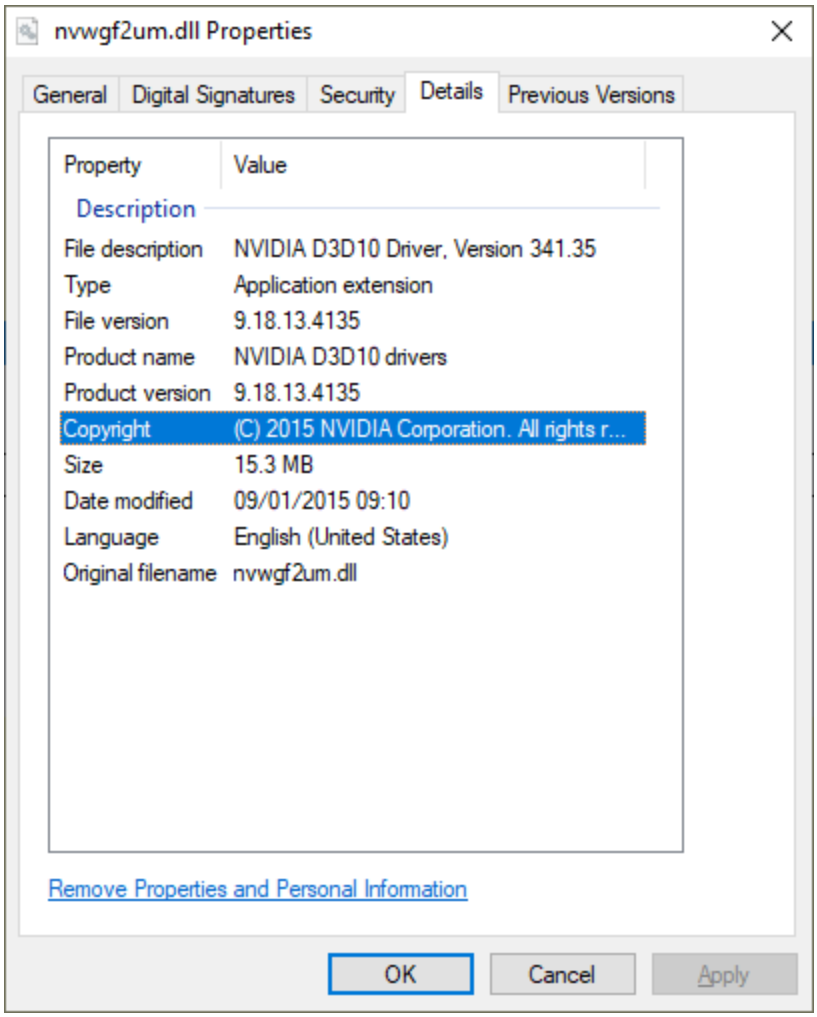
When you know which objects have a reference count of zero, you know that they have been deallocated.

This means you can concentrate on any COM objects that have *not* been highlighted, since reference count is non zero.

Menu options: Properties

- **Properties...** › display the Windows File Property dialog for the DLL.

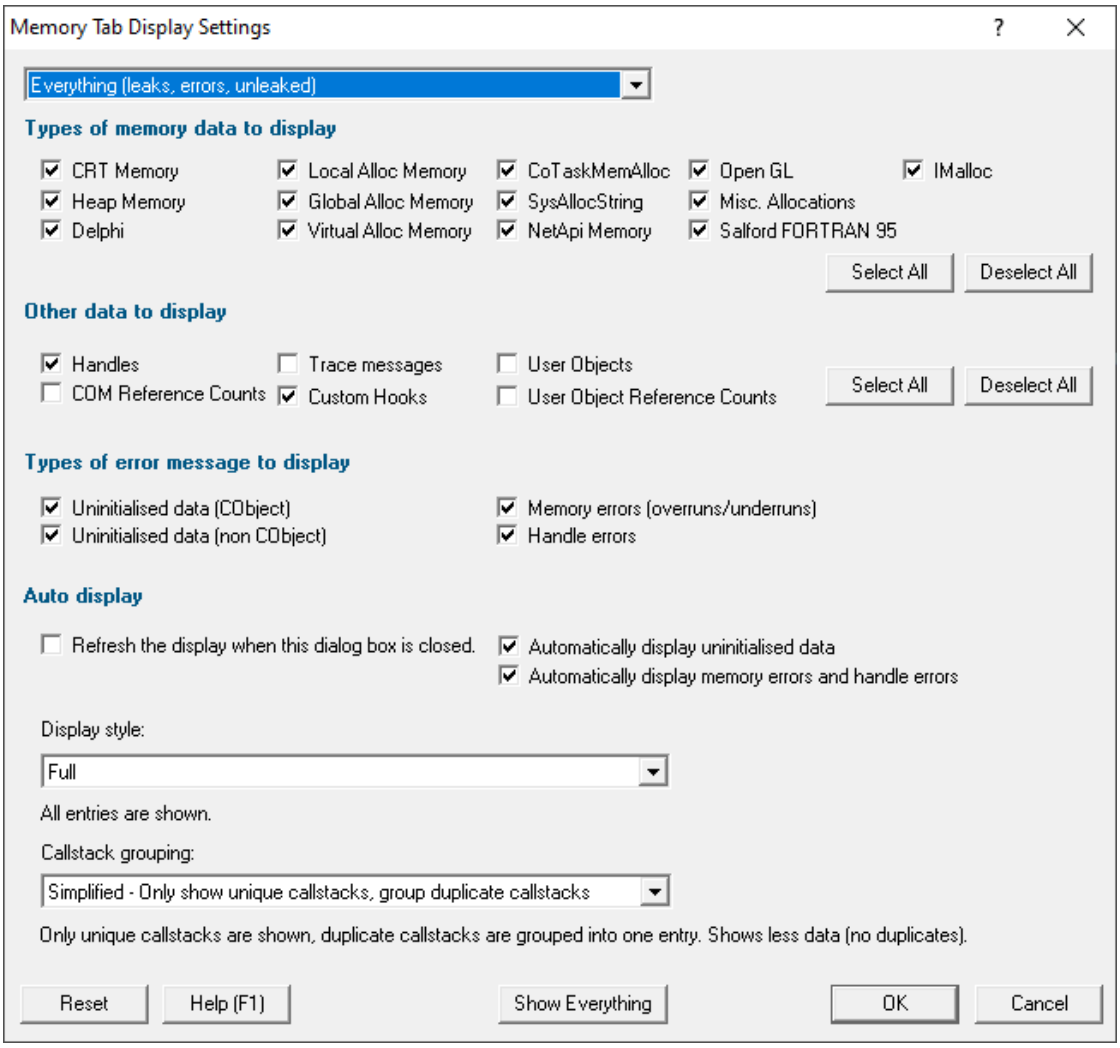




3.7.2.1.1 Memory Display Settings

The Memory Tab Display Settings control the memory, handles and messages displayed on the Memory tab.

The default options are shown below:



Note that these settings control what is *displayed*, not what is *collected*. The data collection settings may have more information about some of the settings below.

Display categories

The combo box at the top of the dialog controls the broad categories of what is displayed, the other sections of the settings dialog provide more find grained control within the broad categories.

Choosing a display category can be a really useful way to quickly simplify the display to just a few items. For example just showing the errors, so that you don't have to wade through 1000s of leaked data reports to find the more serious errors.

- **Everything** › all data is displayed
- **Leaks and errors** › items that are leaked (or potentially leaked) and errors are displayed
- **Leaks** › items that are leaked (or potentially leaked) are displayed
- **Errors** › items that are errors are displayed
- **Unleaked** › items that are not leaked and not errors are displayed

Note that even though you've chosen a broad category (for example Everything) the other criteria on this dialog also have to be satisfied in order for the item to be displayed.

Types of memory to display

Setting	Displays memory that is not deallocated and was allocated by...
• CRT memory	CRT memory
• Heap	the HeapAlloc() group of functions
• Delphi	Delphi's allocation functions
• Local Alloc	the LocalAlloc() group of functions
• Global Alloc	the GlobalAlloc() group of functions
• Virtual Alloc	the VirtualAlloc() group of functions (see below)
• CoTaskMemAlloc	the CoTaskMemAlloc() group of functions
• SysAllocString	the SysAllocString() group of functions
• NetApi	the NetApi group of functions
• Open GL	the OpenGL group of functions
• Misc	various other memory functions that allocate and manage memory
• Salford FORTRAN 95	Salford Software's FORTRAN 95 allocation functions
• IMalloc	the IMalloc() interface

Using VirtualAlloc?

Note that your call to VirtualAlloc/VirtualAllocEx should include the flag MEM_RESERVE or MEM_COMMIT.

If including MEM_COMMIT without MEM_RESERVE the first page of the proposed address range should be non-reserved or your proposed address should be NULL allowing the operating system to choose the address for you.

Other data to display

Setting	Displays...
• Handles	handles that have not been deallocated
• COM Reference Counts	COM Objects and reference counts (Query Interface, AddRef, Release) that Memory Validator is aware of
• Trace Messages	TRACE() and OutputDebugString() messages received by Memory Validator
• Custom Hooks	memory allocated by custom memory allocation functions
• User Objects	memory allocated by the user defined allocation functions which has not been deallocated

• User Object Reference Counts	reference counts received from the user defined reference count function

Error messages

Setting	Displays...
• Uninitialized Data (CObject)	memory marked as uninitialized and derived from CObject
• Uninitialized Data (non CObject)	memory marked as uninitialized and not derived from CObject
• Memory Errors (overruns/underruns)	memory marked as damaged due to an overrun or underrun You may also need to set the **All Memory (leaks, errors, unleaked)** check box to see changes as memory error traces are not considered memory leaks.
• Handle Errors	attempts to use handles incorrectly (passing them to the wrong functions)

Auto

Setting	Displays...
• Automatically Display UnInit Data	uninitialized data information received by Memory Validator
• Automatically Display Memory Errors	memory damage data information received by Memory Validator

Other settings

Setting	Displays...
• Display style	how much information is displayed on the screen
• Callstack grouping	display all callstacks or just unique callstacks
• Refresh the display when this dialog box is closed	updates when this settings dialog is closed

The display style can be one of the following values:

- **Full** › information about every allocation and error is displayed (unless filtered)
- **Simplified - your source code at root** › Only traces that have a callstack with your source code at the top of the callstack are displayed
- **Simplified - your source code not at root** › Only traces that have a callstack with your source code in the callstack (except for the top position) are displayed
- **Simplified - your source code anywhere** › Only traces that have a callstack with your source code anywhere in the callstack are displayed
- **Simplified - compiler vendor source code at root** › Only traces that have a callstack with your compiler vendor source code at the top of the callstack are displayed

- **Simplified - compiler vendor source code not at root** › Only traces that have a callstack with your compiler vendor source code in the callstack (except for the top position) are displayed
- **Simplified - compiler vendor source code anywhere** › Only traces that have a callstack with your compiler vendor source code anywhere in the callstack are displayed
- **Simplified - no source code** › Only traces that have a callstack with no source code are displayed

The callstack grouping can be one of the following values:

- **Full** › every callstack is shown
- **Simplified - Only show unique callstacks** › Traces that share the same callstack are displayed once. A summary is shown indicating the number of allocations, how many bytes in those allocations and the size of the largest allocation.

Reset

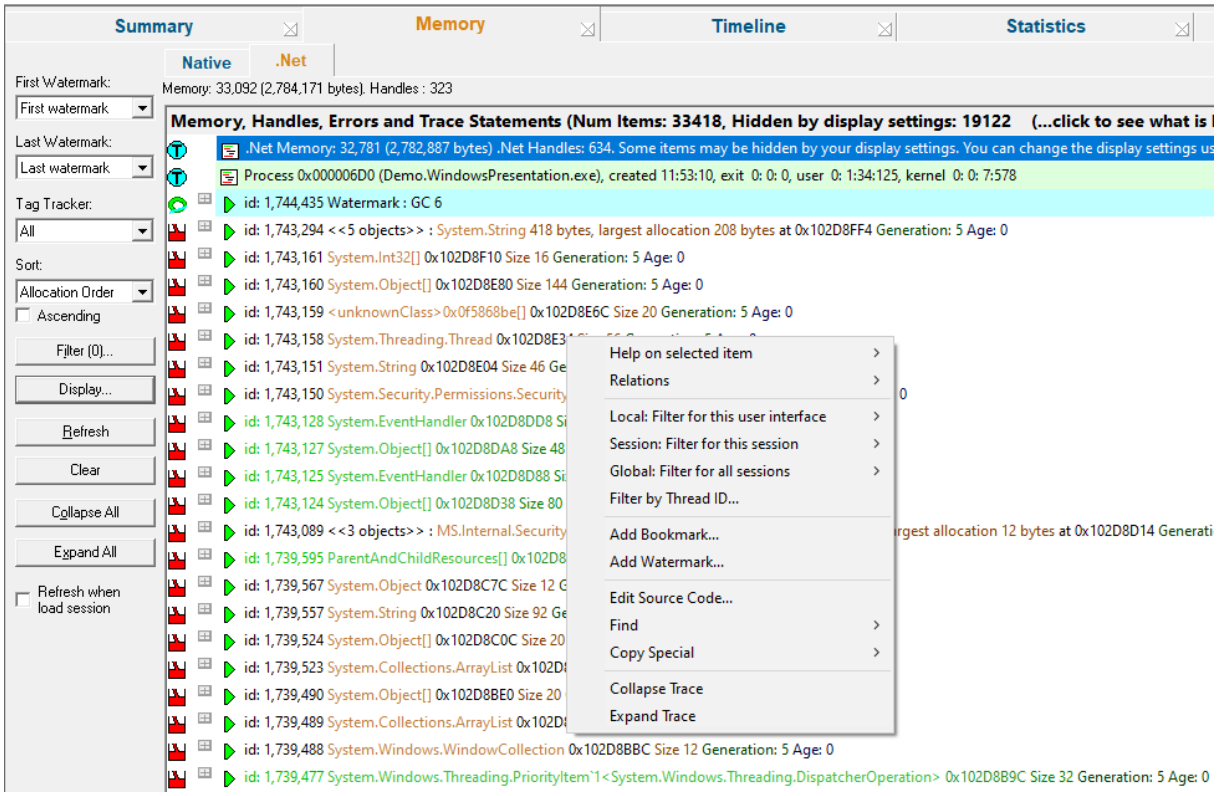
- **Reset** › resets all the display related settings for this tab

3.7.2.2 .Net Memory

The **.Net Memory** tab displays .Net memory and .Net handle allocations that are still waiting to be garbage collected.

This display is very similar to the Native Memory tab, but it only displays .Net memory allocations and the context menu is a subset of the context menu on the Native Memory tab because some of the options don't apply to .Net memory/handle allocations.

 Read on, or click a part of the image below to jump straight to the help for that area (the icons link to a different page).



Status Information

The very top line of the memory tab gives a brief summary of:

- memory allocated and number of blocks
- handles allocated

The status line is only updated when the Refresh button is pressed or when data is added to the display automatically.

More up-to-date information can be found on the Running Totals Dialog or the Types view.

This example shows a status line after the target application has exited (hence 0 bytes for *current* memory):

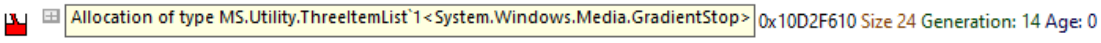
Memory: 2,024 (166,138 bytes). Handles : 47

Collected data

The main display shows the *live* collected data during or after the application running. Memory that has been allocated and then garbage collected is not shown.

Initially no data will be displayed until the data is manually refreshed or the automatically updated with significant errors as they happen.

Each line has an icon at the left, indicating its type, and has an explanatory tooltip:

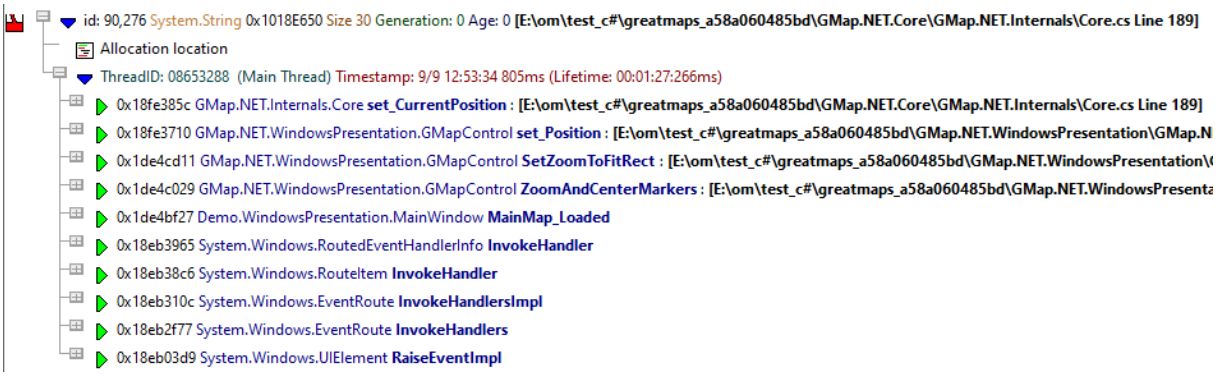


The text on each line indicates:

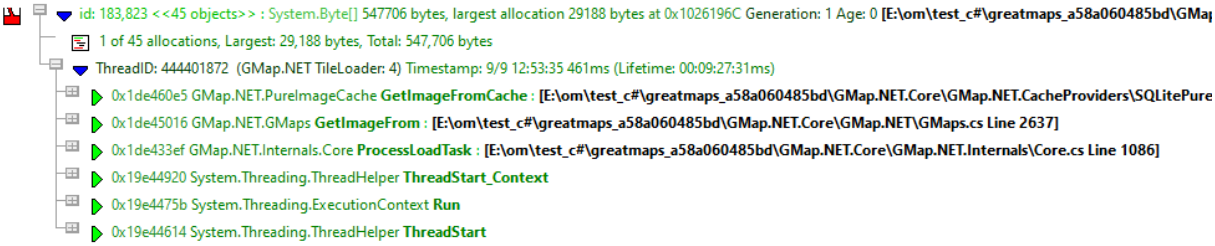
- datatype (if known)
- size
- allocation address/handle value
- source file and line number (if available) where the allocation occurred
- optional event sequence id at the beginning of the line

There are two types of callstack representation.

The first callstack representation is that of an object that has been allocated on a callstack that has never had any objects allocated on it that have been garbage collected.



The second callstack representation is that of an object that has been allocated on a callstack that has had objects allocated on it garbage collected.



This colouring is controlled by the **Enhanced callstack colouring** option on the Callstack settings of the settings dialog.

Controls

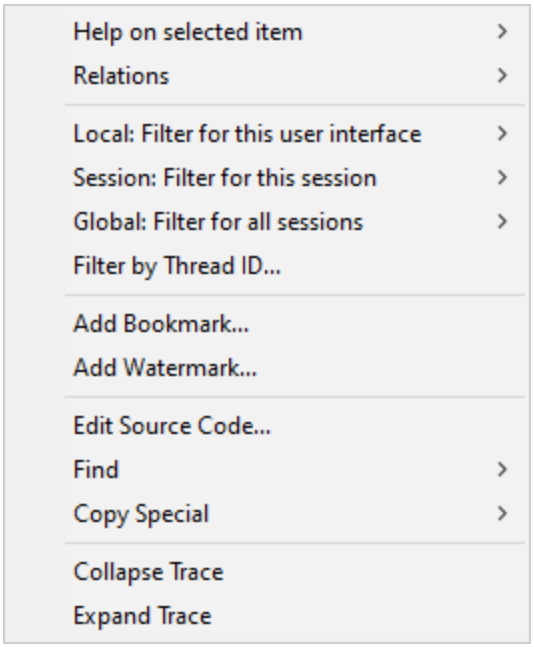
Most of the controls on the .Net Memory tab work identically to the controls on the Native Memory tab. The filters are shared between the two tabs.
One thing works differently is the Display Settings.

- **Display...** › shows the .Net Memory Tab Display Settings dialog to set the types of data and messages displayed in the data view

Memory view popup menu

The following popup menu is available over the data area. This menu is a subset of the menu on the Native Memory tab.
The descriptions for this menu are in the Native Memory tab.

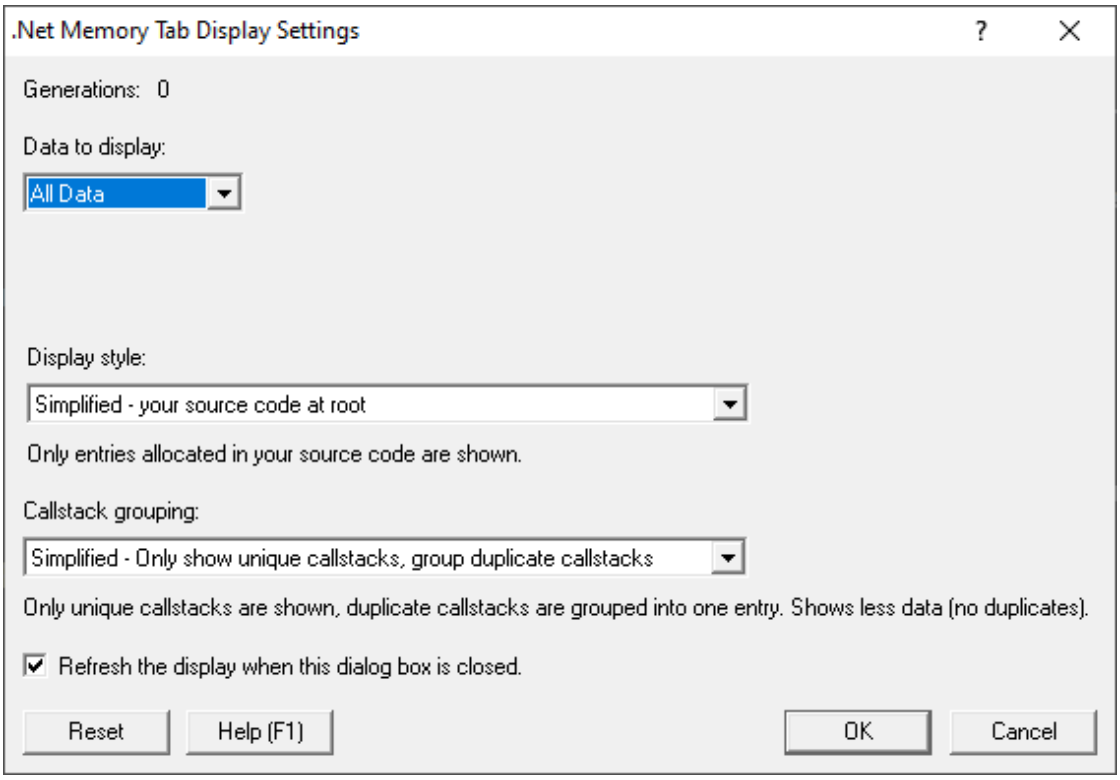
Click on any part of the menu to jump straight to the topic below:



3.7.2.2.1 .Net Memory Display Settings

The .Net Memory Tab Display Settings control the .Net memory, .Net handles and displayed on the .Net Memory tab.

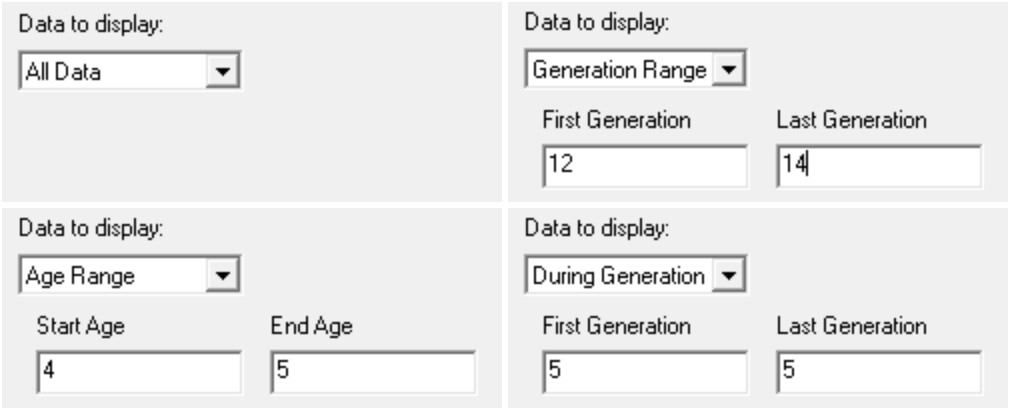
The default options are shown below:



Note that these settings control what is *displayed*, not what is *collected*. The .Net data collection settings may have more information about some of the settings below.

The Data to display combo causes the dialog to update as it is changed. The combo has four values

- **All Data** › display all information (unless filtered)
- **Generation Range** › displays information about every allocation between two generations (unless filtered)
- **Age Range** › displays information about every allocation between in the specified age range (unless filtered)
- **During Generation** › displays information about every allocation that was alive during generation (unless filtered)



Other settings

Setting	Displays...
• Group by Callstack	display all callstacks or just unique callstacks
• Refresh the display when this dialog box is closed	updates when this settings dialog is closed

The display style can be one of the following values:

- **Full** › information about every allocation and error is displayed (unless filtered)
- **Simplified - your source code at root** › Only traces that have a callstack with your source code at the top of the callstack are displayed
- **Simplified - your source code not at root** › Only traces that have a callstack with your source code in the callstack (except for the top position) are displayed
- **Simplified - your source code anywhere** › Only traces that have a callstack with your source code anywhere in the callstack are displayed
- **Simplified - compiler vendor source code at root** › Only traces that have a callstack with your compiler vendor source code at the top of the callstack are displayed
- **Simplified - compiler vendor source code not at root** › Only traces that have a callstack with your compiler vendor source code in the callstack (except for the top position) are displayed
- **Simplified - compiler vendor source code anywhere** › Only traces that have a callstack with your compiler vendor source code anywhere in the callstack are displayed
- **Simplified - no source code** › Only traces that have a callstack with no source code are displayed

The callstack grouping can be one of the following values:

- **Full** › every callstack is shown
- **Simplified - Only show unique callstacks** › Traces that share the same callstack are displayed once. A summary is shown indicating the number of allocations, how many bytes in those allocations and the size of the largest allocation.

Reset

- **Reset** › resets all the display related settings for this tab

3.7.3 Timeline

The **Timeline** tab gives you a set of graphical timelines showing all memory, objects and gdi handles, user32 handles and other handles tracked by Memory Validator.


Click a part of the image below to jump straight to the help for that area.

Timeline display styles

At the top of the Timeline tab is a combo box containing a list of many display styles, allowing you to choose what data to watch.

- **Adaptive** › chooses a display style determined by the type of data collected from the target application. The resulting displays will be equivalent to **All Native T/C**, **All .Net T/C** and **All Native and .Net T/C**.
- **Native Memory** › presents total native memory as well as unfreed memory, allocations and deallocations
- **Native Handles** › shows similar data for native handles
- **Native GDI Handles** › shows similar data for native GDI handles
- **Native USER32 Handles** › shows similar data for native USER32 handles
- **Native Objects** › tracks the *number* of native allocations and deallocations
- **.Net Memory** › presents total .Net memory as well as unfreed memory, allocations and garbage collected memory
- **.Net Handles** › shows similar data for .Net handles
- **.Net Objects** › tracks the *number* of .Net allocations and garbage collected memory
- **Native and .Net Memory** › presents total native and .net memory as well as unfreed memory, allocations and deallocations
- **Native and .Net Handles** › shows similar data for native and .net handles

- **Native and .Net Objects** › tracks the *number* of native and .net allocations and deallocations
- **Native Allocations T/C** › Shows all native allocation data with topics horizontally and categories vertically
- **Native Allocations C/T** › Shows all native allocation data with topics vertically and categories horizontally
- **Native Allocations T/C** › Shows all .Net allocation data with topics horizontally and categories vertically
- **Native Allocations C/T** › Shows all .Net allocation data with topics vertically and categories horizontally
- **Native Allocations and Deallocations T/C** › Shows all native allocation and deallocation data allocation with topics horizontally and categories vertically
- **Native Allocations and Deallocations C/T** › Shows all native allocation and deallocation data allocation with topics vertically and categories horizontally
- **Native Allocations and Deallocations T/C** › Shows all .Net allocation and deallocation data allocation with topics horizontally and categories vertically
- **Native Allocations and Deallocations C/T** › Shows all .Net allocation and deallocation data allocation with topics vertically and categories horizontally
- **All Native T/C** › Shows all Native data with topics vertically and categories horizontally
- **All Native T/C** › Shows all Native data with topics horizontally and categories vertically
- **All .Net T/C** › Shows all Native data with topics vertically and categories horizontally
- **All .Net T/C** › Shows all Native data with topics horizontally and categories vertically
- **All Native and .Net T/C** › Shows all Native data with topics vertically and categories horizontally
- **All Native and .Net T/C** › Shows all Native data with topics horizontally and categories vertically

The difference between T/C and C/T displays is that the data shown in rows is shown in columns and vice versa. Depending on the data you're interested in you may find having the data on the same row is more beneficial, or having the data in the same column in different rows is more beneficial. We provide with you both options so that you have more freedom interpreting the data.

Why T/C and C/T? We wanted two letters that didn't look similar. We started out with X/Y and Y/X but after a while realised they are too similar.

C and T are impossible to mis-read and have the benefit of mapping to topics and categories which is helpful when thinking about the graphs rather than an abstract notion like X and Y.

Sampling periods - the key to understanding the graphs

Before covering the features, it's important to understand what these graphs are showing.

Once a second, Memory Validator takes a snapshot of the activity in the target program and plots another line on the graph to indicate how that activity affected resources.

The top graph shows snapshots of total *consumption* at each sample, while the others show *change* during each sample period.

In any given second, memory may be allocated, but not freed - although it may be freed later. This shows as a spike on the graph because there was an overall change during the sample period.

Memory allocated and freed within the *same* sample period does not contribute to a spike since there's no overall change.

The maximum value on any graph is the maximum change resulting from activity during any single sample period

Timeline graphs

Each memory or handle group consists of five graphs:

- **Total** › displays the total consumption, either as an amount of memory, number of handles, or number of allocations
- **Unfreed** › the amount of allocated data, not freed in the sample period.

  To calculate this add up all the allocations in the time period, and add up all the deallocations in the time period. Now subtract the deallocations from the allocations. If the result is negative, it is limited at 0.
- **Allocations (in order)** › shows allocation change during the sample
- **Deallocations (in order)** › shows deallocation changes during the sample period
- **Deallocations (in allocation order)** › timeline for deallocations in the order the related allocation happened

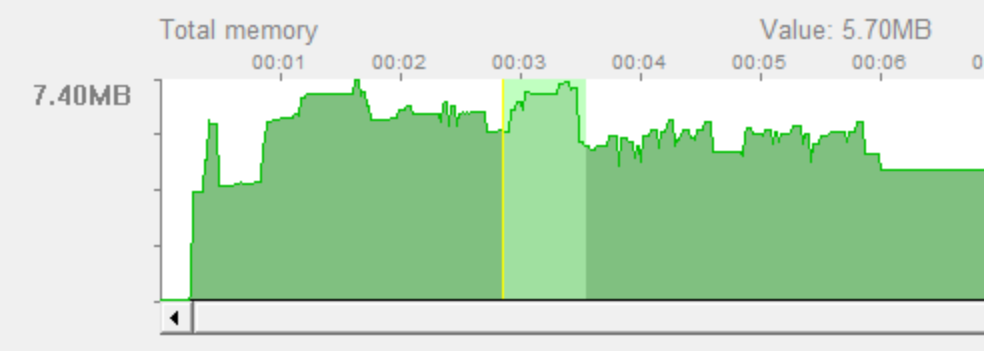
Each of these graphs is available for memory, handles and objects (count of all activities) for both native and .Net. Five data sources for six different counts. That makes for total of up to 30 different graphs that can be displayed depending on the choice you make with the combo at the top of the display. Which graphs you choose to use will depend on the problem you're looking at and what you find interesting.

The T/C and C/T variations of the graph allow you to examine things from different points of view. Do you want to see deallocations lined up below allocations (so that you can easily see deallocation vs allocation behaviour), or is it more interesting to see native memory and .Net memory lined up underneath each other so that you can see if the allocations in native memory correspond to the allocations in the .Net memory space?

Moving the mouse over a graph shows a yellow dotted marker indicating the current sample period, and some information above the graph.

- **Value** › displays memory usage at the sample point, or the number of allocations or handles depending on the graph

- **Sequence IDs** › the lowest and highest event sequence ids used in that sample period



Why is the maximum for deallocations sometimes different to the maximum for allocationMultiple related allocations and deallocations may interleave with each other. Because of the one second sampling they may also span multiple sample periods. This means that the total amount of freed memory in one period may not match that for total allocated in another period.

Why two deallocation graphs?

Comparing the two graphs and the allocation graph lets you see if deallocations are generally happening near (and in the same order) as the allocations or if they are happening some time after the allocation

You may also be able to see if the allocations are being deallocated in a different order to their allocation.

This can be useful for analysing memory usage behaviour and performance.

Timeline view popup menu

The following popup menu is available over the top three timelines to examine data in more detail, and is identical to the one over the Sizes tab.

Menu option: Showing locations - drilling down into the data

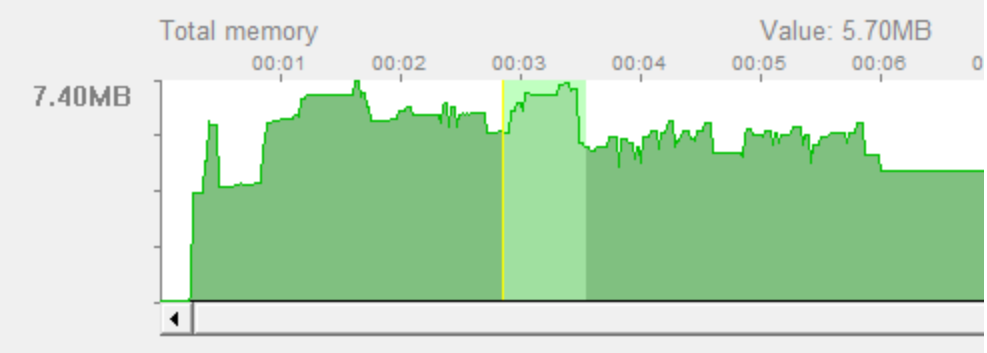
The following three options all open the Analysis Tab, adding a callstack for every allocation or deallocation in the selected sample period(s).

This enables a deeper inspection of where and how objects of this size are allocated or freed.

- **Show Allocation Locations** › shows allocations only
- **Show Deallocation Locations** › shows deallocations only
- **Show Allocation and Deallocation Locations** › shows both

The range of data to which the menu applies is as follows:

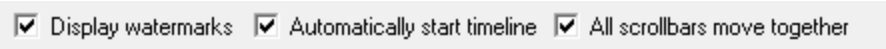
- If you have selected a range of sample periods, all the events in that range will be shown



- If no range is highlighted, we will only show the events in the single sample period at the exact point you click the mouse

Timeline controls

There are just three check boxes to control the display of data.



- **Display watermarks** › shows all watermarks on the graphs as a dashed vertical blue line
- **Automatically start timeline** › starts the timeline at the point the application is launched
- **All scrollbars move together** › synchronises the use of scrollbars on each timeline so they all move together - just uncheck this for independent control

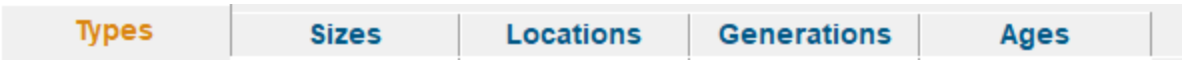
If you manually scroll the timeline to look at some data and then wish the timeline to scroll as new data is added, scroll the timeline to the extreme right. It will automatically start scrolling as new data is added.

➡ You can control how long the timeline can show data for (and hence the memory used for the timeline) in the Timeline page of the settings dialog.

3.7.4 Statistics

The **Statistics** tab provides access to statistics for types, sizes, allocation locations, generations and ages.

Click on an item in the picture below to find out more about each of the tabbed windows, or use the list further below:

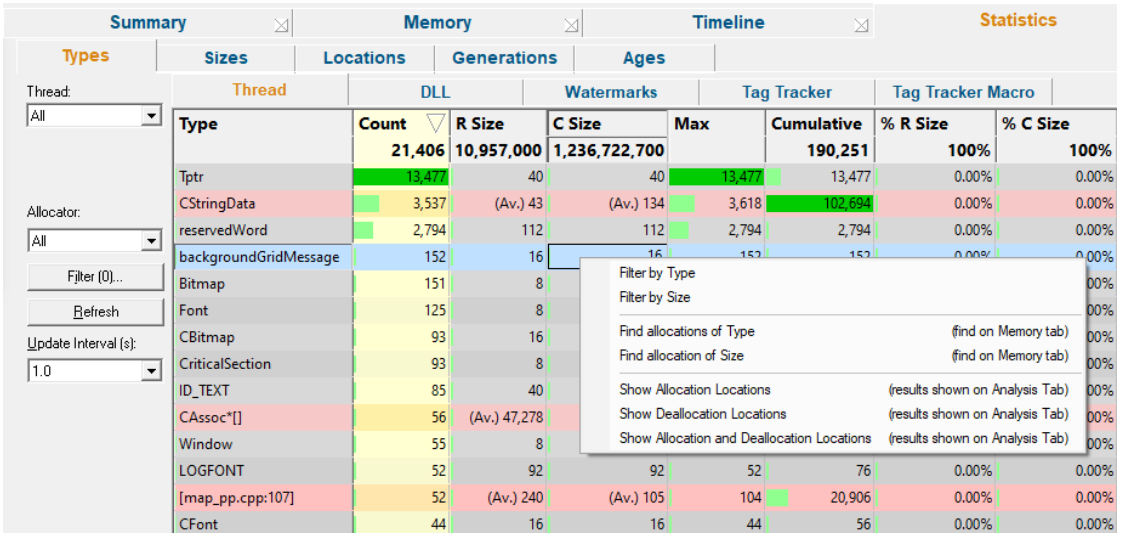


3.7.4.1 Types

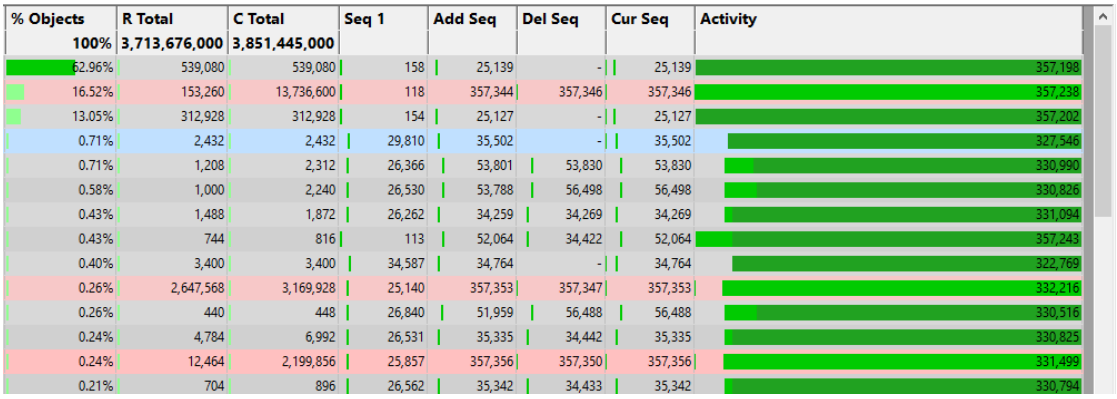
The **Types** tab summarises all the types of objects in the target program.

Read on, or click a part of the images below to jump straight to the help for that area.

The left hand side of the view shows controls and a variety of statistics:



The right hand side of the view shows information about event sequence id:



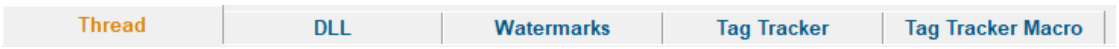
The view lists of all the objects in the program with their type, size and number allocated as well as other information.

You can constrain the displayed objects to those in particular threads or DLLs, and those between watermarks or being tracked by tag trackers.

➜ Much of the statistics and settings here are also relevant for the Sizes tab.

Types tab

At the top of the Types tab is a set of five more tabs allowing you to change the scope within which objects and types are shown in the table.



- **Thread** › show stats for an individual thread in the list or **All** threads (the default)
- **DLL** › show stats for a chosen DLL or **All** of them (the default)
- **Watermark** › filter the display to show objects used between two watermarks
- **Tag Tracker** › display stats for an individual tag tracker or for **All** trackers
- **Tag Tracker Macro** › shows amalgamated values of all object types in each tag tracker group, allowing overviews of memory use within each tag

Colours used in the display

Each object type's row is coloured according to whether the object has:

an increasing count
a decreasing count
a static count - i.e. not recently changed
a zero count

The importance of each value within most of the columns is highlighted with a percentage bar:

the object type with the maximum value in a given column
relative contribution of the object value in each column

➜ See also the Data Highlighting dialog to customise the first two colours.

The data columns

The data in each column is summarised below and described in more detail further down, with the help of examples.

Some of the header columns display a total for the column underneath the column name.

- **Type** object type
- **R Size** running allocation size
- **C Size** cumulative allocation size
- **Count** number of live allocations
- **Max** maximum number of live allocations
- **Cumulative** cumulative number of allocations
- **% R Size** size as percentage of total running allocations
- **% C Size** size as percentage of total cumulative allocations
- **% Objects** object count as percentage of total number of objects
- **R Total** running total size (R Size x Count)
- **C Total** cumulative total size (C Size x Cumulative)
- **Seq 1** sequence id of first allocation of this type
- **Add Seq** id of most recent allocation
- **Del Seq** id of most recent deallocation
- **Cur Seq** id of most recent allocation or deallocation
- **Activity** the span between first and most recent event sequence ids

To best explain the numbers in each column we'll use a simple example scenario which has a series of `char[]` allocations and deallocations shown in order of their event sequence id.

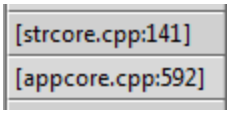
This event sequence below (not actual code) will be used to demonstrate how some of the figures in the columns would change with each event. Note this would leak allocations s4 and s5.

```
1  s1 = new char [10];
2  delete s1;

3  s1 = new char [10];
4  s2 = new char [10];
5  s3 = new char [20];
6  s4 = new char [20];
7  s5 = new char [10];

8  delete s1;
9  delete s2;
10 delete s3;
```

Type

The type is simply that of the memory allocation, determined by parsing the source code at the allocation location, eg `char[]` in our example above.

When a type cannot be determined, a pseudo-type is created by merging the filename and the line number for the allocation:



About Running and Cumulative values (R & C)

Some of the columns make reference to R or C values.

Running (R) values increase with each allocation and then decrease with deallocations, thus giving a snapshot of the live or current state.

Cumulative (C) figures only ever increase, ignoring deallocations, so give a quantitative extent of overall or historical activity.

R Total and C Total

Although not first in the column order, these two values are used to calculate some of the other column values.

- **R Total** › the running total gives the total size of all live objects of each type
- **C Total** › the cumulative total tells you how much of each type has ever been allocated

		R Total	C Total
1	s1 = new char [10];	+10 10	+10 10
2	delete s1;	-10 0	10
3	s1 = new char [10];	+10 10	+10 20
4	s2 = new char [10];	+10 20	+10 30
5	s3 = new char [20];	+20 40	+20 50
6	s4 = new char [20];	+20 60	+20 70
7	s5 = new char [10];	+10 70	+10 80
8	delete s1;	-10 60	80
9	delete s2;	-10 50	80
10	delete s3;	-20 30	80

R Size and C size

- **R Size** › the running size for each object type is **R Total / Count** - i.e. the average size of all *live* objects

		R Total	Count	R Size
1	s1 = new char [10];	+10 10	+1 1	10 / 1 = 10
2	delete s1;	-10 0	-1 0	0
3	s1 = new char [10];	+10 10	+1 1	10 / 1 = 10
4	s2 = new char [10];	+10 20	+1 2	20 / 2 = 10
5	s3 = new char [20];	+20 40	+1 3	40 / 3 = 13.3
6	s4 = new char [20];	+20 60	+1 4	60 / 4 = 15
7	s5 = new char [10];	+10 70	+1 5	70 / 5 = 14
8	delete s1;	-10 60	-1 4	60 / 4 = 15
9	delete s2;	-10 50	-1 3	50 / 3 = 16.7
10	delete s3;	-20 30	-1 2	30 / 2 = 15

- **C Size** › the cumulative size for each object type is **C Total / Cumulative** - i.e. the average size of all allocated objects whether live or freed

		C Total	Cumulative	C Size
1	s1 = new char [10];	+10 10	+1 1	10 / 1 = 10
2	delete s1;	10	1	
3	s1 = new char [10];	+10 20	+1 2	20 / 2 = 10
4	s2 = new char [10];	+10 30	+1 3	30 / 3 = 10
5	s3 = new char [20];	+20 50	+1 4	50 / 4 = 12.5
6	s4 = new char [20];	+20 70	+1 5	70 / 5 = 14
7	s5 = new char [10];	+10 80	+1 6	80 / 6 = 13.3
8	delete s1;	80	6	
9	delete s2;	80	6	
10	delete s3;	80	6	

For objects with a fixed size these values will both always be the size of the object.

Objects with an average size calculation have the value prefixed with (Av.):

Count, Max and Cumulative

- **Count** › the number of *live* objects of each type, so in our example the count increases and decreases, resulting in an overall increase of 2
- **Max** › the maximum value that the count ever reaches - i.e. the peak number of live objects at any one time
- **Cumulative** › increases with every single allocation, giving a historical total.

		Count	Max	Cumulative
1	s1 = new char [10];	+1 1	+1 1	+1 1
2	delete s1;	-1 0	1	1
3	s1 = new char [10];	+1 1	1	+1 2
4	s2 = new char [10];	+1 2	+1 2	+1 3
5	s3 = new char [20];	+1 3	+1 3	+1 4
6	s4 = new char [20];	+1 4	+1 4	+1 5
7	s5 = new char [10];	+1 5	+1 5	+1 6
8	delete s1;	-1 4	5	6
9	delete s2;	-1 3	5	6
10	delete s3;	-1 2	5	6

Percentage contributions: % R Size, % C Size and % Objects

The percentage figures give an indication of how important each type is relative to others:

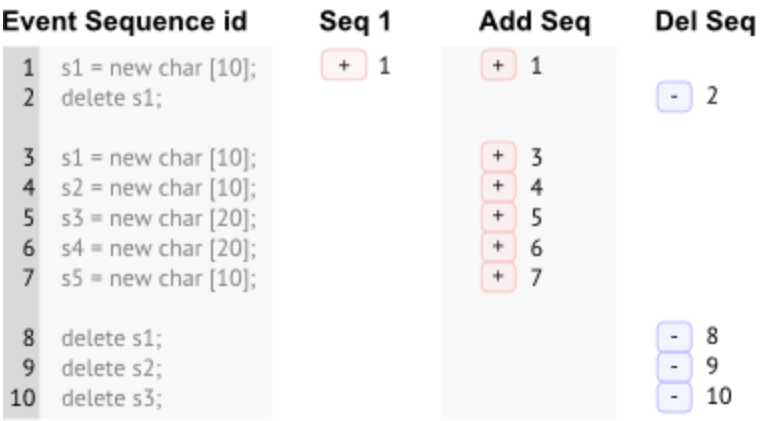
- **% R Size** › the percentage of total live object sizes represented by each type i.e. the contribution of R Total for each object to that of all object types combined
- **% C Size** › similar to above but for cumulative figures - the contribution of each C Total to that of all object types combined
- **% Objects** › the contribution of each object count to the combined counts of all objects

Sequence numbers: Seq 1, Add Seq, Del Seq and Cur Seq

The sequence numbers show significant event sequence ids relating to each object type:

- **Seq 1** › the first allocation sequence id for each object type
- **Add Seq** › the sequence id for the most recent allocation
- **Del Seq** › the sequence id for the most recent deallocation

  If no objects of a type have ever been deleted, the value just shows a hyphen
- **Cur Seq** › the most recent event sequence id related to each object type - usually the greater of Add Seq or Del Seq



Looking at the values for the range Seq 1 to Cur Seq helps give an indication of the span of activity of each object type.

Event sequence id markers

The four sequence id columns show markers visualizing the event's position relative to the the total number of events so far.

Each row shows green markers denoting the relative position of the first and most recent sequence ids for the object type.

In the following example there have been approximately 3000 events to date:

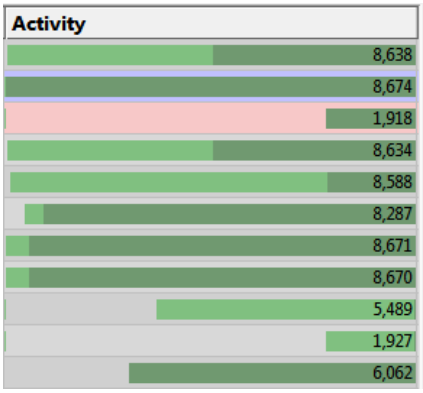
Seq 1	Add Seq	Del Seq	Cur Seq
8,395	19,158,996	19,158,991	19,158,996
50,203	19,159,046	19,143,628	19,159,046
1,139	214,564	-	214,564
1,040	197,291	-	197,291
203,589	19,159,032	19,143,627	19,159,032

These markers can help you see the relative timing and order of object type allocation and deallocation much more quickly than scanning through the numbers alone.

Object activity

The 'activity' of an object type is the span between its first allocation id and the most recent event sequence id at which at least one object was still live.

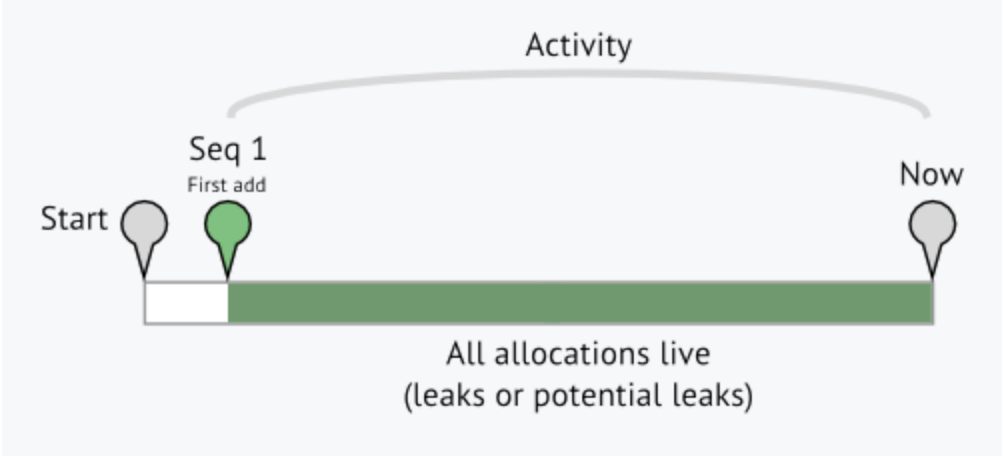
The **Activity** column in the view shows a graph of that lifespan, with the value being the number of events spanned:



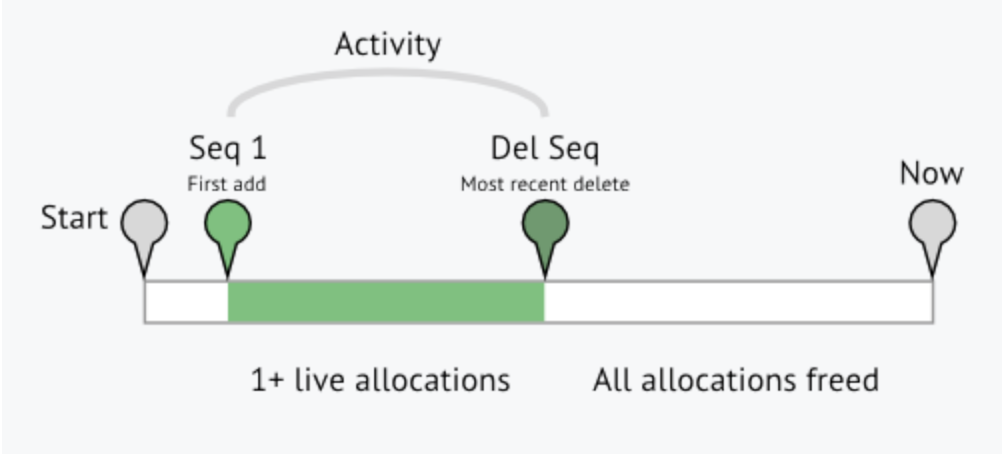
Below are some examples of what the activity graph means, where:

- *Start* corresponds to the first known event in the target program
- *Now* indicates the most recent event sequence id for any event in the target program

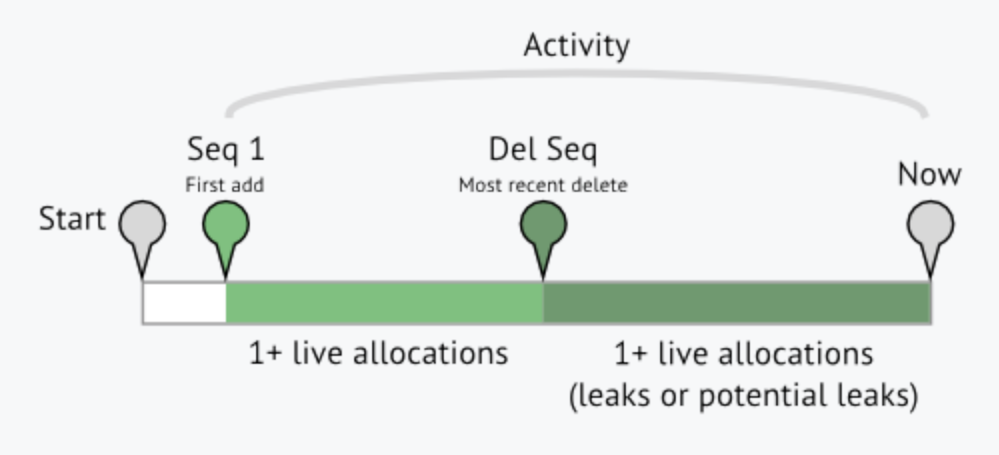
Example 1: Some objects of the given type have been allocated, but none freed



Example 2: Some objects of the given type have been allocated, and all have been freed

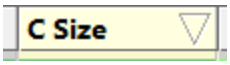


Example 3: Some objects of the given type have been allocated. Some of those have been freed, but not all



Sorting columns

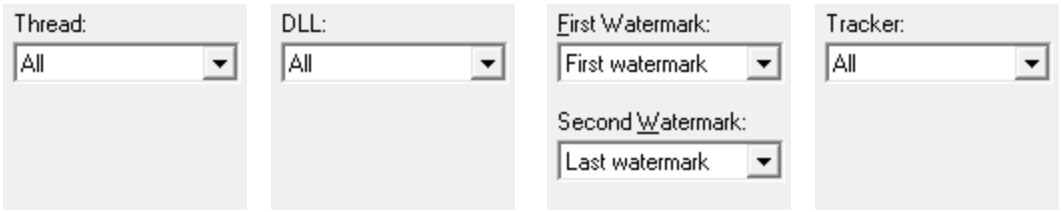
Sorted columns are shown yellow. Just click on the column header to change the sorting column or it's sort direction order.



The sorted column takes effect in each of the five object tab views, Thread, DLL, etc.

Object tab options

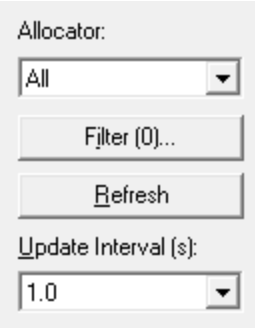
Each of the five object tab views has different options at the top, with Tag Tracker Macro having none.



The first two tabs allow you to choose a single Thread or DLL for which to show object types used.

The Memory tab topic describes use of the Watermark and Tracker options in detail.

The following options are common to all five tabs:



Updating the display

- **Refresh** › updates the display - as does the button on the Tools menu and toolbar

  With an update interval set to Never, you'll need to use this Refresh button to update the display.

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds - or never!

Allocator

- **Allocator** › updates the display to show types allocated by the specified allocator

The allocator can be one of the values in this table. The default is All.

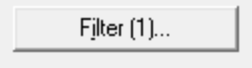
All	All types
All Native Memory	All native memory types
All Native Handles	All native handle types
All .Net	All .Net types
CRT	All types from CRT allocations
HeapAlloc	All types from HeapAlloc allocations
LocalAlloc	All types from LocalAlloc allocations
GlobalAlloc	All types from GlobalAlloc allocations
SysAllocString	All types from SysAllocString allocations
CoTaskMemAlloc	All types from CoTaskMemAlloc allocations
IMalloc	All types from allocations tracked by IMalloc
NetAPI	All types from NetAPI allocations
Misc	All types from Misc allocations
VirtualAlloc	All types from VirtualAlloc allocations

VirtualAllocEx	All types from VirtualAllocEx allocations
VirtualAllocVlm	All types from VirtualAllocVlm allocations
User Defined (API)	All types from allocations reported by the User Defined Types API
Custom Hook	All types from allocations tracked by the Custom Hooks settings
COM	All types from COM allocations
Com AddRef	All types from COM AddRef tracking
OpenGL	All types from OpenGL allocations
Crypt	All types from Crypt API allocations
Handles	All types from handles not represented by other allocators in this list
GDI Handles	All types from GDI handle allocations
USER32 Handles	All types from USER32 handle allocations
Internet Handles	All types from internet related allocations (socket, WinHttp...)
Printer Handles	All types from WinSpool allocations
Fortran	All types from Fortran allocations
Delphi	All types from Delphi allocations
.Net Objects	All types from .Net objects
.Net Large Objects	All types from .Net large objects (>= 85,000 bytes in size)
.Net Handles	All types from .Net handles
.Net VTables	All types from .Net VTables

Local filters and settings

- **Filter...** > shows the local filters dialog for the types tab, allowing you to fine tune what is included or excluded from the display

The filter button also indicates the *number* of local filters, although not all of these may be enabled

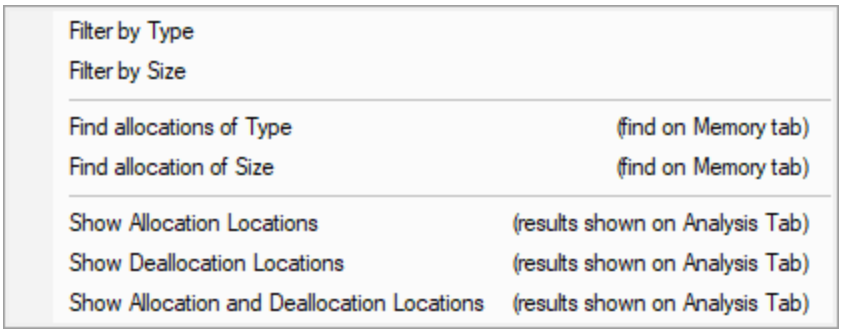


➜ See also the popup menu options to create filters based on the selected object type

See the Data Highlighting settings dialog to set the appearance of rows in the table depending on the object count and whether it's increasing or decreasing

Types view popup menu

The following popup menu is available over the data area to allow filter creation or data drill-down.



Menu option: Filters

- **Filter By Type** › creates a filter using the selected object type, so that it is removed from the view
- **Filter By Size** › creates a filter based on the size of the selected object so all rows with an object type of this size are removed

Filters created this way are added to the local filters dialog where they can be modified or removed.

Filters apply to each of the five object tabs and do not affect the values in each column - only whether the object type is actually shown in the view.

Menu option: Finding types - drilling down into the data

- **Find allocation of Type** › searches the Memory tab for allocations of the selected type

Menu option: Finding sizes - drilling down into the data

- **Find allocation of Size** › searches the Memory tab for allocations of the selected size

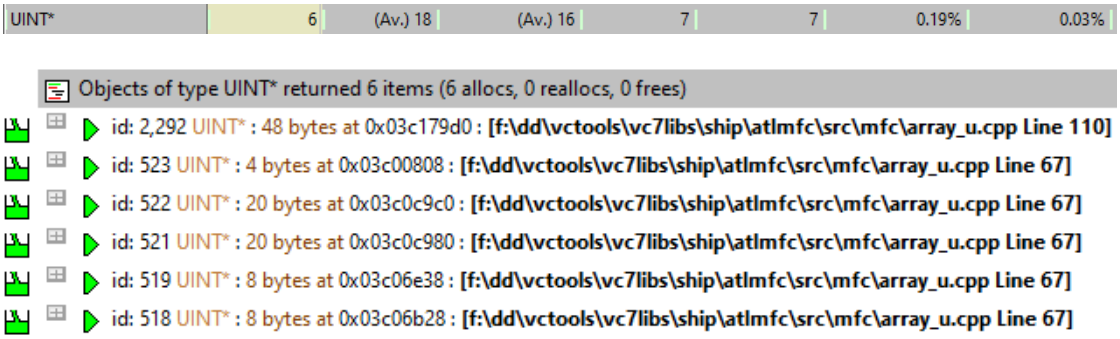
Menu option: Showing locations - drilling down into the data

The following three options all open the Analysis Tab, adding a callstack for every allocation or deallocation of the selected object type.

This enables a deeper inspection of where and how instances of an object type is allocated or freed.

- **Show Allocation Locations** › shows allocations only
- **Show Deallocation Locations** › shows deallocations only
- **Show Allocation and Deallocation Locations** › shows both

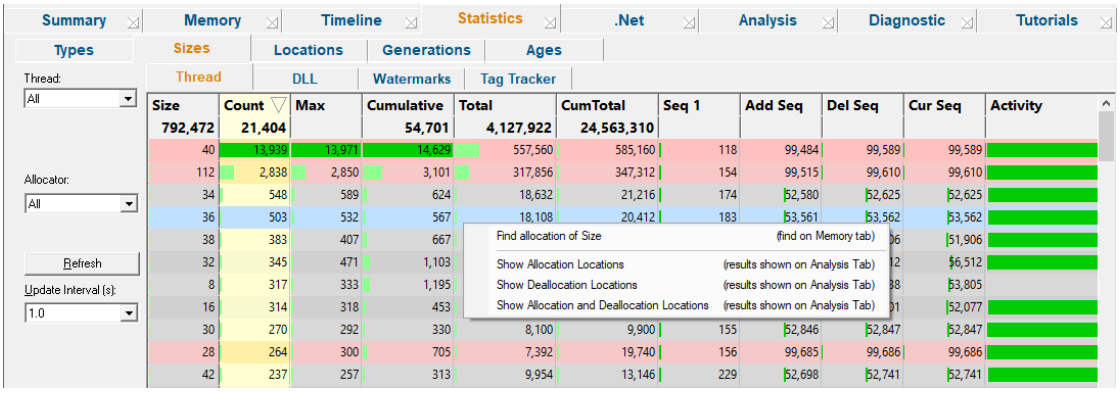
For example, showing allocations for the following row in the Types tab will show the callstacks for six allocations of various sizes in the Analysis tab below:



3.7.4.2 Sizes

The **Sizes** tab summarises all the allocations in the target program by their size, as opposed to by type.

Click a part of the image below to jump straight to the help for that area.



The display shows information about allocations in threads, allocations in DLLs, allocations for each watermark set in the session and allocations tracked by tag trackers.

The view lists of all the objects in the program in size order with the numbers of each allocated as well as other information.

As in the types tab, here you can constrain the objects shown to those in particular threads or DLLs, and those between watermarks or being tracked by tag trackers.

The features here are roughly a subset of those on the Types tab. If you're going through this help and already read the about the Types tab, you could skip this topic and go on to the Timeline tab if you wish.

Size tabs

At the top of the Size tab is a set of four tabs, each filtering which object sizes are shown in the table.

- **Thread** › show stats for an individual thread in the list or **All** threads (the default)
- **DLL** › show stats for a chosen DLL or **All** of them (the default)
- **Watermark** › filter the display to show objects used between two watermarks
- **Tag Tracker** › display stats for an individual tag tracker or for **All** trackers

Colours used in the display

Each row is coloured according to whether the object has:

an increasing count for the number of live objects of the size
a decreasing count
a static count
a zero count - i.e. where all allocated objects of the size have been freed

The importance of each value within the column is highlighted with a percentage bar:

the object size with the maximum value in a given column (not shown for all columns)
relative contribution of the value in each column

➜ See also the Data Highlighting settings dialog to customise the first two colours.

The size data columns

The data in each column for all types of allocation are summarised below:

Some of the header columns display a total for the column underneath the column name.

- **Size** object size
- **Count** number of live allocations of each size
- **Max** maximum number of live allocations
- **Cumulative** cumulative number of allocations

- **Total** running (live) total size (Size x Count)
- **Cum Total** cumulative total size (Size x Cumulative)
- **Seq 1** sequence id of first allocation of this size
- **Add Seq** id of most recent allocation
- **Del Seq** id of most recent deallocation
- **Cur Seq** id of most recent allocation or deallocation
- **Activity** the span between first and most recent event sequence ids

To best explain the numbers in each column we'll use a simple example scenario which has a series of `char[]` allocations and deallocations shown in order of their event sequence id.

This event sequence below (not actual code) will be used to demonstrate how some of the figures in the columns would change with each event. Note this would leak allocations s4 and s5.

```
1  s1 = new char [10];
2  delete s1;

3  s1 = new char [10];
4  s2 = new char [10];
5  s3 = new char [20];
6  s4 = new char [20];
7  s5 = new char [10];

8  delete s1;
9  delete s2;
10 delete s3;
```

Size

The size is simply that of the allocated memory. Many different types of objects may have the same size, especially the smaller sizes.

When a size cannot be determined, a size of -1 may be used:

Zero is a valid allocation size, for example `new char[0]` or `SysAllocString(L"")`.

Count, Max and Cumulative

- **Count** › the number of *live* objects of each size, so in our example the count for objects of size 10 increases and decreases, resulting in an overall increase of 1
- **Max** › the maximum value that the count ever reaches - i.e. the peak number of live objects of each size at any one time
- **Cumulative** › increases with every single allocation, giving a historical total

	For objects of size 10:	Count	Max	Cumulative
1	s1 = new char [10];	+1 1	+1 1	+1 1
2	delete s1;	-1 0	1	1
3	s1 = new char [10];	+1 1	1	+1 2
4	s2 = new char [10];	+1 2	+1 2	+1 3
5	s3 = new char [20];			3
6	s4 = new char [20];			3
7	s5 = new char [10];	+1 3	+1 3	+1 4
8	delete s1;	-1 2	3	4
9	delete s2;	-1 1	3	4
10	delete s3;		3	4

Total and Cumulative Total

- **Total** › the running total gives the total size of all live objects of each type. This is simply Size * Count
- **C Total** › the cumulative total tells you how much of each type has ever been allocated. This is Size * Cumulative

	For objects of size 10:	Total	Cum Total
1	s1 = new char [10];	+10 10	+10 10
2	delete s1;	-10 0	10
3	s1 = new char [10];	+10 10	+10 20
4	s2 = new char [10];	+10 20	+10 30
5	s3 = new char [20];		30
6	s4 = new char [20];		30
7	s5 = new char [10];	+10 30	+10 40
8	delete s1;	-10 20	40
9	delete s2;	-10 10	40
10	delete s3;	10	40

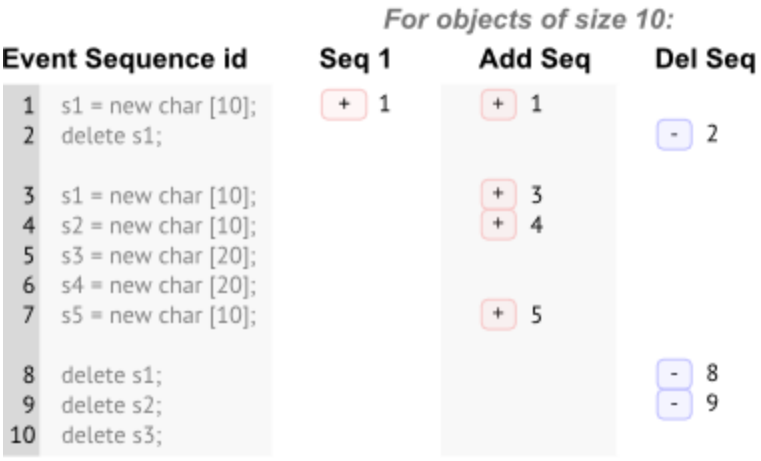
Sequence numbers: Seq 1, Add Seq, Del Seq and Cur Seq

The sequence numbers show significant event sequence ids relating to each size of object:

- **Seq 1** › the first allocation sequence id for each object size
- **Add Seq** › the sequence id for the most recent allocation
- **Del Seq** › the sequence id for the most recent deallocation

  If no objects of a given size have ever been deleted, the value just shows a hyphen

- **Cur Seq** › the most recent event sequence id related to each object size - usually the greater of Add Seq or Del Seq



Looking at the values for the range Seq 1 to Cur Seq helps give an indication of the span of activity of each size of object.

Event sequence id markers

The four sequence id columns show markers visualizing the event's position relative to the the total number of events so far.

Each row shows green markers denoting the relative position of the first and most recent sequence ids for the objects of each size.

In the following example there have been approximately 3000 events to date:

Seq 1	Add Seq	Del Seq	Cur Seq
8,395	19,158,996	19,158,991	19,158,996
50,203	19,159,046	19,143,628	19,159,046
1,139	214,564	-	214,564
1,040	197,291	-	197,291
203,589	19,159,032	19,143,627	19,159,032

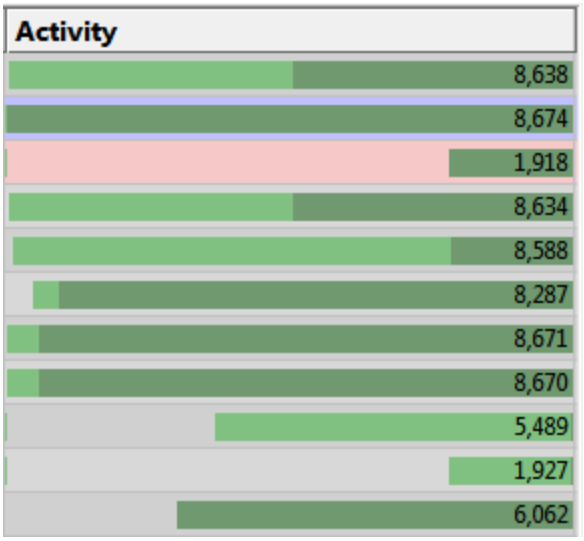
These markers can help you see the relative timing and order of object size allocation and deallocation much more quickly than scanning through the numbers alone.

Object activity

The 'activity' of an object size is the span between its first allocation id and the most recent event sequence id at which at least one object of that size was still live.

The **Activity** column in the view shows a graph of that lifespan, with the value being the number of events spanned:

➡ See also, explanatory examples of the graphs in the Types view.



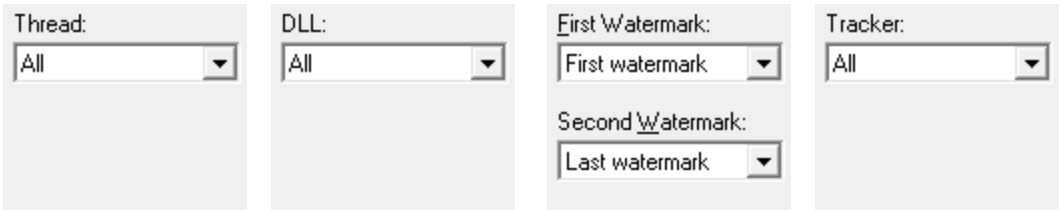
Sorting columns

Sorted columns are highlighted yellow. Just click on the column header to change the sorting column or it's sort direction order.

The sorted column takes effect in each of the five object tab views, Thread, DLL, etc.

Size tab options

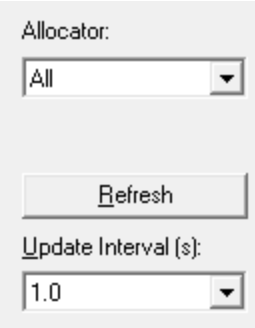
Each of the four size tab views has different options at the top, with Tag Tracker Macro having none.



The first two tabs allow you to choose a single Thread or DLL for which to show object types used.

The Memory tab topic describes use of the Watermark and Tracker options in detail.

The following options are common to all five tabs:



Updating the display

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds - or never!
- **Refresh** › updates the display - as does the button on the Tools menu and toolbar

  With an update interval set to Never, you'll need to use this Refresh button to update the display.

Allocator

- **Allocator** › updates the display to show types allocated by the specified allocator

The allocator can be one of the values in this table. The default is All.

All	All types
All Native Memory	All native memory types
All Native Handles	All native handle types
All .Net	All .Net types
CRT	All types from CRT allocations
HeapAlloc	All types from HeapAlloc allocations
LocalAlloc	All types from LocalAlloc allocations
GlobalAlloc	All types from GlobalAlloc allocations
SysAllocString	All types from SysAllocString allocations
CoTaskMemAlloc	All types from CoTaskMemAlloc allocations
IMalloc	All types from allocations tracked by IMalloc
NetAPI	All types from NetAPI allocations
Misc	All types from Misc allocations
VirtualAlloc	All types from VirtualAlloc allocations

VirtualAllocEx	All types from VirtualAllocEx allocations
VirtualAllocVlm	All types from VirtualAllocVlm allocations
User Defined (API)	All types from allocations reported by the User Defined Types API
Custom Hook	All types from allocations tracked by the Custom Hooks settings
COM	All types from COM allocations
Com AddRef	All types from COM AddRef tracking
OpenGL	All types from OpenGL allocations
Crypt	All types from Crypt API allocations
Handles	All types from handles not represented by other allocators in this list
GDI Handles	All types from GDI handle allocations
USER32 Handles	All types from USER32 handle allocations
Internet Handles	All types from internet related allocations (socket, WinHttp...)
Printer Handles	All types from WinSpool allocations
Fortran	All types from Fortran allocations
Delphi	All types from Delphi allocations
.Net Objects	All types from .Net objects
.Net Large Objects	All types from .Net large objects (>= 85,000 bytes in size)
.Net Handles	All types from .Net handles
.Net VTables	All types from .Net VTables

Display settings

- **Display...** > shows the Data Highlighting settings dialog to set the appearance of rows in the table depending on the object count and whether it's increasing or decreasing

Sizes view popup menu 

The following popup menu provides options for examine data in more detail in the Analysis tab.

Menu option: Finding sizes - drilling down into the data

- **Find allocation of Size** › searches the Memory tab for allocations of the selected size

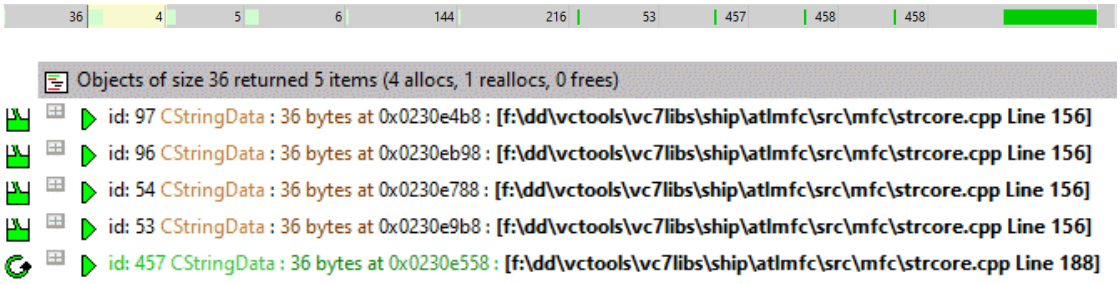
Menu option: Showing locations - drilling down into the data

The following three options all open the Analysis Tab, adding a callstack for every allocation or deallocation of the selected object size.

This enables a deeper inspection of where and how objects of this size are allocated or freed.

- **Show Allocation Locations** › shows allocations only
- **Show Deallocation Locations** › shows deallocations only
- **Show Allocation and Deallocation Locations** › shows both

For example, showing allocations for the following row in the Sizes tab will show the callstacks for four allocations and one reallocation of 36 bytes in the Analysis tab below:

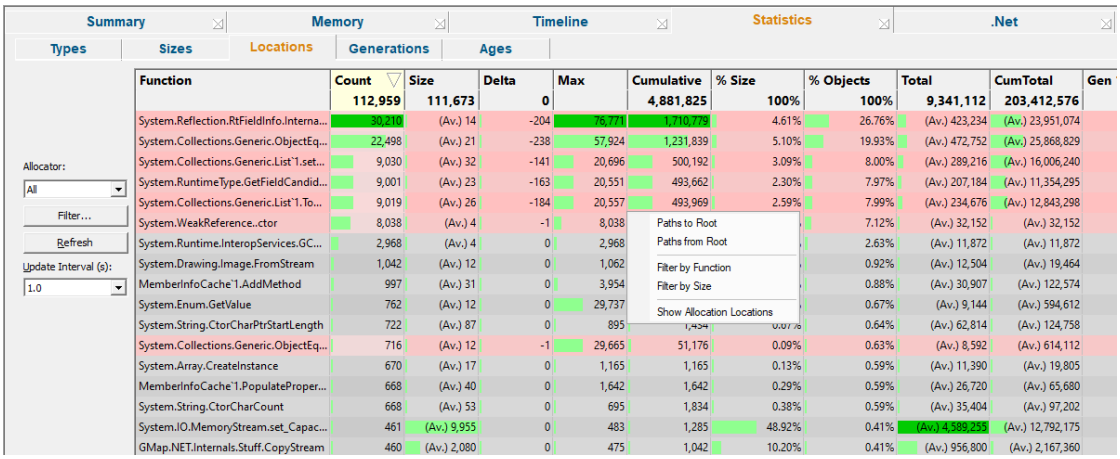


3.7.4.3 Locations

The **Locations** tab summarises all the allocations in the target program by their allocation location, as opposed to by type.

Click a part of the image below to jump straight to the help for that area.

The left hand side of the view shows controls and a variety of statistics:



Function	Count 112,959	Size 111,673	Delta 0	Max	Cumulative 4,881,825	% Size 100%	% Objects 100%	Total 9,341,112	CumTotal 203,412,576	Gen
System.Reflection.RtFieldInfo.Interna...	30,210	(Av.) 14	-204	76,771	1,710,779	4.61%	26.76%	(Av.) 423,234	(Av.) 23,951,074	
System.Collections.Generic.ObjectEq...	22,498	(Av.) 21	-238	57,924	1,231,839	5.10%	19.93%	(Av.) 472,752	(Av.) 25,868,829	
System.Collections.Generic.List`1.set...	9,030	(Av.) 32	-141	20,696	500,192	3.09%	8.00%	(Av.) 289,216	(Av.) 16,006,240	
System.RuntimeType.GetFieldCandid...	9,001	(Av.) 23	-163	20,551	493,662	2.30%	7.97%	(Av.) 207,184	(Av.) 11,354,295	
System.Collections.Generic.List`1.To...	9,019	(Av.) 26	-184	20,557	493,969	2.59%	7.99%	(Av.) 234,676	(Av.) 12,843,298	
System.WeakReference..ctor	8,038	(Av.) 4	-1	8,038			7.12%	(Av.) 32,152	(Av.) 32,152	
System.Runtime.InteropServices.GC...	2,968	(Av.) 4	0	2,968			2.63%	(Av.) 11,872	(Av.) 11,872	
System.Drawing.Image.FromStream	1,042	(Av.) 12	0	1,062			0.92%	(Av.) 12,504	(Av.) 19,464	
MemberInfoCache`1.AddMethod	997	(Av.) 31	0	3,954			0.88%	(Av.) 30,907	(Av.) 122,574	
System.Enum.GetValue	762	(Av.) 12	0	29,737			0.67%	(Av.) 9,144	(Av.) 594,612	
System.String.CtorCharPtrStartLength	722	(Av.) 87	0	895			0.64%	(Av.) 62,814	(Av.) 124,758	
System.Collections.Generic.ObjectEq...	716	(Av.) 12	-1	29,665	51,176	0.09%	0.63%	(Av.) 8,592	(Av.) 614,112	
System.Array.CreateInstance	670	(Av.) 17	0	1,165	1,165	0.13%	0.59%	(Av.) 11,390	(Av.) 19,805	
MemberInfoCache`1.PopulateProper...	668	(Av.) 40	0	1,642	1,642	0.29%	0.59%	(Av.) 26,720	(Av.) 65,680	
System.String.CtorCharCount	668	(Av.) 53	0	695	1,834	0.38%	0.59%	(Av.) 35,404	(Av.) 97,202	
System.IO.MemoryStream.set_Capac...	461	(Av.) 9,955	0	483	1,285	48.92%	0.41%	(Av.) 4,589,255	(Av.) 12,792,175	
GMap.NET.Internals.Stuff.CopyStream	460	(Av.) 2,080	0	475	1,042	10.20%	0.41%	(Av.) 956,800	(Av.) 2,167,360	

The right hand side of the view shows information about generations and event sequence id:

Gen 1	Add Gen	Del Gen	Seq 1	Add Seq	Del Seq	Cur Seq	Activity
0	34	33	77,879	10,726,569	10,591,191	10,726,589	10,989,665
0	34	33	2,378	10,726,745	10,591,184	10,726,755	11,065,166
0	34	33	846	10,726,912	10,591,186	10,726,912	11,066,698
0	34	33	75,020	10,727,083	10,591,185	10,727,087	10,992,524
0	34	33	75,034	10,727,251	10,591,187	10,727,255	10,992,510
0	34	-	823	10,713,918	-	10,713,918	11,066,721
0	34	-	1,211	10,681,822	-	10,681,822	11,066,333
1	34	33	461,398	10,708,533	10,527,777	10,708,533	10,606,146
0	0	0	74,996	174,898	334,324	334,324	10,992,548
0	33	33	2,373	10,405,856	10,565,334	10,565,334	11,065,171
0	34	30	78,236	10,634,541	9,472,361	10,634,541	10,989,308
0	34	33	2,368	10,714,066	10,583,514	10,714,066	11,065,176
0	34	33	228	10,676,983	10,549,531	10,676,983	11,067,316
0	0	0	75,054	113,743	275,218	275,218	10,992,490
0	0	0	75,038	113,040	274,586	274,586	10,992,506
0	0	0	107,723	10,684,807	10,685,015	10,685,015	10,959,821
0	34	32	111,968	10,652,991	10,433,700	10,652,991	10,955,576
0	34	33	115,449	10,668,452	10,583,513	10,668,452	10,952,095
1	34	33	462,748	10,668,448	10,583,511	10,668,448	10,604,796
0	0	-	82,490	98,834	-	98,834	10,985,054

The view lists of all the objects in the program in allocation location order with the numbers of each allocated as well as other information.

The features here are roughly a subset of those on the Types tab. If you're going through this help and already read the about the Types tab, you could skip this topic and go on to the Timeline tab if you wish.

Colours used in the display

Each row is coloured according to whether the object has:

an increasing count for the number of live objects of the size

 a decreasing count

 a static count

 a zero count - i.e. where all allocated objects of the size have been freed

The importance of each value within the column is highlighted with a percentage bar:

 the object size with the maximum value in a given column (not shown for all columns)

 relative contribution of the value in each column

➡ See also the Data Highlighting settings dialog to customise the first two colours.

The locations data columns

The data in each column for all types of allocation are summarised below:

Some of the header columns display a total for the column underneath the column name.

- **Function** allocation location
- **Count** number of live allocations of each size
- **Size** size of the allocations made at this location
- **Delta** change in count since last refresh
- **Max** maximum number of live allocations
- **Cumulative** cumulative number of allocations
- **%Size** percentage of all allocations by total size
- **%Objects** percentage of all allocations by object count
- **Total** running (live) total size (Size x Count)
- **Cum Total** cumulative total size (Size x Cumulative)
- **Gen 1** first generation allocated
- **Add Gen** recent generation allocated
- **Del Gen** recent generation deallocated (or garbage collected)
- **Seq 1** sequence id of first allocation of this size
- **Add Seq** id of most recent allocation
- **Del Seq** id of most recent deallocation
- **Cur Seq** id of most recent allocation or deallocation
- **Activity** the span between first and most recent event sequence ids

Function

The function in which the memory was allocated for this allocation location.

Size

The size is simply that of the allocated memory. Many different types of objects may have the same size, especially the smaller sizes.

When a size cannot be determined, a size of -1 may be used:

Zero is a valid allocation size, for example `new char[0]` or `SysAllocString(L"")`.

Count, Max and Cumulative

- **Count** › the number of *live* objects for each allocation location, so in our example the count for objects of size 10 increases and decreases, resulting in an overall increase of 1
- **Max** › the maximum value that the count ever reaches - i.e. the peak number of live objects for each allocation location at any one time
- **Cumulative** › increases with every single allocation, giving a historical total

Total and Cumulative Total

- **Total** › the running total gives the total size of all live objects of each allocation location. This is simply Size * Count
- **C Total** › the cumulative total tells you how much for each allocation location has ever been allocated. This is Size * Cumulative

Generation numbers, Gen 1, Add Gen, Del Gen

The generation numbers show significant generation ids relating to each allocation location.

- **Seq 1** › the generation for the first allocation at this location
- **Add Seq** › the generation for the most recent allocation at this location
- **Del Seq** › the generation for the most recent deallocation of memory allocated at this location

    If no objects have ever been deleted for a given location, the value just shows a hyphen

Looking at the values for the range Gen 1 to Del Gen helps give an indication of the span of activity of each allocation location.

Sequence numbers: Seq 1, Add Seq, Del Seq and Cur Seq

The sequence numbers show significant event sequence ids relating to each allocation location:

- **Seq 1** › the first allocation sequence id for each allocation location
- **Add Seq** › the sequence id for the most recent allocation
- **Del Seq** › the sequence id for the most recent deallocation

    If no objects of a given size have ever been deleted, the value just shows a hyphen

- **Cur Seq** › the most recent event sequence id related to each allocation location - usually the greater of Add Seq or Del Seq

Looking at the values for the range Seq 1 to Cur Seq helps give an indication of the span of activity of each size of object.

Event sequence id markers

The four sequence id columns show markers visualizing the event's position relative to the the total number of events so far.

Each row shows green markers denoting the relative position of the first and most recent sequence ids for the objects of each allocation location.

In the following example there have been over 19 million events:

Seq 1	Add Seq	Del Seq	Cur Seq
8,395	19,158,996	19,158,991	19,158,996
50,203	19,159,046	19,143,628	19,159,046
1,139	214,564	-	214,564
1,040	197,291	-	197,291
203,589	19,159,032	19,143,627	19,159,032

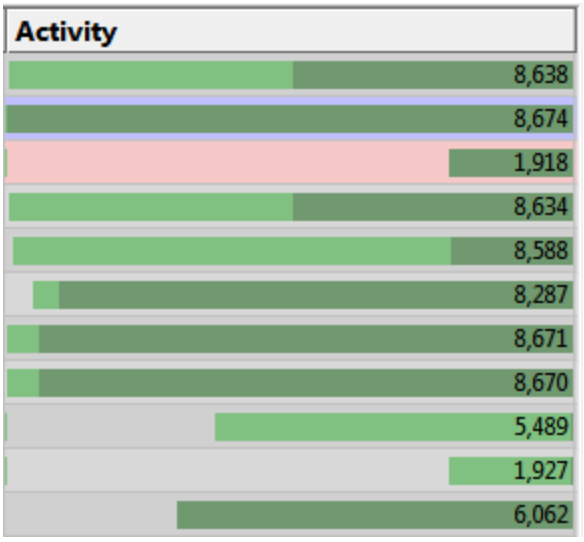
These markers can help you see the relative timing and order of object size allocation and deallocation much more quickly than scanning through the numbers alone.

Allocation location activity

The 'activity' of an allocation location is the span between its first allocation id and the most recent event sequence id at which at least one object of that size was still live.

The **Activity** column in the view shows a graph of that lifespan, with the value being the number of events spanned:

➜ See also, explanatory examples of the graphs in the Types view.

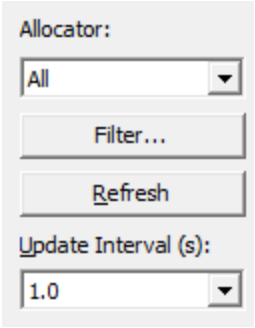


Sorting columns

Sorted columns are highlighted yellow. Just click on the column header to change the sorting column or it's sort direction order.

Locations tab options

The following options are available:



Updating the display

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds - or never!
- **Refresh** › updates the display - as does the button on the Tools menu and toolbar

  With an update interval set to Never, you'll need to use this Refresh button to update the display.

Allocator

- **Allocator** › updates the display to show types allocated by the specified allocator

The allocator can be one of the values in this table. The default is All.

All	All types
All Native Memory	All native memory types
All Native Handles	All native handle types
All .Net	All .Net types
CRT	All types from CRT allocations
HeapAlloc	All types from HeapAlloc allocations
LocalAlloc	All types from LocalAlloc allocations
GlobalAlloc	All types from GlobalAlloc allocations
SysAllocString	All types from SysAllocString allocations
CoTaskMemAlloc	All types from CoTaskMemAlloc allocations
IMalloc	All types from allocations tracked by IMalloc
NetAPI	All types from NetAPI allocations
Misc	All types from Misc allocations
VirtualAlloc	All types from VirtualAlloc allocations
VirtualAllocEx	All types from VirtualAllocEx allocations
VirtualAllocVlm	All types from VirtualAllocVlm allocations
User Defined (API)	All types from allocations reported by the User Defined Types API
Custom Hook	All types from allocations tracked by the Custom Hooks settings
COM	All types from COM allocations
Com AddRef	All types from COM AddRef tracking
OpenGL	All types from OpenGL allocations
Crypt	All types from Crypt API allocations
Handles	All types from handles not represented by other allocators in this list
GDI Handles	All types from GDI handle allocations
USER32 Handles	All types from USER32 handle allocations
Internet Handles	All types from internet related allocations (socket, WinHttp...)

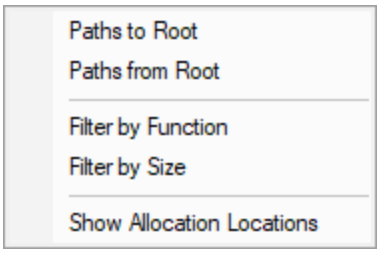
Printer Handles	All types from WinSpool allocations
Fortran	All types from Fortran allocations
Delphi	All types from Delphi allocations
.Net Objects	All types from .Net objects
.Net Large Objects	All types from .Net large objects (>= 85,000 bytes in size)
.Net Handles	All types from .Net handles
.Net VTables	All types from .Net VTables

Filter settings

- **Filter...** › shows the Location Filters settings dialog to edit the filters.

Locations view popup menu

The following popup menu provides options for filtering and examining data in more detail.



Menu option: Paths to Root, Paths from Root

The following options are only active for .Net allocation locations. These options are disabled for native allocation locations.

- **Paths to Root** › For all live objects allocated at this location displays all the paths from the object to the most recent heap dump roots.
- **Paths from Root** › For all live objects allocated at this location displays all the paths most recent heap dump roots to the live objects.

Menu option: Filter by Function, Filter by Size

The following options allow you to remove data from the display using filters.

- **Filter by Function** › creates a filter with the selected allocation location function.
- **Filter by Size** › creates a filter with the selected allocation location size.

By default filters prevent data from being displayed. You can change this using the Location Filters.

Filters can be edited using the **Filter...** button to display the Location Filters.

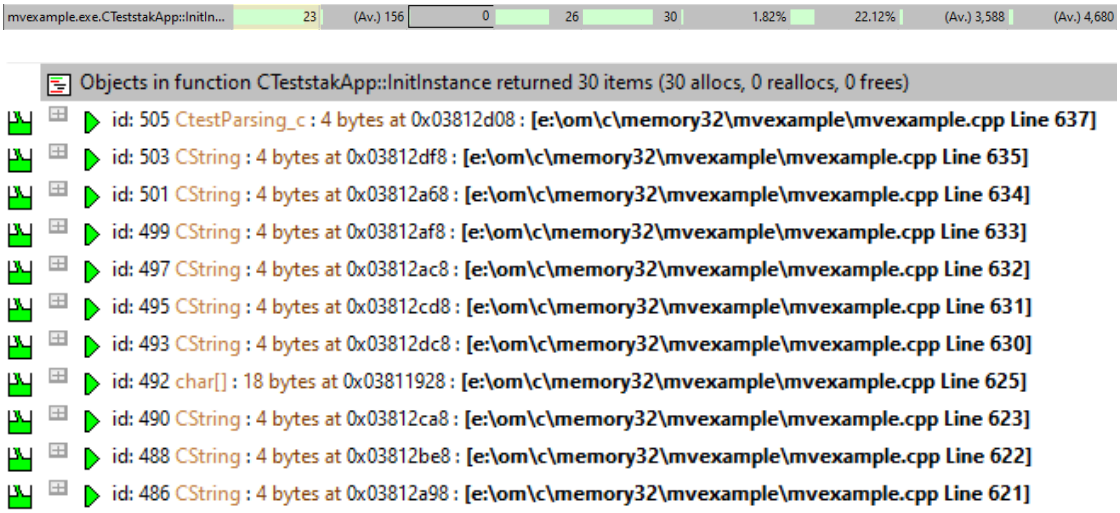
Menu option: Showing locations - drilling down into the data

The following option opens the Analysis Tab, adding a callstack for every allocation or deallocation of the selected object size.

This enables a deeper inspection of where and how objects of this size are allocated or freed.

- **Show Allocation Locations** › shows allocations only

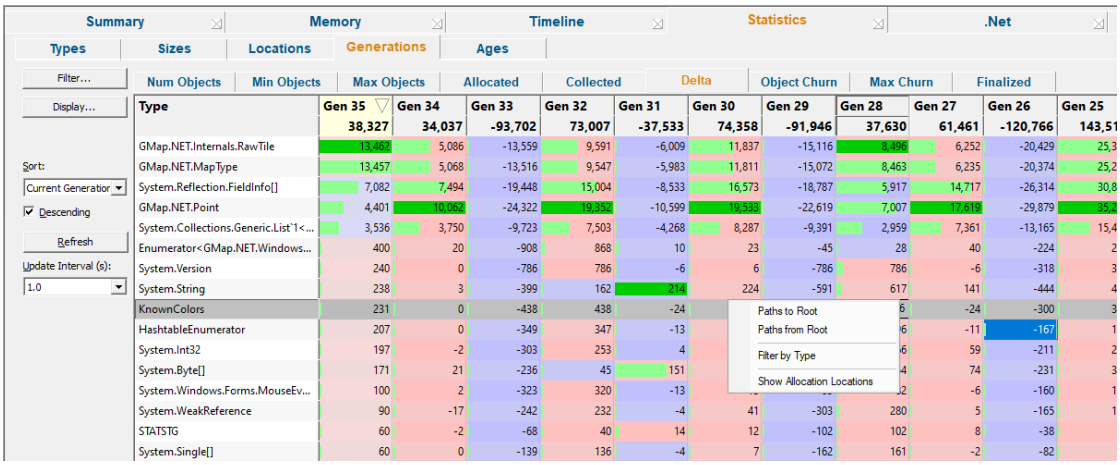
For example, showing allocations for the following row in the Locations tab will show the callstacks for four allocations of 6000 bytes in the Analysis tab below:



3.7.4.4 Generations

The **Generations** tab summarises all the allocations in the target program by their generation, as opposed to by type.

Click a part of the image below to jump straight to the help for that area.



The view lists of all the object types in the program in generation order, with the ability to view a range of related statistics.

The features here are roughly a subset of those on the Types tab. If you're going through this help and already read the about the Types tab, you could skip this topic and go on to the Timeline tab if you wish.

Generation tabs

Each tab provides a different group of generation statistics:

- **Num Objects** › How many objects of this type in this generation
- **Min Objects** › Minimum number of objects of this type in this generation
- **Max Objects** › Maximum number of objects of this type in this generation
- **Allocated** › Number of objects of this type allocated in this generation
- **Collected** › Number of objects of this type collected in this generation

  If you only see 0 values this may indicate leaking objects. For example:

[dnmvExample.Square]	0	0	0	0	0
[dnmvExample.Shape]	0	0	0	0	0

- **Delta** › Difference between number of objects of this type allocated and collected

  If you only see positive deltas this may indicate leaking objects. For example:

[dnmvExample.Square]	5,035	0	5,041	5,093	4,954
[dnmvExample.Circle]	4,965	0	4,959	4,907	5,046

- **Object Churn** › Object churn for this object type for this generation

  Percentage of objects collected in the generation divided by the number of objects allocated in the generation. This number can exceed 100%.

  High object churn - objects are being rapidly created and garbage collected

  Low object churn - objects are rarely being created and rarely garbage collected. Low churn may indicate leaking objects.

- **Max Churn** › Maximum churn for this object type for this generation

  Percentage of objects collected in the generation divided by the maximum number of objects in the generation.

- **Finalized** › How many objects of this type have been finalized.

  For many objects this will always be zero.

The above values need to assessed together to determine if a particular object type may be leaking. A collected count of zero, on it's own is meaningless. But if combined with a history of allocations and only positive deltas, that's a pretty strong hint that this object type may be leaking and that you need to examine it further.

The generations data columns

The first column is the object type.

All other columns represent each generation of data. The ordering of the generation columns is controlled by the Display Settings.

The header columns display a total for the column underneath the column name.

The value in each column is determined by which tab of data is being displayed.

Colours used in the display

Each row is coloured according to whether the object has:

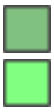
an increasing count for the number of live objects of the size

a decreasing count

a static count

a zero count - i.e. where all allocated objects of the size have been freed

The importance of each value within the column is highlighted with a percentage bar:



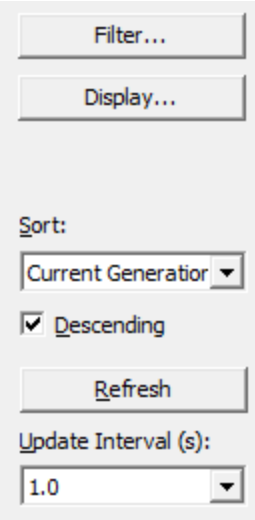
the object size with the maximum value in a given column (not shown for all columns)

relative contribution of the value in each column

➜ See also the Data Highlighting settings dialog to customise the first two colours.

Generations tab options

The following options are available:



Sorting columns

Sorted columns are highlighted yellow. Just click on the column header to change the sorting column or it's sort direction order.

As well as the column headers we also provide explicit sorting controls to provide an extra option that isn't available from the column headers ("Current generation").

- **Sort** › choose the generation to sort
- **Descending** › choose the sort direction

Updating the display

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds - or never!
- **Refresh** › updates the display - as does the button on the Tools menu and toolbar

With an update interval set to Never, you'll need to use this Refresh button to update the display.

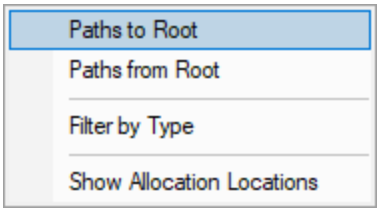
- **Display...** › display the Generation Settings dialog.

Filter settings

- **Filter...** › shows the Generations Filters settings dialog to edit the filters.

Generations view popup menu

The following popup menu provides options for filtering and examining data in more detail.



Menu option: Paths to Root, Paths from Root

The following options are only active for .Net object types. They are disabled for native object types.

- **Paths to Root** › For all live objects of this type display all the paths from the object to the most recent heap dump roots.
- **Paths from Root** › For all live objects of this type display all the paths most recent heap dump roots to the live objects.

Menu option: Filter by Function

The following options allow you to remove data from the display using filters.

- **Filter by Type** › creates a filter with the selected object type.

By default filters prevent data from being displayed. You can change this using the Generation Filters.

Filters can be edited using the **Filter...** button to display the Generation Filters.

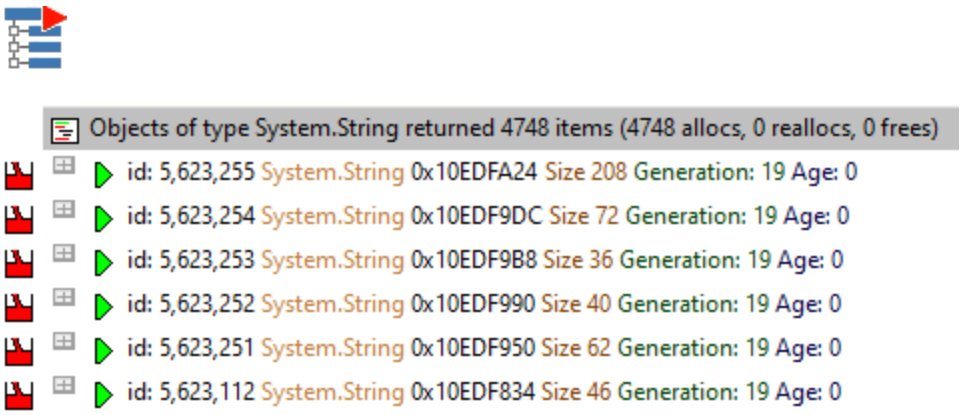
Menu option: Showing locations - drilling down into the data

The following option opens the Analysis Tab, adding a callstack for every allocation or deallocation of the selected object size.

This enables a deeper inspection of where and how objects of this size are allocated or freed.

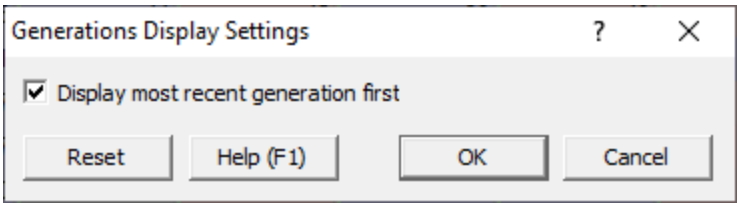
- **Show Allocation Generations** › shows allocations only

For example, showing allocations for the following row in the Generations tab will show the callstacks for 4748 allocations of **System.String** in the Analysis tab below:



3.7.4.4.1 Generation Settings Dialog

The Generation Settings dialog allows you to control how the display looks on the Generations view.



- **Display most recent generation first** › change how the generations are displayed.

Most recent generation first

Type	Gen 14 36,727	Gen 13 125,955	Gen 12 121,542	Gen 11 93,459	Gen 10 202,209	Gen 9 200,885	Gen 8 202,384	Gen 7 202,182	Gen 6 202,171	Gen 5 202,400	Gen 4 201,928	Gen 3 202,534	Gen 2 202,695	Gen 1 94,212	Gen 0 92,736
System.String	5,230	5,369	6,433	5,640	5,229	5,239	5,229	5,229	5,229	5,229	5,229	5,229	5,229	6,620	11,948
System.Windows.EffectiveValueEn...	1,048	1,566	1,086	1,576	989	989	989	989	989	989	989	989	989	1,129	1,892
System.RuntimeType	961	961	961	961	958	958	956	956	956	956	956	956	956	956	949
System.Object[]	792	970	1,192	913	801	801	801	801	801	801	802	801	801	1,165	2,530

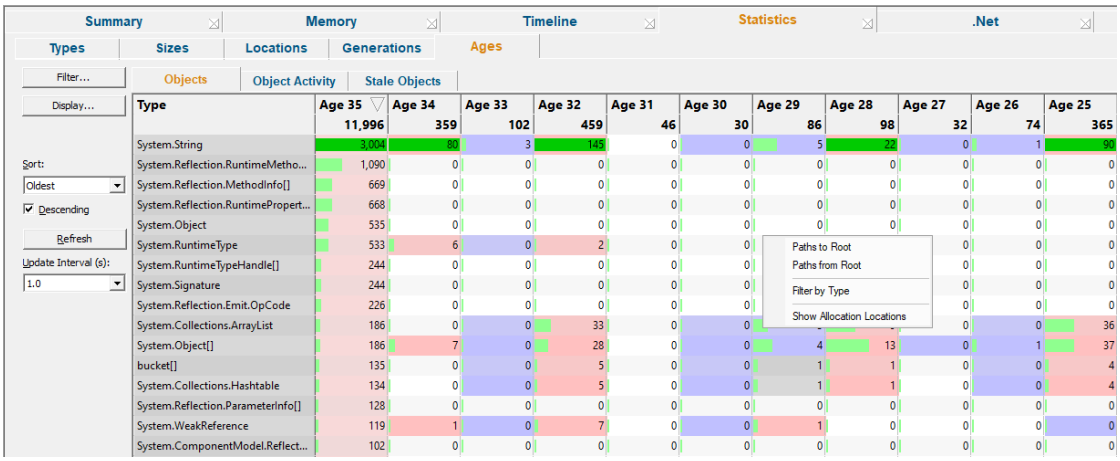
Most recent generation last

Type	Gen 0 36,727	Gen 1 125,955	Gen 2 121,542	Gen 3 93,459	Gen 4 202,209	Gen 5 200,885	Gen 6 202,384	Gen 7 202,182	Gen 8 202,171	Gen 9 202,400	Gen 10 201,928	Gen 11 202,534	Gen 12 202,695	Gen 13 94,212	Gen 14 92,736
System.String	11,948	6,620	5,229	5,229	5,229	5,229	5,229	5,229	5,229	5,239	5,229	5,640	6,433	5,369	5,230
System.Windows.EffectiveValueEn...	1,892	1,129	989	989	989	989	989	989	989	989	989	1,576	1,086	1,566	1,048
System.RuntimeType	949	956	956	956	956	956	956	956	956	958	958	961	961	961	961
System.Object[]	2,530	1,165	801	801	802	801	801	801	801	801	801	913	1,192	970	792

3.7.4.5 Ages

The **Ages** tab summarises all the allocations in the target program by their age, as opposed to by type.

Click a part of the image below to jump straight to the help for that area.



Type	Age 35	Age 34	Age 33	Age 32	Age 31	Age 30	Age 29	Age 28	Age 27	Age 26	Age 25
	11,996	359	102	459	46	30	86	98	32	74	365
System.String	3,004	80	3	145	0	0	5	22	0	1	90
System.Reflection.RuntimeMetho...	1,090	0	0	0	0	0	0	0	0	0	0
System.Reflection.MethodInfo[]	669	0	0	0	0	0	0	0	0	0	0
System.Reflection.RuntimePropert...	668	0	0	0	0	0	0	0	0	0	0
System.Object	535	0	0	0	0	0	0	0	0	0	0
System.RuntimeType	533	6	0	2	0	0	0	0	0	0	0
System.RuntimeTypeHandle[]	244	0	0	0	0	0	0	0	0	0	0
System.Signature	244	0	0	0	0	0	0	0	0	0	0
System.Reflection.Emit.OpCode	226	0	0	0	0	0	0	0	0	0	0
System.Collections.ArrayList	186	0	0	33	0	0	0	0	0	0	36
System.Object[]	186	7	0	28	0	0	4	13	0	1	37
bucket[]	135	0	0	5	0	0	1	1	0	0	4
System.Collections.Hashtable	134	0	0	5	0	0	1	1	0	0	4
System.Reflection.ParameterInfo[]	128	0	0	0	0	0	0	0	0	0	0
System.WeakReference	119	1	0	7	0	0	1	0	0	0	0
System.ComponentModel.Reflect...	102	0	0	0	0	0	0	0	0	0	0

The view lists of all the object types in the program in generation order, with the ability to view a range of related statistics.

The features here are roughly a subset of those on the Types tab. If you're going through this help and already read the about the Types tab, you could skip this topic and go on to the Timeline tab if you wish.

Age tabs

Each tab provides a different group of generation statistics:

- **Objects** › How many objects of this type are this age
- **Object Activity** › How many objects of this type are this age have any object activity. If an object has not been used for a long time there is a good chance it has been leaked.
- **Stale Objects** › How many objects of this type are this age are considered to be stale.

  Stale objects may be candidates for memory leak inspection. These objects are based upon simple hueristics guided by some parameters you can set using the settings dialog. The hueristics allow you to tailor the stale object detection for your application because a good strategy for one application may not be very useful for another application (compare a word processor to a server process - they both have different characteristics in memory usage).

Object Activity detection and Stale Object detection is controlled by the Object Activity Settings.

Watching the Ages display as your application runs can give you a very good instantaneous view of which object types may be candidates for further inspection for memory leaks.

Note that the Ages display is CPU intensive. If you do not need to view the Ages display you are advised to view a different display whilst collecting data to maximise the performance of Memory Validator.

The ages data columns

The first column is the object type.

All other columns represent each object age. The ordering of the ages columns is controlled by the Display Settings.

The header columns display a total for the column underneath the column name.

The value in each column is determined by which tab of data is being displayed.

Colours used in the display

Each row is coloured according to whether the object has:


an increasing count for the number of live objects of the size


a decreasing count


a static count


a zero count - i.e. where all allocated objects of the size have been freed

The importance of each value within the column is highlighted with a percentage bar:

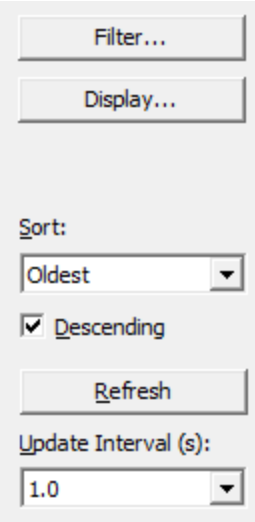

the object size with the maximum value in a given column (not shown for all columns)


relative contribution of the value in each column

➔ See also the Data Highlighting settings dialog to customise the first two colours.

Ages tab options

The following options are available:



Sorting columns

Sorted columns are highlighted yellow. Just click on the column header to change the sorting column or it's sort direction order.

As well as the column headers we also provide explicit sorting controls to provide an extra option that isn't available from the column headers ("Type" and "Inactive").

- **Sort** › choose the generation to sort
- **Descending** › choose the sort direction

Updating the display

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds - or never!
- **Refresh** › updates the display - as does the button on the Tools menu and toolbar

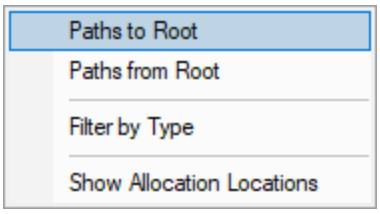
  With an update interval set to Never, you'll need to use this Refresh button to update the display.
- **Display...** › display the Ages Settings dialog.

Filter settings

- **Filter...** › shows the Ages Filters settings dialog to edit the filters.

Ages view popup menu

The following popup menu provides options for filtering and examining data in more detail.



Menu option: Paths to Root, Paths from Root

The following options are only active for .Net object types. They are disabled for native object types.

- **Paths to Root** › For all live objects of this type display all the paths from the object to the most recent heap dump roots.
- **Paths from Root** › For all live objects of this type display all the paths most recent heap dump roots to the live objects.

Menu option: Filter by Function

The following options allow you to remove data from the display using filters.

- **Filter by Type** › creates a filter with the selected object type.

By default filters prevent data from being displayed. You can change this using the Generation Filters.

Filters can be edited using the **Filter...** button to display the Generation Filters.

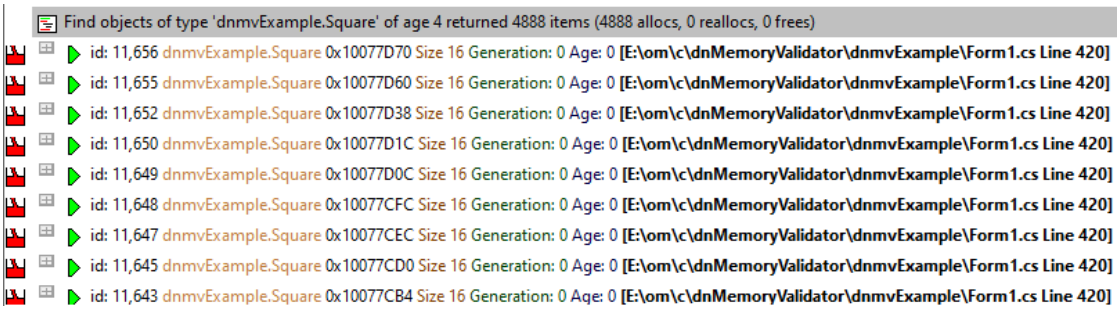
Menu option: Showing locations - drilling down into the data

The following option opens the Analysis Tab, adding a callstack for every allocation or deallocation of the selected object size.

This enables a deeper inspection of where and how objects of this size are allocated or freed.

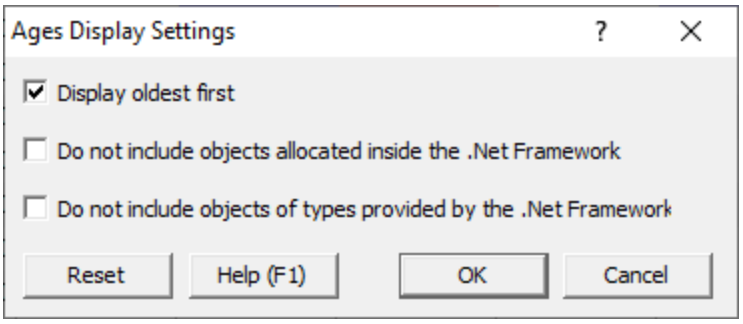
- **Show Allocation Ages** › shows allocations only

For example, showing allocations for the following row in the Ages tab will show the callstacks for 2634 allocations of **System.Collections.DictionaryEntry** in the Analysis tab below:



3.7.4.5.1 Ages Settings Dialog

The Ages Settings dialog allows you to control how the display looks on the Ages view.



- **Display oldest first** › change how the ages are displayed.

  Oldest first

Type	Age 20	Age 19	Age 18	Age 17	Age 16	Age 15	Age 14	Age 13	Age 12	Age 11	Age 10	Age 9	Age 1
	21,257	272	11	70	22	50	31	10	74	147	41	30	32
System.String	4,694	24	0	0	0	0	0	0	0	9	13	0	7
System.RuntimeType	949	10	0	0	0	0	0	0	0	8	0	0	2
System.Windows.FrameworkProp...	786	0	0	0	0	0	0	0	0	0	0	0	0
FromNameKey	758	9	0	0	0	0	0	0	0	9	0	0	0

  Youngest first

Type	Age 1	Age 9	Age 10	Age 11	Age 12	Age 13	Age 14	Age 15	Age 16	Age 17	Age 18	Age 19	Age 20
	32	30	41	147	74	10	31	50	22	70	11	272	21,257
System.String	7	0	13	9	0	0	0	0	0	0	0	24	4,694
System.RuntimeType	2	0	0	8	0	0	0	0	0	0	0	10	949
System.Windows.FrameworkProp...	0	0	0	0	0	0	0	0	0	0	0	0	786
FromNameKey	0	0	0	9	0	0	0	0	0	0	0	9	758

Filtering

Some simple filtering is provided which may be useful in reducing the scope of the types you are examining to just those types defined by your program.

- **Do not include objects allocated inside the .Net framework** › ignore anything the .Net framework allocates.
- **Do not include objects of types provided by the .Net framework** › ignore any types the .Net framework provides.

3.7.5 .Net

The .Net tab provides three .Net specific tools for identifying where .Net memory is allocated and which locations may be leaking memory.

Click on an item in the picture below to find out more about each of the tabbed windows, or use the list further below:

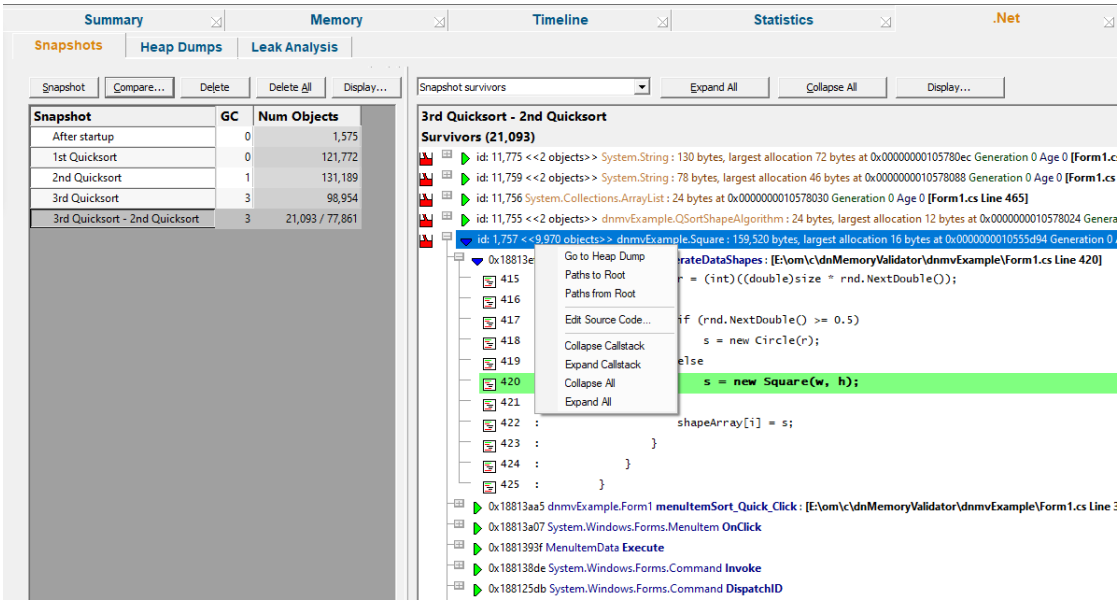
3.7.5.1 .Net Snapshots

Snapshots are a useful way of identifying memory (or handles) that are loitering in memory for longer than you expected.

These may be leaked, or they may be getting garbage collected later than you expected.

The **Snapshots** view allows you to create, compare, display and delete snapshots.

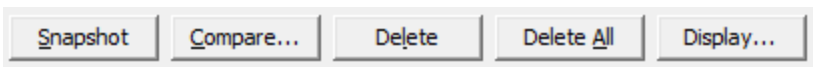
Click a part of the image below to jump straight to the help for that area.



The view is split into two halves. The left hand pane manages the creation, comparison, and deletion of snapshots. The right hand pane displays a snapshot or snapshot comparison.

Managing Snapshots

The controls for managing snapshots are listed at the top of the pane.



- **Snapshot** › Create a snapshot. If the snapshot is not automatically named, you will be prompted to name the snapshot
- **Compare...** › Create a snapshot comparison. If the snapshot comparison is not automatically named, you will be prompted to name the snapshot comparison
- **Delete** › Delete the selected snapshot
- **Delete All** › Delete all snapshots
- **Display...** › Edit settings related to the naming of snapshots and snapshot comparisons.

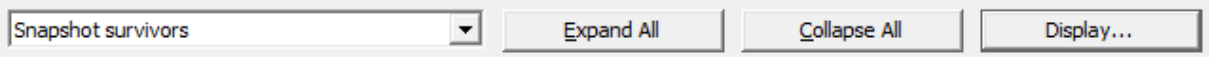
The grid below the controls lists each snapshot or comparison, the garbage collection they are related to and how many objects are represented by the snapshot.

Snapshot	GC	Num Objects
After startup	0	1,575
1st Quicksort	0	121,772
2nd Quicksort	1	131,189
3rd Quicksort	3	98,954
3rd Quicksort - 2nd Quicksort	3	21,093 / 77,861

Selecting a snapshot or snapshot comparison in the grid on the left hand pane will display the snapshot or snapshot comparison on the right hand side.

Displaying Snapshots

The controls for displaying snapshots are at the top of the right hand pane.



- **Snapshot dropdown** › Choose which group of objects to display

  For snapshots there is only one option: **Snapshot objects**.

  For snapshot comparisons there are two options: **Snapshot new objects** and **Snapshot survivors**.

  - **Snapshot objects** › Shows the objects in the snapshot
  - **Snapshot new objects** › Shows the objects in the snapshot comparison that are in the second snapshot but not in the first snapshot.
  - **Snapshot survivors** › Shows the objects in the snapshot comparison that are present in the first snapshot and the second snapshot.

- **Expand All** › Expand all callstacks

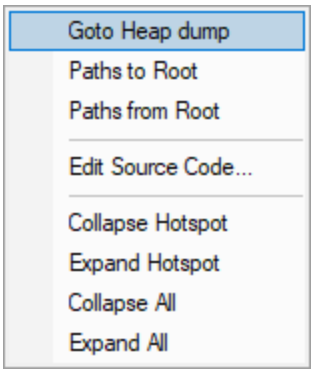
- **Collapse All** › Collapse all callstacks
- **Display...** › Edit settings related to the displaying snapshots and snapshot comparisons.

The top row of the grid will show the snapshot name (or snapshot comparison name). The row below will indicate what type of objects are present (Objects, New objects, Survivors) and how many.

The remainder of the grid will display callstacks for each object (or group of objects with a common callstack) that are in the group of objects being displayed.

Snapshots view popup menu

The following popup menu provides options for filtering and examining data in more detail.



Menu option: Go to Heap Dump, Paths to Root, Paths from Root

The following options are only active for .Net object types. They are disabled for native object types.

- **Go to Heap Dump** › Find the heap dump entries for the selected objects.
- **Paths to Root** › For the selected objects display all the paths from the object to the most recent heap dump roots with the Paths to Root dialog.
- **Paths from Root** › For the selected objects display all the paths most recent heap dump roots to the live objects with the Paths to Root dialog.

Menu option: editing source code

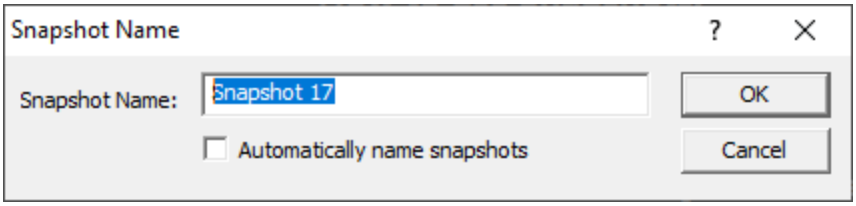
- **Edit Source Code...** › opens the default or preferred editor to edit the source code

Menu options: Collapse Hotspot, Expand Hotspot, Collapse All, Expand All

- **Collapse Hotspot** or **Expand Hotspot** › simply shows and hides data item information, the same as using the ⊞ or ⊟ buttons
- **Collapse All** or **Expand All** › collapses or expands all callstack entries

3.7.5.1.1 .Net Snapshot Creation Dialog

The Snapshot Creation dialog allows you to specify a name for a newly created snapshot.



The dialog is initially displayed with a suggested default name for the snapshot (as shown above).

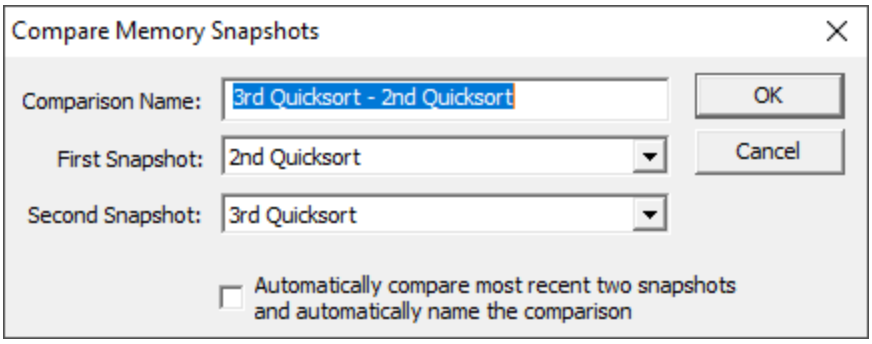
If you would prefer a more informative name, type your own snapshot name.

- **Automatically name snapshots** › All future snapshots will be named automatically. You won't see this dialog again.

  If you change your mind and wish to start naming your snapshots again, go to the Snapshots Display Settings dialog to change the setting.

3.7.5.1.2 .Net Snapshot Comparison Dialog

The Snapshot Comparison dialog allows you to specify a name for a newly created snapshot.



The dialog is initially displayed with a suggested default name for the snapshot comparison (as shown above).

If you would prefer a more informative name, type your own snapshot name.

- **Automatically compare most recent...** › All future snapshots will be named automatically and compared automatically.

  The comparison will be between the penultimate snapshot and the most recent snapshot. You won't see this dialog again.

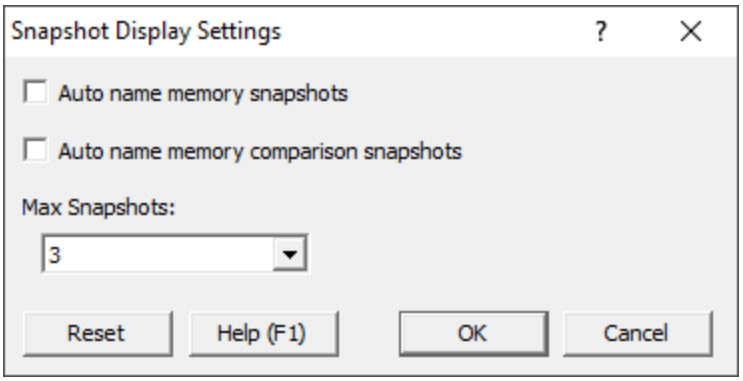
  If you change your mind and wish to start naming your snapshot comparisons again, go to the Snapshots Display Settings dialog to change the setting.

The first and second snapshots default to the penultimate snapshot and the most recent snapshot.

- **First snapshot** › The snapshot to be compared against.
- **Second snapshot** › The snapshot that you think may contain leaked objects.

3.7.5.1.3 .Net Snapshot Display Settings Dialog

The Snapshot Display Settings dialog allows you to manage automatic naming and the maximum number of snapshots.



Automatic Naming

- **Auto name memory snapshots** › All snapshots will be automatically named.
- **Auto name memory snapshot comparisons** › All future snapshot comparisons will be automatically named and automatically compared.

Snapshots

Snapshots can contain a lot of objects. If you create too many of these Memory Validator might run of memory.

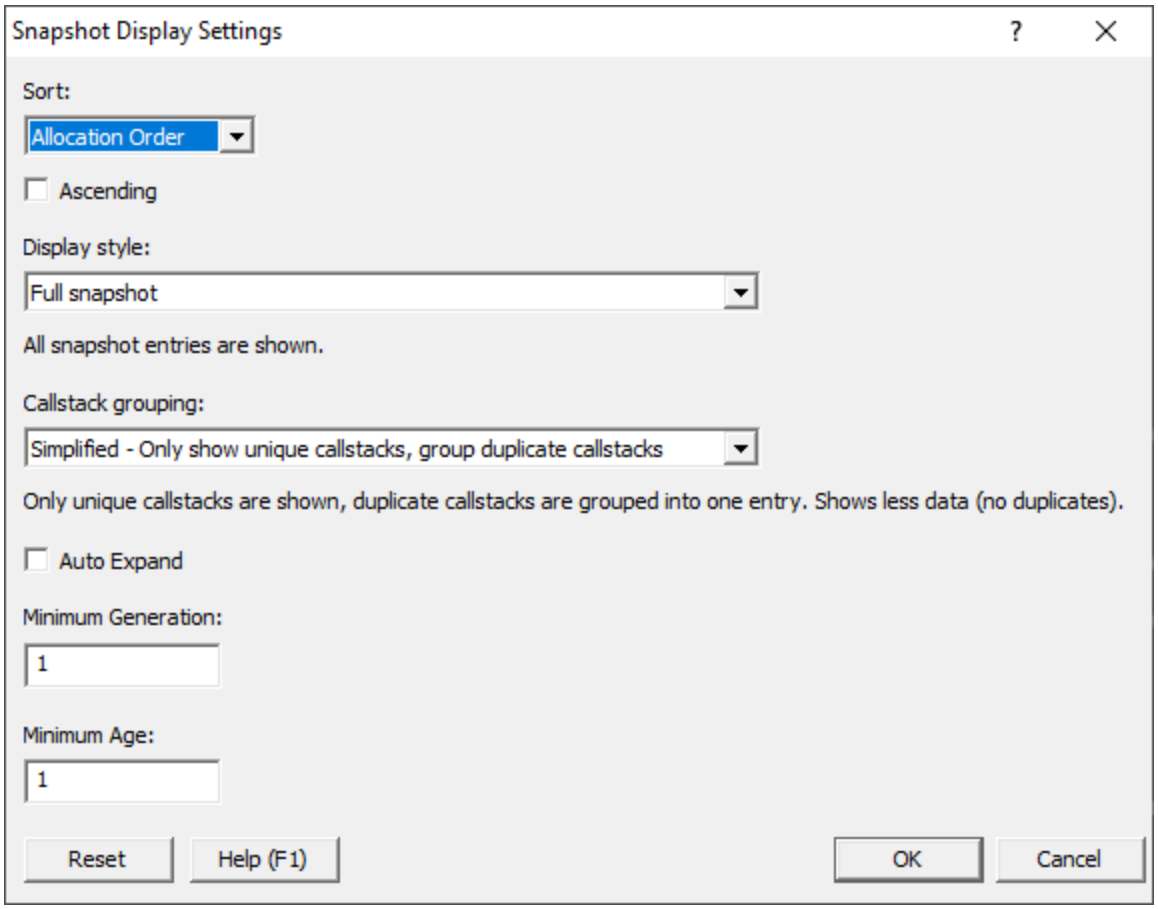
To keep this manageable we allow you to set a maximum number of snapshots.

When the snapshot limit is exceeded the oldest snapshot is automatically deleted.

- **Max Snapshots** › The maximum number of snapshots to keep.

3.7.5.1.4 .Net Snapshot Callstack Display Settings Dialog

The Snapshot Callstack Display settings dialog allows you to control how data is displayed on the snapshot callstacks display.



Display Ordering

The callstacks in a snapshot can be sorted prior to display.

- **Allocation Order** › the order objects were allocated in
- **Size** › by the size of the allocation
- **Object Type** › alphabetical comparison of object type
- **Filename** › alphabetical comparison of filename
- **Namespace** › alphabetical comparison of namespace
- **Address** › the object id of each object

Making the data easier to understand

Some options affect how much data is shown. Displaying less data can sometimes make for a more readable display.

- **Display Style** › determine which objects to display.
  - **Full** › information about every allocation and error is displayed (unless filtered)
  - **Simplified - your source code at root** › Only traces that have a callstack with your source code at the top of the callstack are displayed
  - **Simplified - your source code not at root** › Only traces that have a callstack with your source code in the callstack (except for the top position) are displayed
  - **Simplified - your source code anywhere** › Only traces that have a callstack with your source code anywhere in the callstack are displayed
  - **Simplified - compiler vendor source code at root** › Only traces that have a callstack with your compiler vendor source code at the top of the callstack are displayed
  - **Simplified - compiler vendor source code not at root** › Only traces that have a callstack with your compiler vendor source code in the callstack (except for the top position) are displayed
  - **Simplified - compiler vendor source code anywhere** › Only traces that have a callstack with your compiler vendor source code anywhere in the callstack are displayed
  - **Simplified - no source code** › Only traces that have a callstack with no source code are displayed
- **Callstack grouping** › display all callstacks or just unique callstacks
  - **Full** › every callstack is shown
  - **Simplified - Only show unique callstacks** › Traces that share the same callstack are displayed once. A summary is shown indicating the number of allocations, how many bytes in those allocations and the size of the largest allocation.
- **Minimum Generation** › only display objects that were created on or after the specified generation. This allows you to ignore objects created at startup, or before a particular point in program execution
- **Minimum Age** › only display objects that have survived at least the specified number of garbage collections. This allows you to ignore objects that are most likely still in use

Callstacks can be displayed in collapsed or expanded form:

- **Auto Expand** › every callstack is displayed expanded so that you can see the callstack as well as the summary

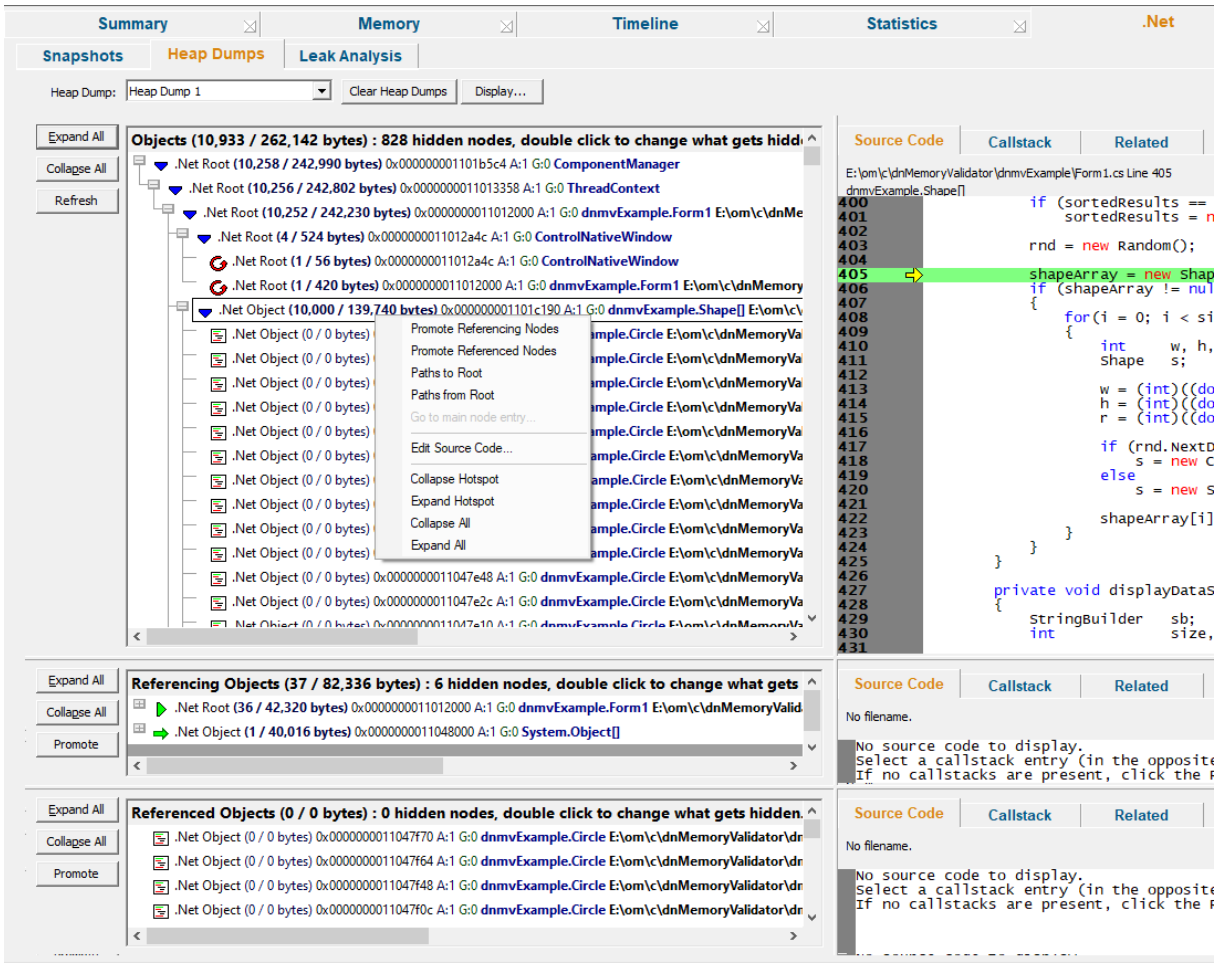
3.7.5.2 .Net Heap Dumps

Heap dumps allow you to understand the relationship between each .Net memory/handle allocation in your program.

Heap dumps can be requested manually via the Heap Dump icon or the .Net Tools menu.

Heap dumps can also be requested each time a garbage collection occurs. This is controlled by the settings dialog.

Click a part of the image below to jump straight to the help for that area.



Heap dump selection

Some simple heap dump management is provided by controls at the top of the Heap dump view.

- **Heap Dump** › Select the heap dump to display. Click **Refresh** to the left of the top panel to display the heap dump.
- **Clear Heap Dumps** › Deletes all heap dumps and removes all data from the display

  After clearing heap dumps any attempts to fetch a new heap dump will fail until enough objects have been allocated to allow a garbage collection to run.
- **Display...** › Displays the Heap Dump Settings dialog to control the display of heap dumps

How the display works

The view is split into two parts horizontally and three parts vertically.

Horizontal

Horizontally the split is between the graph on the left and source code, callstacks and related objects on the right.

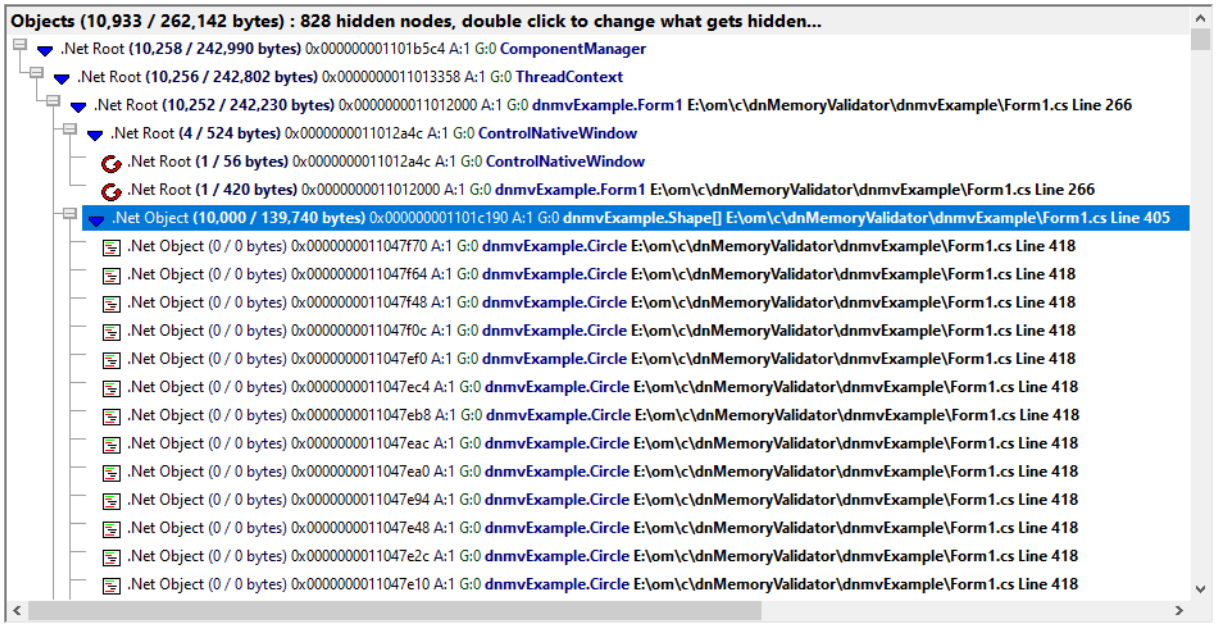
Vertical

Vertically the split is between the data we're looking, objects referencing what we're looking at, referencing objects and referenced objects. The referencing objects and referenced objects are only populated when an object is selected in the top heap dump graph. As you change which object is selected the two lower graphs update, and the information displayed to the right also updates - the source code, the allocation callstack and any related objects, these are all updated.

In each vertical section the horizontal split is the same; heap dump graph on the left, source code, callstack, related objects on the right.

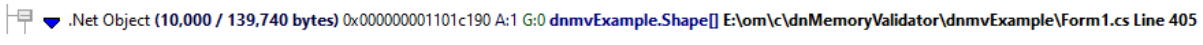
How the data relates to each other

Let's take a tour of the user interface by examining each panel. First we need some data to display. We've already performed a heap dump (or a garbage collection has created one for us). Click **Refresh** to cause the user interface to display the selected heap dump. We start the tour in the top left.



Here we can see various objects, some of which are heap dump roots, and an object with 10,000 referenced objects.

Each object is displayed indicating root, or non-root, how many child objects and their size, object id, age, the generation it was created, it's type and allocation filename and line number. For example:

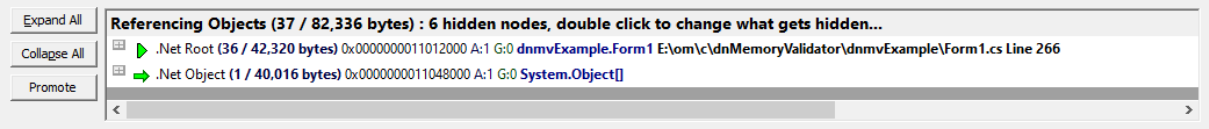


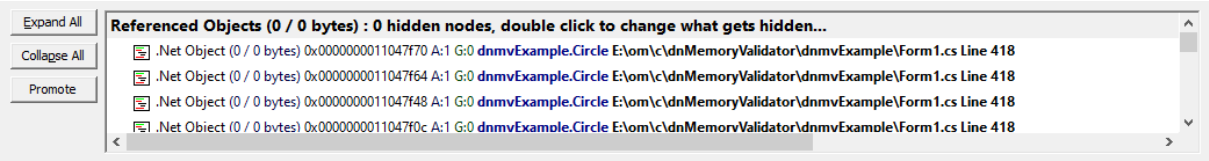
Non-root
Referencing 10,000 objects of size 139.740 bytes
Object id 0x000000001101c190
Age 1
Generation created: 0
Type **shape** (in the module **dnmvExample**)
Filename **e:\om\c\dnMemoryValidator\dnmvExample\Form1.cs** and line **405**

At the left of the panel are three buttons

- **Expand All** › Expands all the graph nodes
- **Collapse All** › Collapses all the graph nodes
- **Refresh** › Displays the selected heap dump. If the heap dump is already displayed it will be redisplayed in it's default state.

In the middle left and bottom left panels we can see the objects referencing the selected object, and the objects referenced by the selected object.

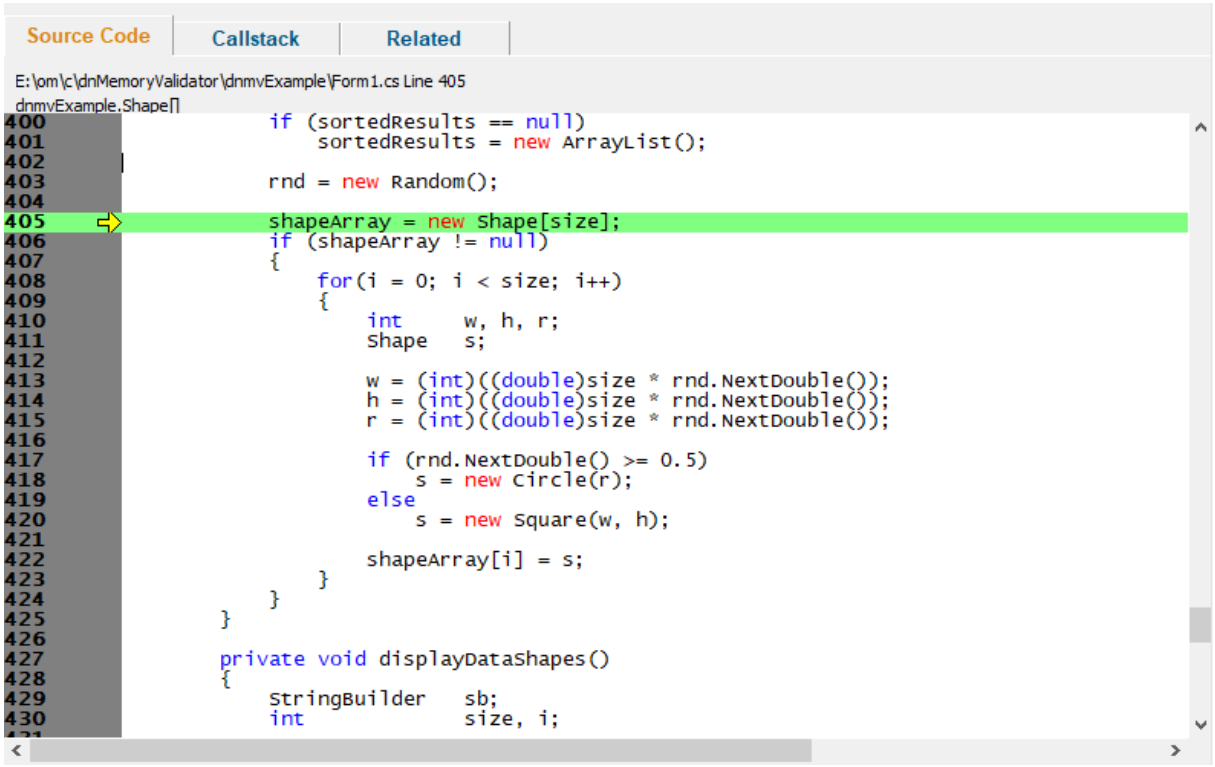


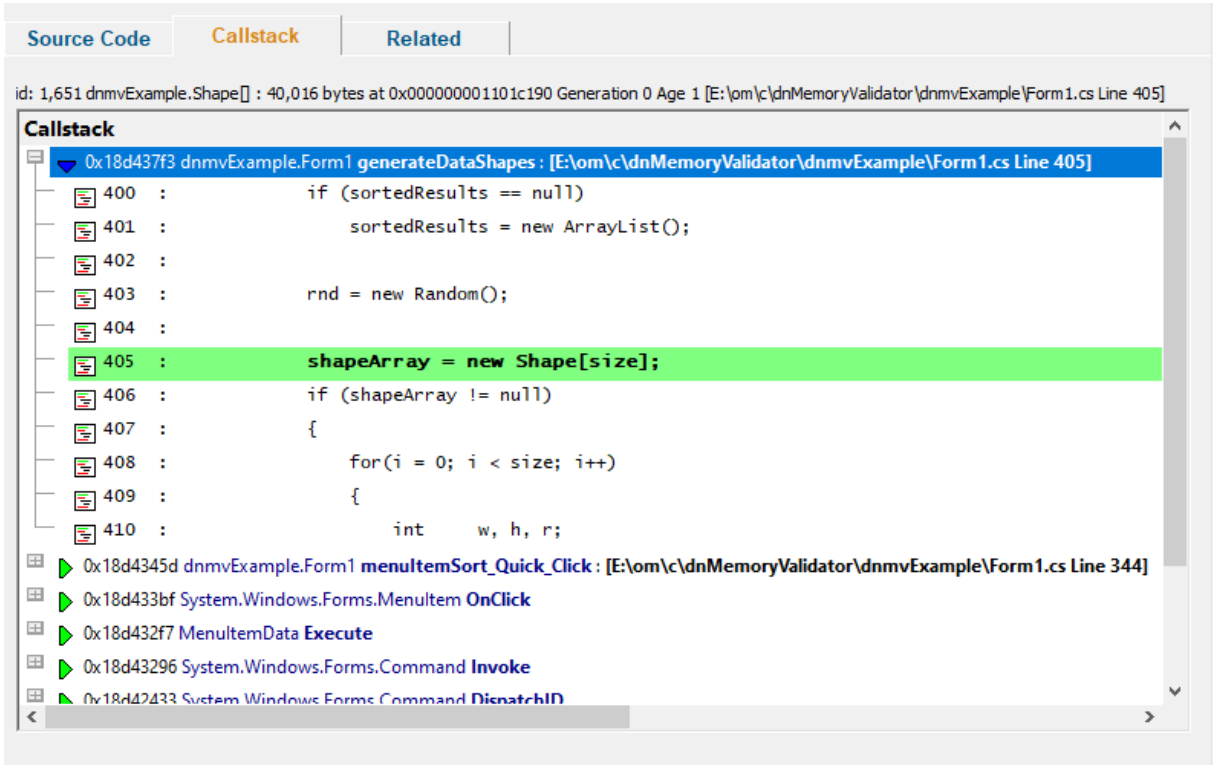


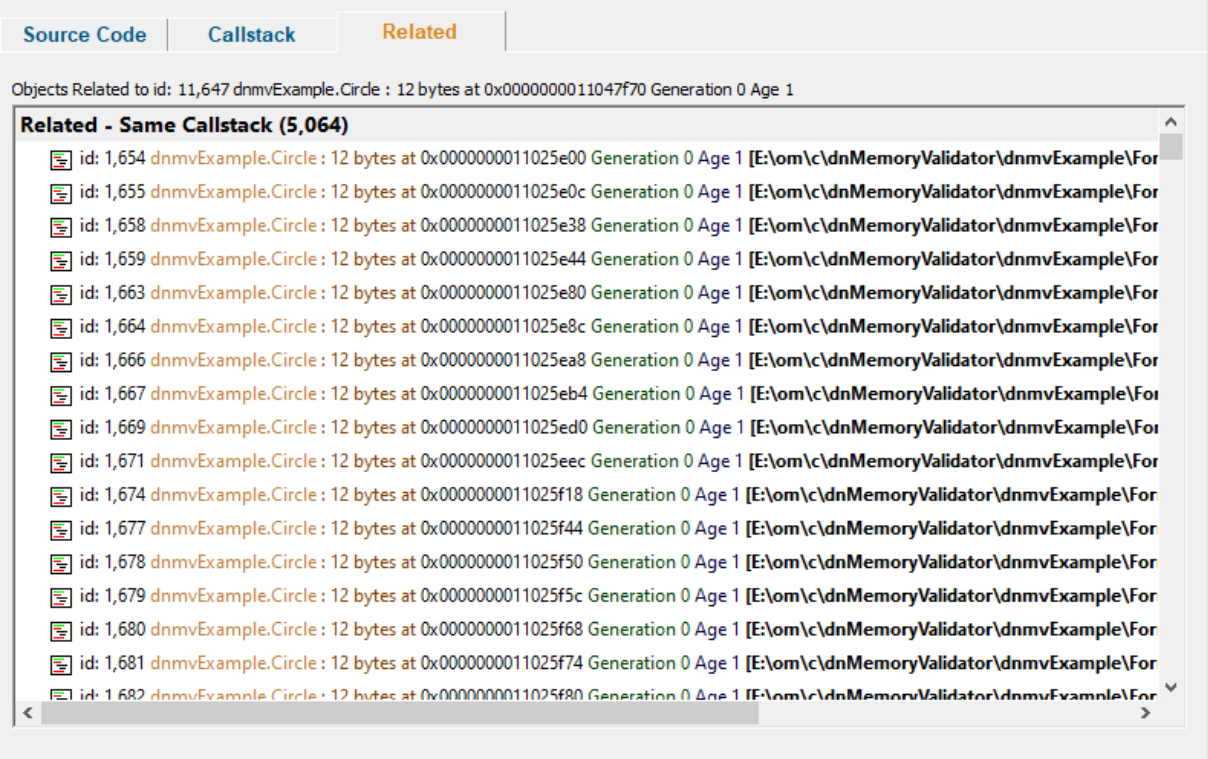
These panels behave in the same way as the top panel, displaying related information on the right. There is one difference - the **Refresh** button has been replaced by a **Promote** button.

- **Promote** › Move all nodes (or just the selected node to the top panel) for exploration.

The panels to the right show source code, the allocation callstack and any related nodes (allocated with the same callstack). These panels are duplicated for each of the three panels on the left.

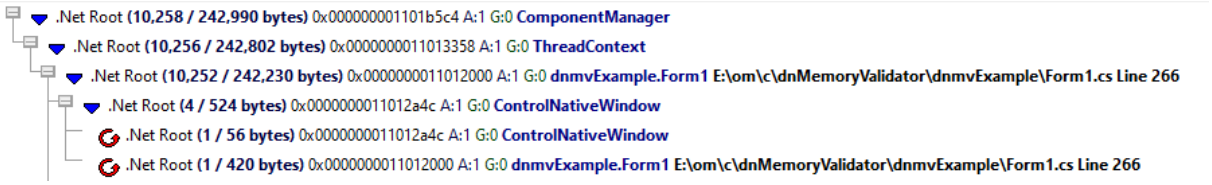






Heap dump recursion

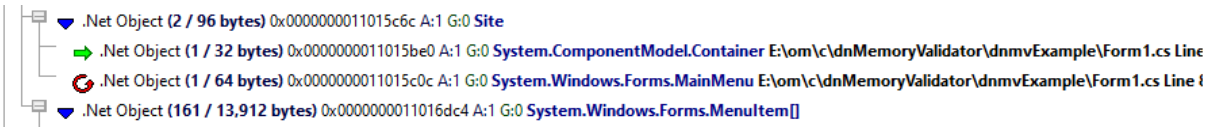
A heap dump is represented as a graph showing which objects have a reference to other objects. Some objects also reference themselves, either directly, or indirectly via intermediate objects, forming a recursive loop.



In the above image we can see 0x0000000010352a4c points to itself, and 0x0000000010352000 points to itself indirectly via 0x0000000010352a4c.

Heap dump links

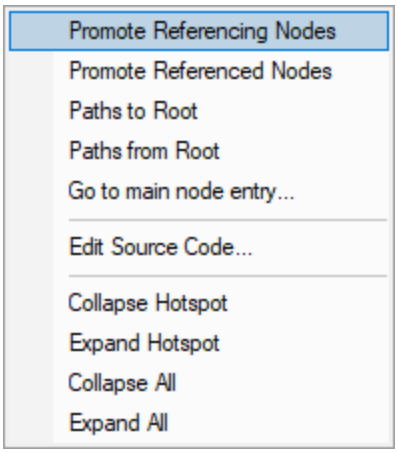
A heap dump has many roots, each with it's own graph of objects beneath it. Because multiple roots may link to the same subgraph we've decided to represent each subgraph only once and show links to that subgraph as an arrow.



In the above image we can see a link to a subgraph. To visit that subgraph right click on the node to show the context menu then choose Go to main node entry....

Heap dumps view popup menu

The following popup menu provides options for filtering and examining data in more detail.



Menu option: Promote, Paths to Root, Paths from Root, Go to main node

The following options are only active for .Net object types. They are disabled for native object types.

- **Promote Referencing Nodes** › Promote the nodes referencing the selected object to the top left panel.
- **Promote Referenced Nodes** › Promote the nodes referenced by the selected object to the top left panel.
- **Paths to Root** › For the selected objects display all the paths from the object to the most recent heap dump roots with the Paths to Root dialog.
- **Paths from Root** › For the selected objects display all the paths most recent heap dump roots to the live objects with the Paths to Root dialog.
- **Go to main node entry...** › Find the heap dump entry for the selected object.

Menu option: editing source code

- **Edit Source Code...** › opens the default or preferred editor to edit the source code

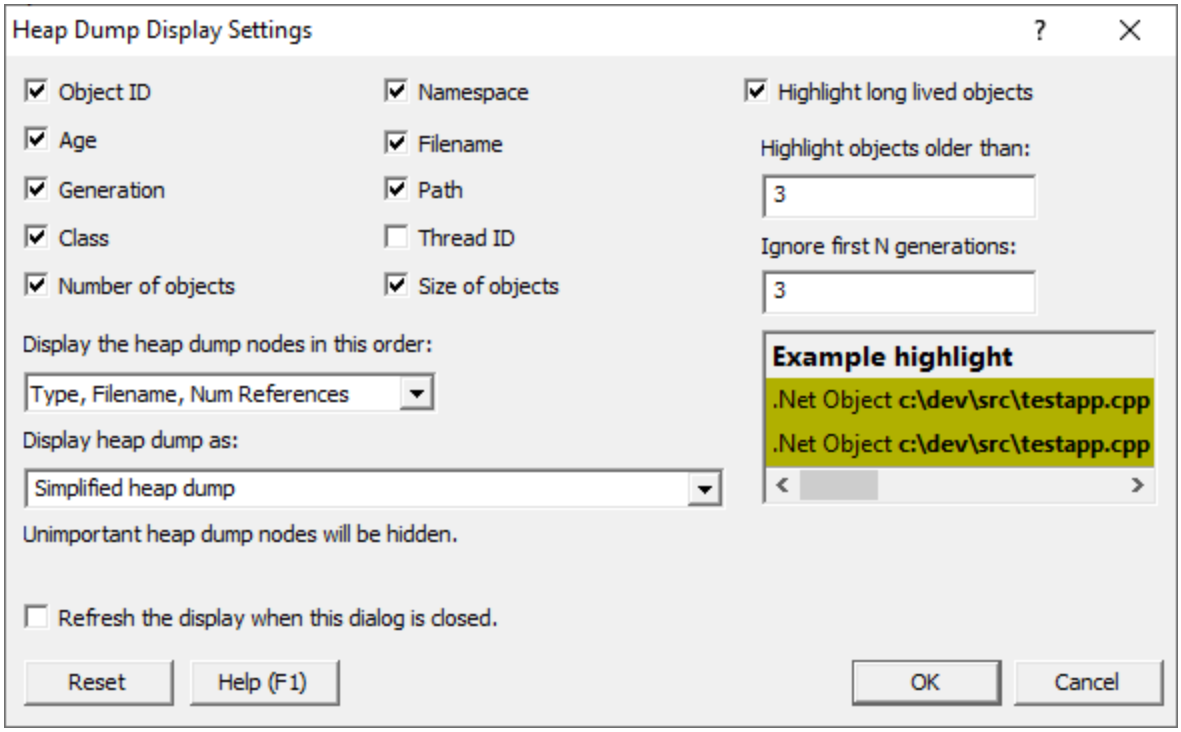
Menu options: Collapse Hotspot, Expand Hotspot, Collapse All, Expand All

- **Collapse Hotspot** or **Expand Hotspot** › simply shows and hides data item information, the same as using the ⊞ or ⊟ buttons
- **Collapse All** or **Expand All** › collapses or expands all callstack entries

3.7.5.2.1 .Net Heap Dump Display Settings Dialog

The Heap Dump Display Settings dialog allows you to manage how heap dumps are displayed, both visually and the content of the heap dump.

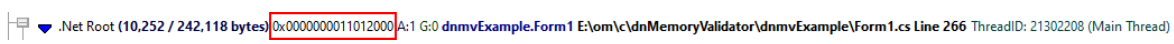
Click a part of the image below to jump straight to the help for that area.



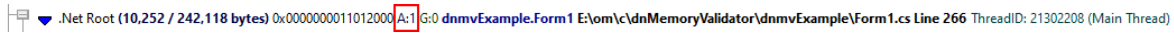
Data Display

Various attributes related to the each node can be displayed:

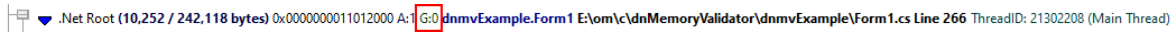
- **Object ID** › Object ID.



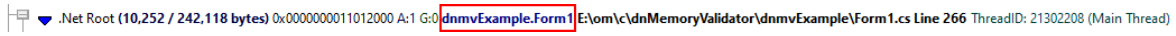
- **Age** › How many garbage collections since this object was created.



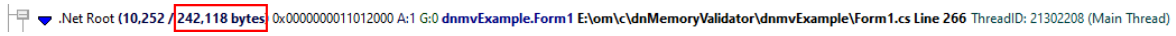
- **Generation** › The generation this object was created in.



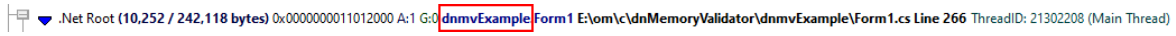
- **Class** › The object class.



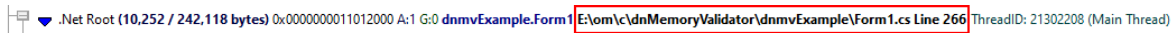
- **Number of objects** › The number of objects this object references.



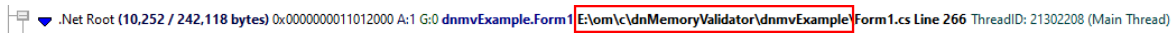
- **Namespace** › The object namespace.



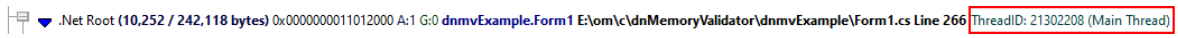
- **Filename** › The source code filename and line number for the allocation location.



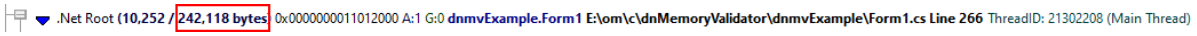
- **Path** › The source code path for the allocation location.



- **Thread ID** › The thread id for the allocation location.



- **Size of objects** › The size in bytes of the number of objects referenced by this object.



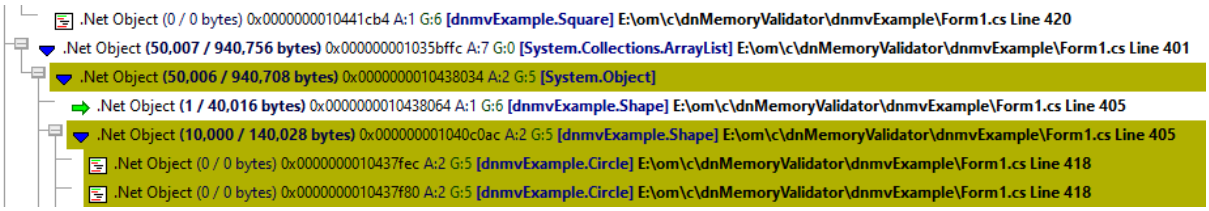
Visual Highlighting

To aid identifying long lived objects that may be of interest when trying to find loitering objects that may be leaked you can highlight objects older than a specified age, and created after a specified generation. This allows you to ignore objects created during the startup phase of the application. A example of highlighting is shown below the highlighting controls.

- **Highlight long lived objects** › Turn visual highlighting of loitering objects on / off.
- **Highlight objects older than** › How old do you think an object needs to be for it get your attention?

- **Ignore first N generations** › Enter the number of garbage collections it takes for your application to startup.

Highlighted objects look like this:



Data ordering

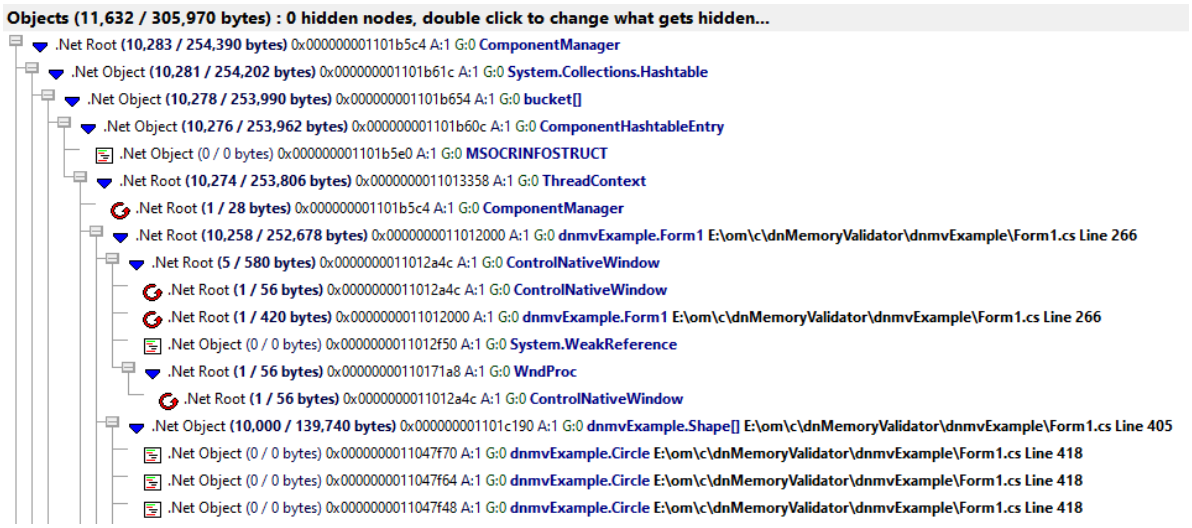
When choosing how to order the heap dump there are a few sorting options:

- **Num References, Type, Filename** › Sort by Num References, then by Type, then by Filename.
- **Type, Filename, Num References** › Sort by Type, then by Filename, then by Num References. This is the default option.
- **Filename, Type, Num References** › Sort by Filename, then by Type, then by Num References.

Making the data easier to understand

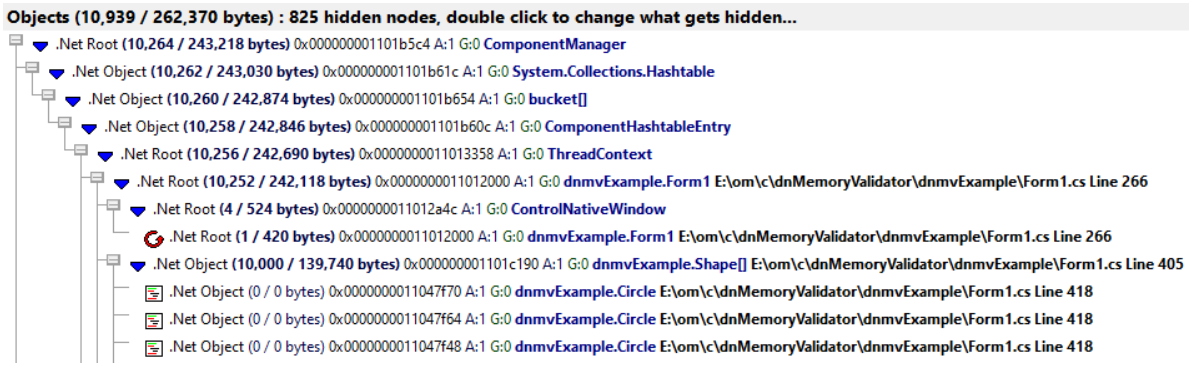
Heap dumps can be very complicated to understand. The complexity is increased when there are objects present that don't really add to your understanding of the heap dump, but which are present all the same. We've provided some options to allow you to tailor the level of complexity to suit the task.

- **Full heap dump** › Everything in the heap dump is shown.

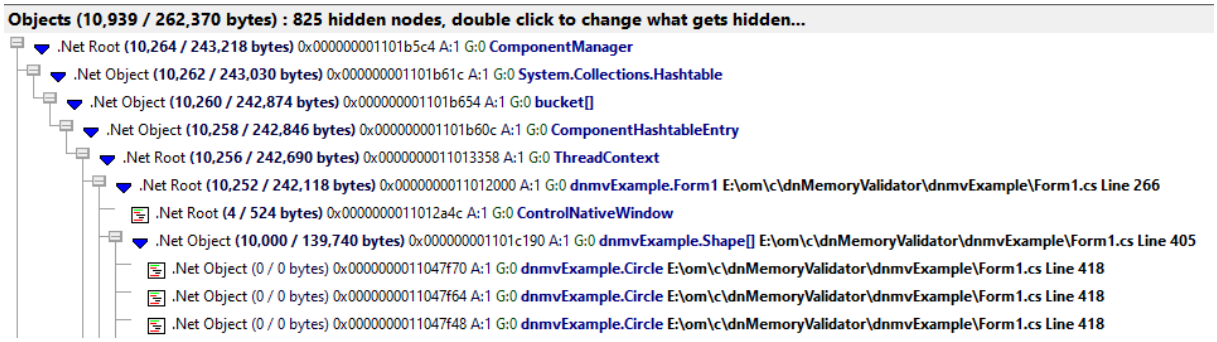


- **Simplified heap dump** › Unimportant nodes in the heap dump are not shown. This is the default option.

  What is classed as unimportant? Any node that references no other nodes and has no source code (CLR/.Net Frameworkd source code is not included) and any node that has reference nodes that are classed as unimportant.



- **Over simplified heap dump** › Unimportant nodes in the heap dump are not shown. Additionally some recursive nodes and some node links are not shown.



The simplified heap dump is easier to display because less data is shown. It includes all recursive nodes and links to other nodes.

The over simplified heap dump has even less data and for many tasks may be easier to understand. But because the recursive nodes and links to other nodes are absent you may get an incorrect sense of the layout of the heap dump.

You need to use the over simplified heap dump with caution.

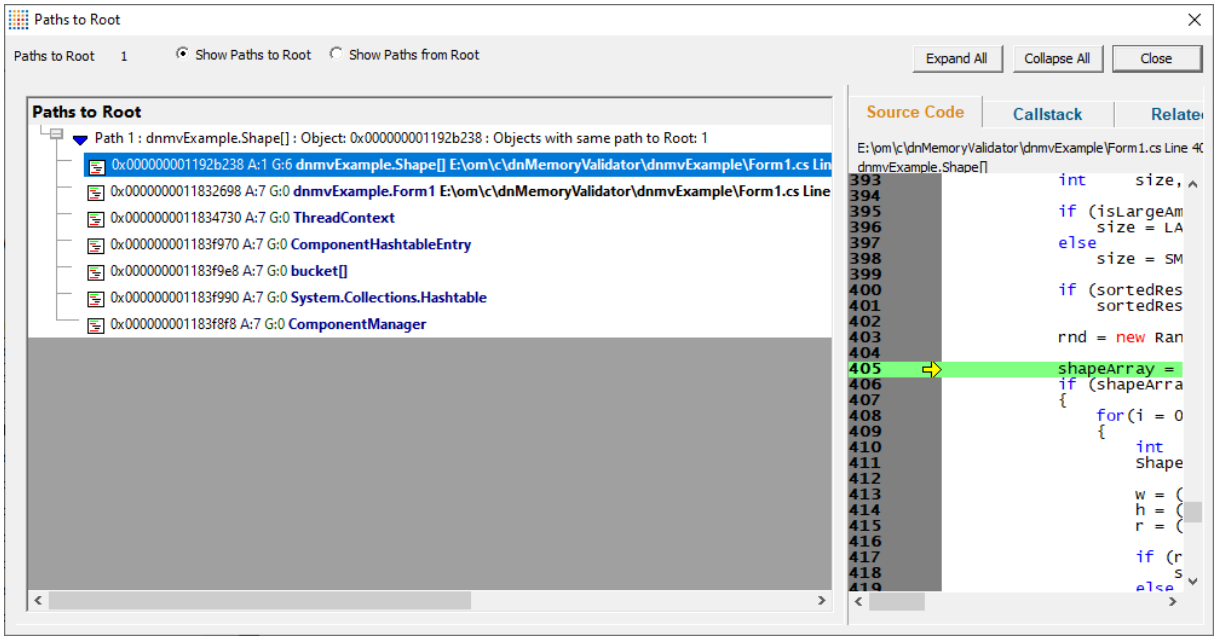
Auto display

- **Refresh the display when the dialog is closed** › The display will be completely refreshed. Do not choose this option if you only wanted to change the visual highlighting or data display.

3.7.5.2.2 .Net Path to Root

The Path to Root dialog shows the path from the selected object to it's root nodes.

The dialog can also show the paths from it's root nodes to the selected object (the Path from Root).



The data displayed in the right hand window is the source code, for selected entry, the callstack and any related objects.

This is the same information that is displayed in the right hand part of the Heap Dumps view. A more detailed description is provided in that help topic.

- **Show Paths to Root** › Displays the paths from the selected object to the heap dump roots.
- **Show Paths from Root** › Displays the paths from the heap dump roots to the selected object.
- **Expand All** › Expands all paths to/from roots.
- **Collapse All** › Collapses all path to/from roots.

3.7.5.3 .Net Leak Analysis

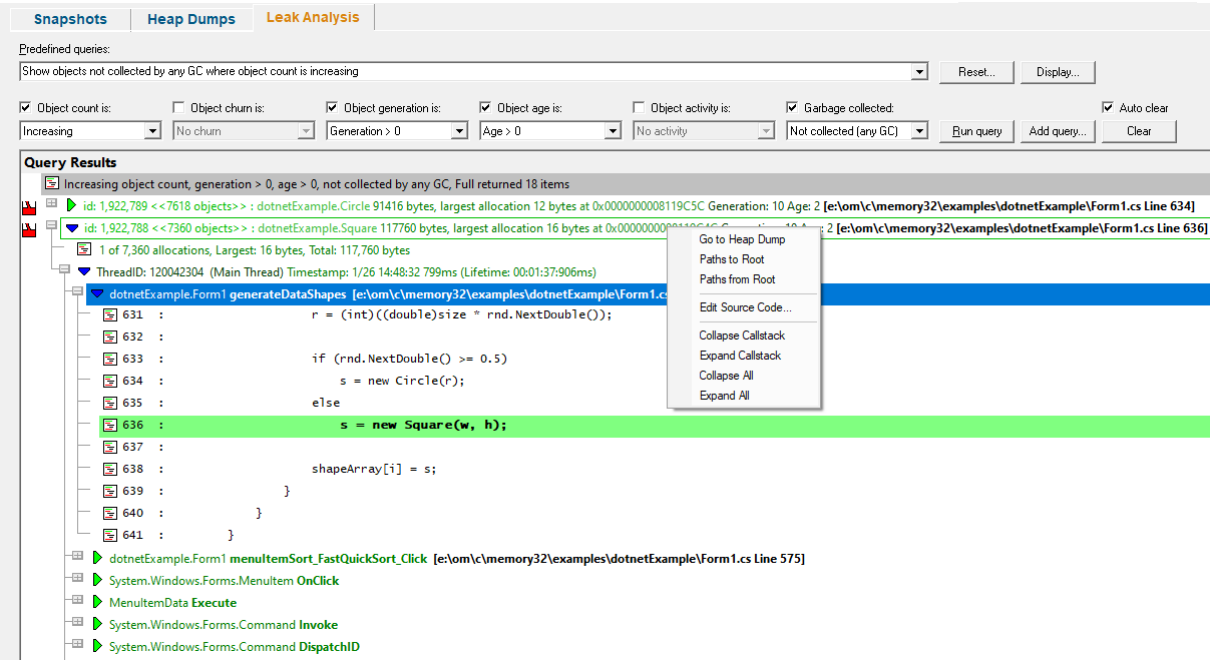
Garbage collected programs, in theory, never leak memory. In practice that isn't the case, which is probably why you're using Memory Validator.

If references to objects are not reset to `null` the garbage collector will not collect the object, even if it will never be used again by the program. Such conditions are hard to detect even with tools like Snapshots and Heap Dumps.

To aid in this task we've created a dedicated .Net leak analysis query facility where you can use predefined queries or design your own query.

Each query is formed from five factors and the display of the results allows for a range of filtering options.

Click a part of the image below to jump straight to the help for that area.



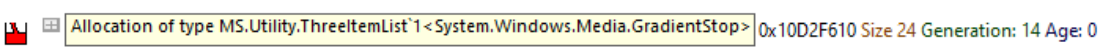
Displaying Data

Data from each query is displayed in the tree control.

Each query is prefixed with a line that describes the query that was run.

The query results are displayed after the description.

Each line has an icon at the left, indicating its type, and has an explanatory tooltip:

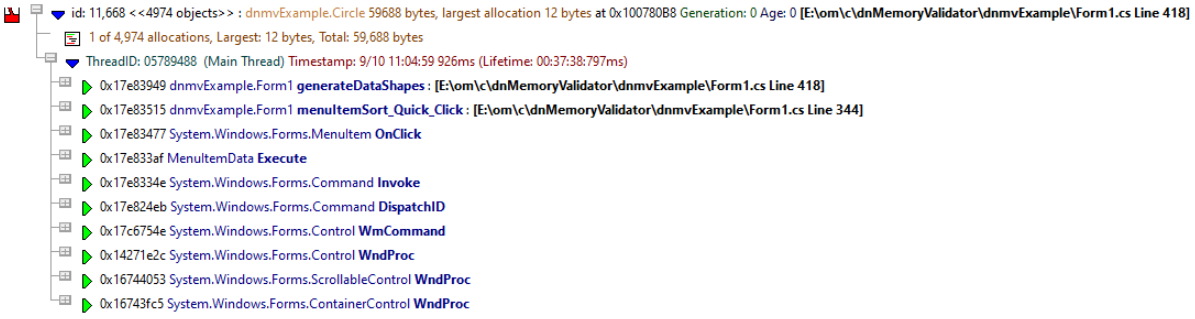


The text on each line indicates:

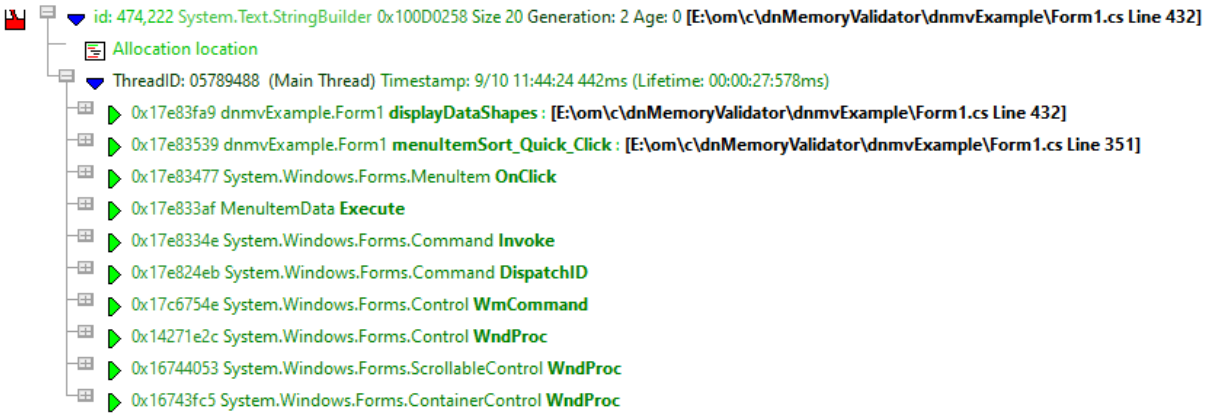
- datatype (if known)
- size
- allocation address/handle value
- source file and line number (if available) where the allocation occurred
- optional event sequence id at the beginning of the line

There are two types of callstack representation.

The first callstack representation is that of an object that has been allocated on a callstack that has never had any objects allocated on it that have been garbage collected.



The second callstack representation is that of an object that has been allocated on a callstack that has had objects allocated on it garbage collected.



This colouring is controlled by the **Enhanced callstack colouring** option on the Callstack settings of the settings dialog.

Display Style

Before displaying the data from a query the data needs to meet the criteria specified by the display style. This is a simple filter based on the availability of source code, and where that source code is on the callstack. The reason for this is that allocations (causing memory leaks) in your source code are fixable, whereas allocations in 3rd party code, or allocations that don't have source code are not fixable because you don't have access to the source code. This doesn't mean you should ignore such allocations as they may be caused by a side effect of one of your allocations or a reference from an object you control, but the ability to filter the data this way can greatly simplify the scope of the query results you are examining and lead to greater insight.

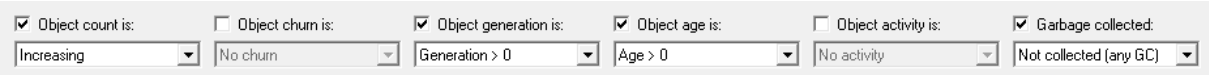
In addition to the display style, data is also displayed according to a callstack group critieria: display all callstacks, or group similar callstacks together.

Both of these styles can be edited using the .Net Leak Analysis display settings dialog.

- **Display...** › displays the .Net Leak Analysis display settings dialog

Defining a Query

A query is defined by five criteria which need to be matched to put each object into the query results.



To use a criteria in a query enable the appropriate check box.

Object count

- **Increasing** › object count increases each generation
- **Increasing or stable** › object count increases or remains stable each generation
- **Decreasing** › object count decreases each generation
- **Decreasing or stable** › object count decreases or remains stable each generation

Object churn

- **No Churn** › There is no object churn
- **N** › Object has no object churn for N generations

  [where N is list of numbers representing each generation except the current generation]

Object generation

- **Generation > 0** › Any object in a generation greater than 0
- **N** › Object generation is greater than the specified generation

  [where N is list of numbers representing each generation except the current generation]

Object age

- **Age > 0** › Any object older than 0
- **N** › Object age is greater than the specified age

  [where N is list of numbers representing each generation except the current generation]

Object activity

- **No activity** › No object activity
- **N** › Object has no activity for N generations

  [where N is list of numbers representing each generation except the current generation]

Garbage collected

- **Collected callstack** › Object allocated on a callstack that previous deallocated objects were allocated on

- **Not collected callstack** › Object allocated on a callstack that no deallocated objects were allocated on
- **Collected (recent GC)** › Object of a type that has been collected in a recent GC
- **Not collected (recent GC)** › Object of a type that has not been collected in a recent GC
- **Collected (any GC)** › Object of a type that has been collected in any GC
- **Not collected (any GC)** › Object of a type that has been collected in any GC

Predefined Queries

Setting up queries can be time consuming and tedious. To help with this there are some built-in predefined queries.

- **Predefined queries** › select a query to use. The controls that define a query will be updated to match the selected query definition.

  The default predefined queries are:

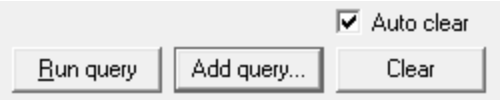
  - Rising count, no churn
  - Rising count, no churn, previous allocations collected on same callstack
  - Rising count, no churn, previous allocations not collected on same callstack
  - Rising or stable count, no activity
  - Rising or stable count, no activity, previous allocations collected on same callstack
  - Rising or stable count, no activity, previous allocations not collected on same callstack
  - No churn for 1 generation and no activity for 1 generation
  - No churn for 2 generations and no activity for 2 generations
  - No churn for 3 generations and no activity for 3 generations
  - No activity for 3 generations
  - Show objects collected by GC
  - Show objects collected by any GC
  - Show objects not collected by any GC
  - Show objects not collected by any GC where object count is increasing
  - Show objects not collected by 3 GCs where object count is increasing
  - Show objects not collected by any GC where no object activity
  - Show objects not collected by 3 GCs where no object activity

- **Add query** › a new query is formed using the selection in the six combo boxes described in the Defining a Query section. The query is added to the list of predefined queries.

- **Reset** › reset the list of predefined queries to the default predefined queries. All custom predefined queries will be lost.

Running Queries

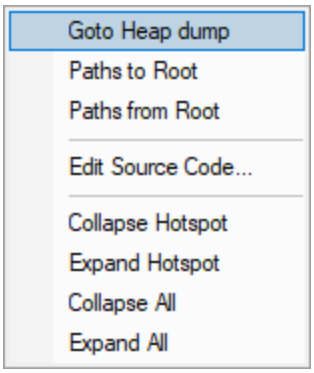
The controls at the top left define how queries are run and managed.



- **Run query** › all objects in the current generation are compared to the current query definition. If they match the query they are displayed according to the display style rules.
- **Add query** › a new query is formed using the selection in the six combo boxes. The query is added to the list of predefined queries.
- **Clear** › all query results are removed
- **Auto clear** › if selected all previous query results will be removed each time a new query is run

Leak Analysis view popup menu

The following popup menu provides options for filtering and examining data in more detail.



Menu option: Go to Heap Dump, Paths to Root, Paths from Root

The following options are only active for .Net object types. They are disabled for native object types.

- **Go to Heap Dump** › Find the heap dump entries for the selected objects.
- **Paths to Root** › For the selected objects display all the paths from the object to the most recent heap dump roots with the Paths to Root dialog.
- **Paths from Root** › For the selected objects display all the paths most recent heap dump roots to the live objects with the Paths to Root dialog.

Menu option: editing source code

- **Edit Source Code...** › opens the default or preferred editor to edit the source code

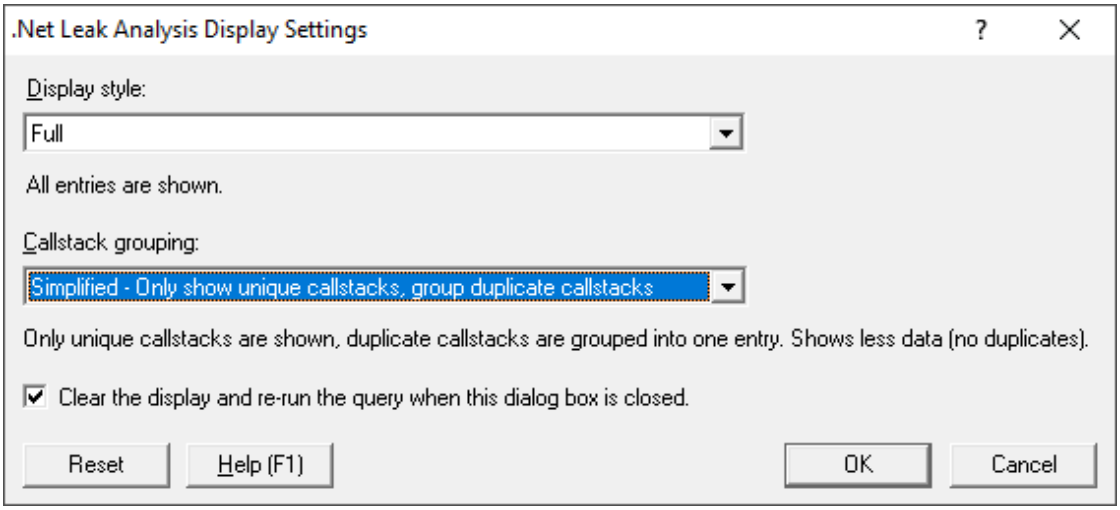
Menu options: Collapse Hotspot, Expand Hotspot, Collapse All, Expand All

- **Collapse Hotspot** or **Expand Hotspot** › simply shows and hides data item information, the same as using the or buttons
- **Collapse All** or **Expand All** › collapses or expands all callstack entries

3.7.5.3.1 .Net Leak Analysis Display Settings

The .Net Leak Analysis Display Settings control the .Net memory, .Net handles and displayed on the .Net Leak Analysis.

The default options are shown below:



The display style can be one of the following values:

- **Full** › information about every allocation and error is displayed (unless filtered)
- **Simplified - your source code at root** › Only traces that have a callstack with your source code at the top of the callstack are displayed
- **Simplified - your source code not at root** › Only traces that have a callstack with your source code in the callstack (except for the top position) are displayed
- **Simplified - your source code anywhere** › Only traces that have a callstack with your source code anywhere in the callstack are displayed
- **Simplified - compiler vendor source code at root** › Only traces that have a callstack with your compiler vendor source code at the top of the callstack are displayed
- **Simplified - compiler vendor source code not at root** › Only traces that have a callstack with your compiler vendor source code in the callstack (except for the top position) are displayed
- **Simplified - compiler vendor source code anywhere** › Only traces that have a callstack with your compiler vendor source code anywhere in the callstack are displayed
- **Simplified - no source code** › Only traces that have a callstack with no source code are displayed

The callstack grouping can be one of the following values:

- **Full** › every callstack is shown. This can be slow if there are many unique results.
- **Simplified - Only show unique callstacks** › traces that share the same callstack are displayed once. A summary is shown indicating the number of allocations, how many bytes in those allocations and the size of the largest allocation.

- **Clear the display and re-run...** › the display will be cleared and the query re-run if this option is selected.

Reset

- **Reset** › resets all the display related settings for this tab

3.7.6 Analysis

The Analysis tab provides five tools for inspecting memory usage.

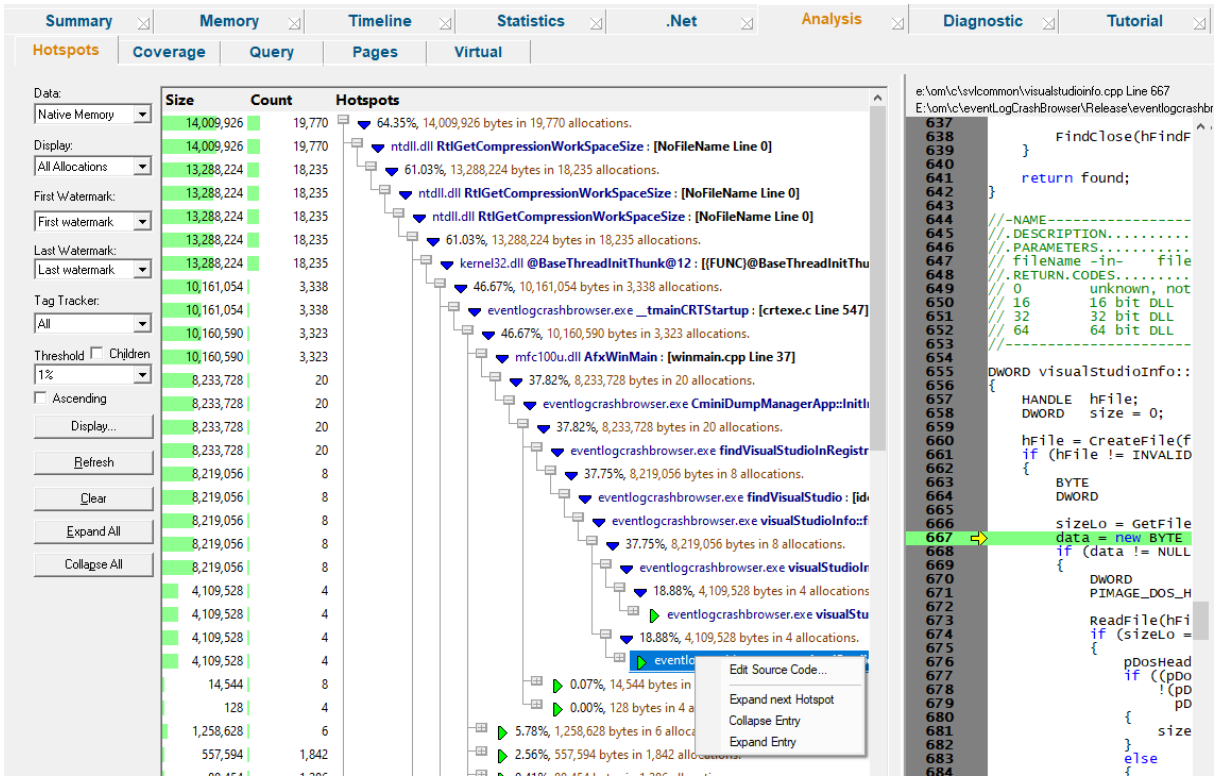
If the Analysis tab isn't visible, use the Data Views menu to set which views are shown.

Click on an item in the picture below to find out more about each of the tabbed windows, or use the list further below:

3.7.6.1 Hotspots

The **Hotspots** tab displays areas of high memory and handle allocation activity.

Read on, or click a part of the image below to jump straight to the help for that area.



Memory hotspots are locations in the target program that are responsible for the highest allocation of memory or handles.

This view displays those hotspots that have a contribution to the total allocated memory, which is greater than the threshold setting below.

The tab is split into two resizeable halves, with the left showing the hotspot callstacks and the right showing the source code for any selected row on the left.

Hotspot display styles

At the top of the Hotspot tab is a combo box containing a list of many display styles, allowing you to choose what data to examine.

- **Native Memory** › hotspot data for native memory allocations
- **Native Handles** › hotspot data native handle allocation hotspots
- **.Net Memory** › hotspot data for .Net memory allocations
- **.Net Handles** › hotspot data .Net handle allocation hotspots
- **All Memory** › hotspot data for all native and .Net memory allocation hotspots
- **All Handles** › hotspot data for all native and .Net handle allocation hotspots

- **All Native** › hotspot data for all native memory and handle allocation hotspots
- **All .Net** › hotspot data for all .Net memory and handle allocation hotspots
- **All** › hotspot data for all native and .Net memory and handle allocation hotspots

The left side of the tab shows three columns of data:

- **Size** › total size of memory or handles allocated within the hotspot area
- **Count** › how many allocations were made within the hotspot
- **Hotspots** › a hierarchical view of the function calls leading to each hotspot location

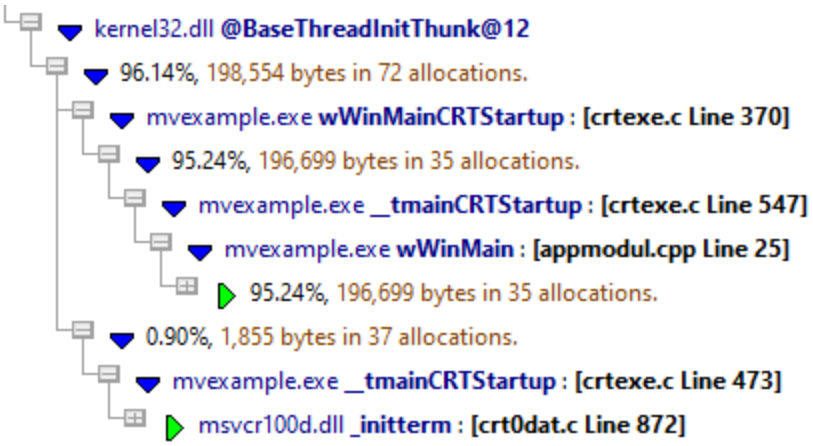
Hotspot function call hierarchy

To reduce the amount of repeated information, allocations sharing a partial callstack are merged together in the hotspot function call hierarchy.

As you expand each node in the tree, it shows the function, file and line number, although you can show more via the display settings.

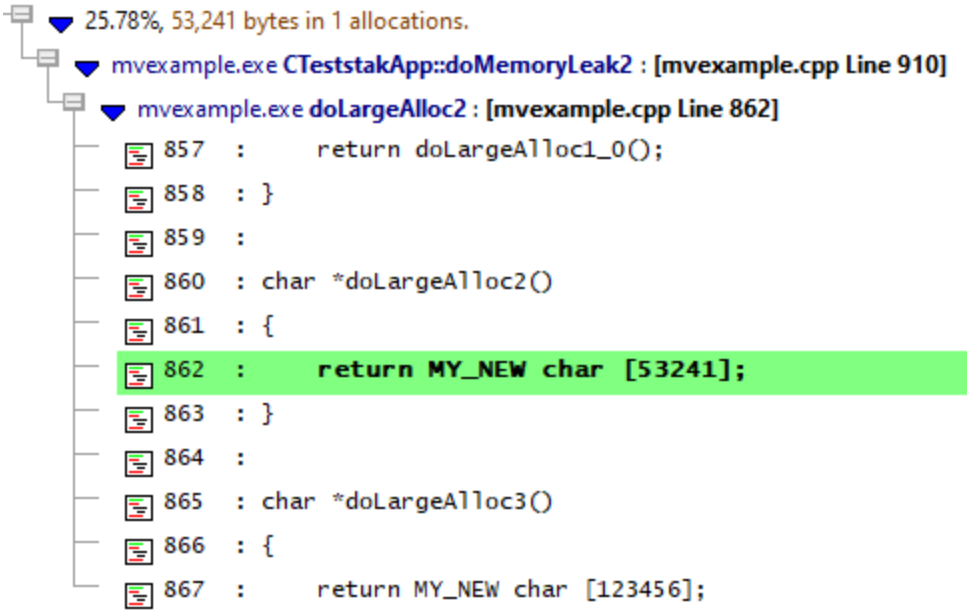
Intermediate nodes indicate the function's percentage contribution to the total allocation, along with the size and count information.

These intermediate nodes are only shown at and after the branches, i.e. where a function splits the allocations between different child callstacks.



Source code

A short section of the source code around the highlighted allocation line is shown at the end of the callstack.



You can browse the whole source file (if available) on the right when you click on any row in the hierarchy.

If you want to edit rather than browse, double clicking a row will open the relevant source code file for editing, using the editor of your choice.

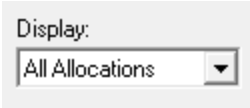
Hotspot tab options

Similar to the other tab pages, a set of options are found on the left, that control the data displayed in both the memory and the handles hotspot views.

The choice of lifetime or live data, and the setting of the hotspot threshold are specific to the hotpot tab.

Lifetime data or live data

You can choose to display all recorded allocations or just the live ones.



- **All allocations** › shows cumulative data for all allocations, reallocations and deallocations and associated hotspots
- **Current allocations** › shows only the live data that has not been freed, and the current hotspots

  If the program has ended, this will be the leaked memory, enabling you to see leak hotspots.

Be sure to click **Refresh** after changing this option and others.

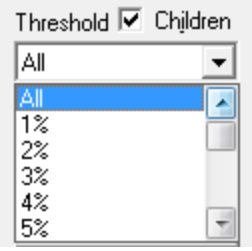
Reducing data with watermarks and tag trackers

As with other tabs, you can restrict displayed data to that between two watermarks, or allocations associated with particular tags.

The Memory tab topic describes use of the Watermark and Tracker options in detail.

Contribution threshold

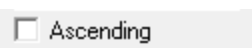
The threshold of whether a location is regarded as a 'hotspot' is determined by the percentage contribution to the overall consumption of memory or handles.



- **All** › shows the hotspot hierarchies incorporating all allocations, barring any filtering via watermarks, tag trackers etc
- **X%** › display only hotspots that contribute at least that much to the overall resources
- **Children** › controls whether hotspots use sub allocations in the hierarchies to meet the hotspot percentage criteria

Sorting

The data on the display can't be sorted, but you can change the ordering direction



- **Ascending** › swaps the ascending or descending ordering of the hotspots. Don't forget to **Refresh**!

Local filters and settings

- **Display...** › shows the Hotspot Tab Display Settings dialog to set the types of data and messages displayed in the hotspot view.

Updating the display

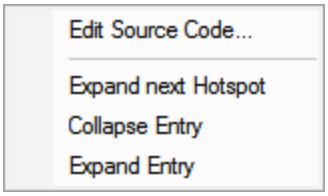
- **Refresh** › updates the display - as does the button on the Tools menu and toolbar
- **Clear** › removes all data from the display

This reduces Memory Validator's own memory usage. If you have lots of data but low memory on your computer, clearing one view before using another view may help.

- **Collapse All** › completely collapses all hotspot data items, including any source code views that were open
- **Expand All** › expands all data items down to but not including the source code snapshots

Hotspot view popup menu

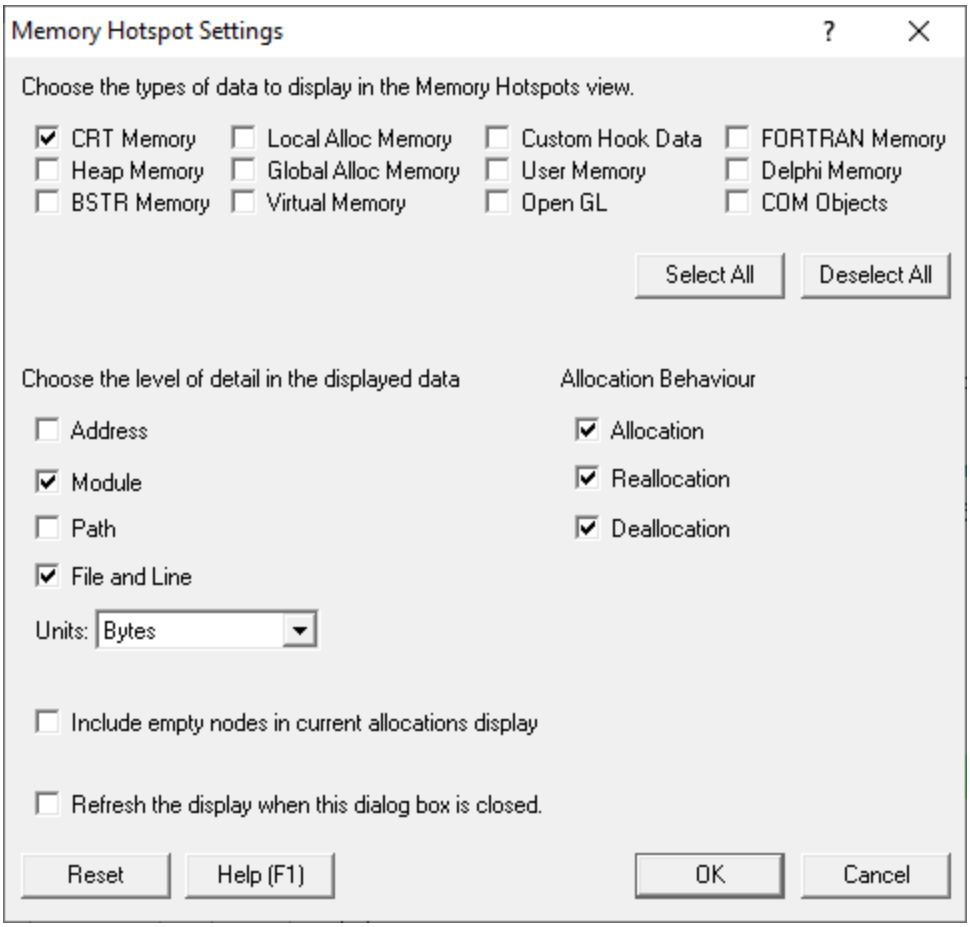
The following popup menu is available over the data area



- **Edit Source Code...** › opens the default or preferred editor to edit the source code
- **Expand next hotspot** › expands the hierarchy only as far as the next hotspot, but not the source code snippet
- **Collapse entry** › collapses the selected hierarchy of hotspots
- **Expand entry** › expands all hotspots in the selected hierarchy

3.7.6.1.1 Hotspot Display Settings

The Memory Hotspot Settings dialog controls the type of data that is displayed on the Hotspot tab.



Types of memory to display

The following types of memory allocations can all be optionally displaying in the hotspots view.

- CRT memory
- Win32 heap
- BSTR objects from SysAllocString function
- LocalAlloc()
- GlobalAlloc()
- Virtual memory
- Custom memory allocations using the Custom Hooks functionality
- User memory allocations using the API
- OpenGL
- FORTRAN memory
- Delphi memory
- COM objects

Only CRT Memory allocations are enabled by default.

Note that these settings control what is *displayed*, not what is *collected*. The data collection settings may have more information about some of the settings above.

Level of detail displayed

You can choose the level of detail displayed in each entry on the hotspots view:

- **Address** › displays the address as part of a hotspot entry
- **Module** › shows the module load address
- **Path** › includes the full file path or just the filename when showing file and line below
- **File and Line** › shows the filename and line number
- **Units** › gives allocation sizes in Bytes, KB or MB

Allocation behaviour

Memory Validator refers to an allocation as having an *allocation behaviour* - indicating one of the following:

- **Allocation**
- **Reallocation**
- **Deallocation**

The default is to include all three behaviours in the hotspot calculations

Empty nodes

When the Memory tab option is set to show only live data rather than lifetime data, it is possible for the total number of bytes (or handles) displayed to be zero.

- **Include empty nodes...** › include zero byte or zero handle entries in the hotspot listing

Refresh and reset

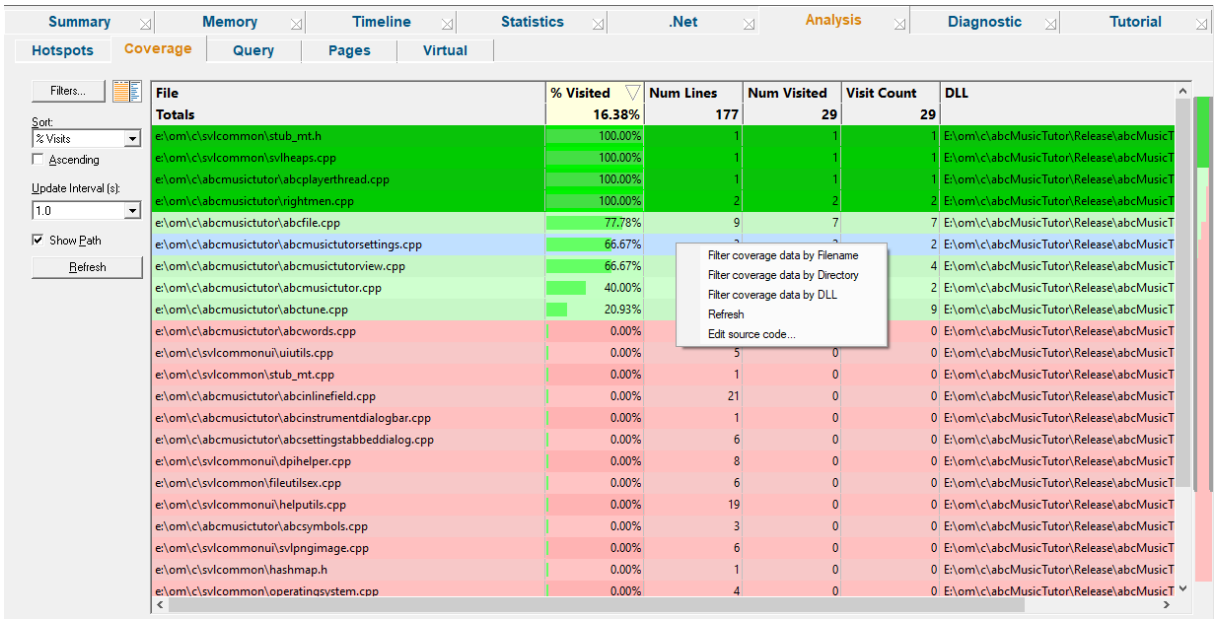
- **Refresh the display...** › enables updates of the hotspot display whenever this settings dialog is closed
- **Reset** › revert all these settings to the default values shown in the picture above

3.7.6.2 Coverage

The **Coverage** tab lets you find untested parts of your program's memory and resource allocation.

If the Coverage tab isn't visible, use the Data Views menu to set which views are shown.

Read on, or click a part of the image below to jump straight to the help for that area.



Using the coverage information

Inspecting the coverage tells you which memory allocation, reallocation and deallocation locations are or are not being tested, including resource handles.

Understanding the coverage helps you plan and improve your regression tests to include areas of code that allocate memory or deallocate memory but are not yet being visited.

The display shows two resizable panes, one with the coverage data, and the other shows source code when you click on a row in the table.

In order to gather coverage statistics, you'll need to switch on the memory coverage setting.

Colours used in the displays

Each file's row is coloured according to whether it has:

no lines visited

some lines visited

 all lines visited

 been filtered out for subsequent sessions (see below)

The % Visited column and the source code view uses:

 for the percentage of lines visited, or visited lines of source code

 for unvisited lines

Coverage data

The data in each column is summarised below

- **File** the statistics are gathered for each source code file found
- **Num Lines** number of lines in each file that allocate, reallocate or free memory and handles
- **Num Visited** the number of those lines that have been visited
- **Visit Count** total number of visits to those lines
- **% Visited** the percentage of relevant lines visited (Num Visited / Num Lines)
- **DLL** the DLL responsible for the file

The Visit Count may be equal to the Num Visited if you have opted to keep the default memory coverage setting of counting visits to each allocation only once. You can change this setting on the fly to start counting multiple visits right away.

At the top of the table is a **Totals** line showing combined results for all files.

The statistics here only cover lines that affect *memory allocation*, unlike SoftwareVerify's sister tool C++ Memory Validator which determines complete code coverage.

Sorting columns

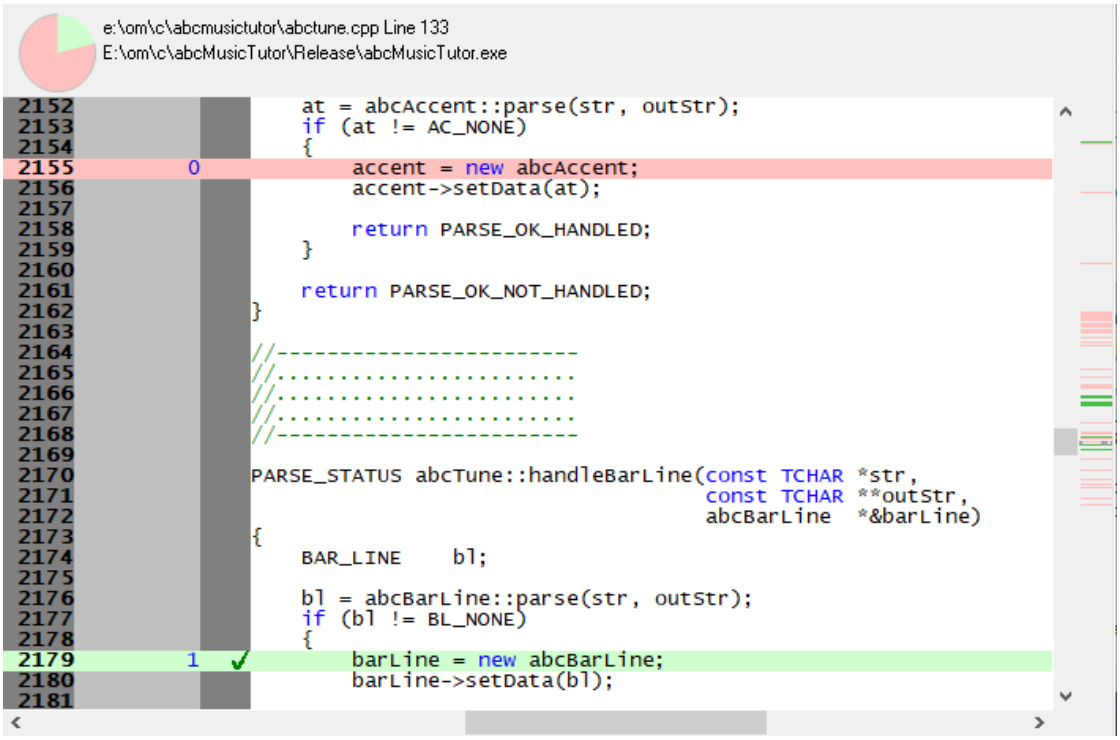
Sorted columns are highlighted yellow in the header. Just click on the header to change the sorting column or it's sort direction order.

➡ See also the sorting option below.

Source code view

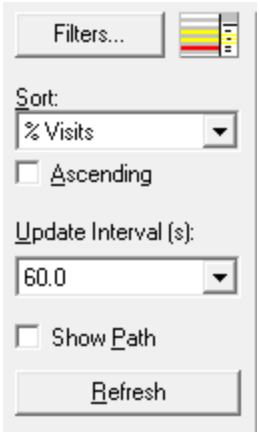
The source code view is syntax-highlighted with green visited and blue unvisited lines.

The columns at the left show line numbers and visit counts for each allocation line which are also available via a tooltip.



Coverage options

The following controls are displayed to the left of the coverage results



Filters

Unwanted results in the coverage can be excluded via the filters according to filename, directory or DLL.

Filters can be managed in the filters dialog or added via the menu options below.

- **Filters...** > shows the coverage filters dialog

The filters dialog is the same one as found on the memory coverage page of the global settings dialog where it is described in detail.

Window orientation

The data and source code panes can be arranged horizontally or vertically with the orientation button.

Sorting

Using the drop down list to select any of the column heading items and using the ascending check box is exactly the same as sorting using the column header.

Updating the display

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds - or not at all!

  Adjust this depending on the complexity of your application.

- **Refresh** › updates the display - as does the button on the Tools menu and toolbar, and the button on the popup menu

  With an update interval set to No Update, you'll need to use this Refresh button to update the display when you wish.

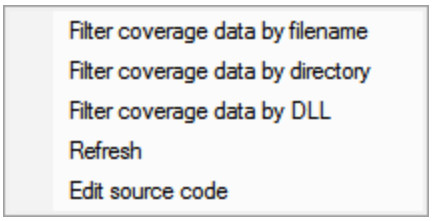
Display settings

On the coverage tab, there's only the one display option:

- **Show Path** › shows the short file name or the longer file path in the **File** column of the data

Coverage view popup menu

The following popup menu is available over the data area to add filters, refresh the view or edit source code.



Menu options: Filtering coverage

The first three menu options let you add filters at different levels of granularity.

Filters become effective at the start of the next session. Adding a filter during a session will show the row in grey so that you can see which files are filtered, but the coverage results will continue to be included for the rest of the session.

- **Filter coverage data by filename** › adds a new filter to the Filters dialog, excluding the selected file from the results of a subsequent session
- **Filter coverage data by directory** › excludes all files in the same directory as the selected file
- **Filter coverage data by DLL** › excludes all files belonging to the same DLL as the selected file

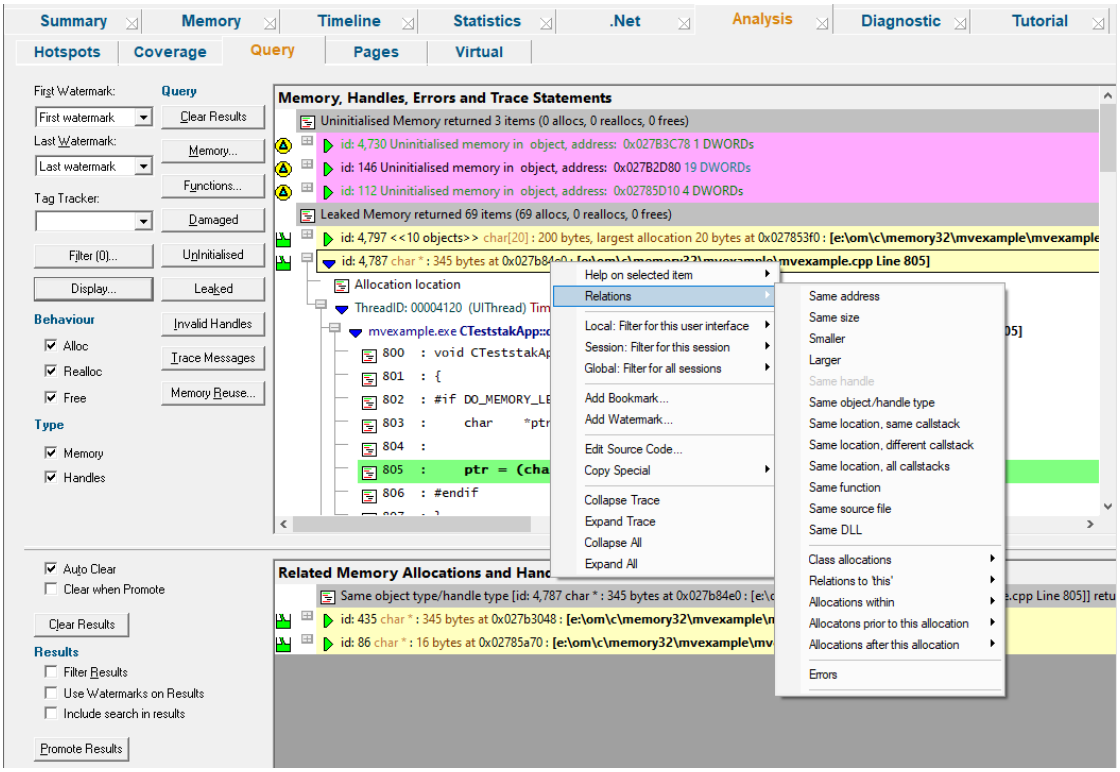
Menu option: editing source code

- **Edit Source Code...** › opens the default or preferred editor to edit the source code

3.7.6.3 Query

The **Query** tab shows groups of search results and allows you to find many types of related data for different memory allocations.

Click a part of the image below to jump straight to the help for that area.



If you've read the previous sections then you'll probably have seen menu options referring to showing results here on the analysis tab.

Think of the analysis tab as a basket where you can send groups of data to inspect at your leisure.

That data will appear in the upper of the two resizable windows and becomes your *working data* from which you can also find related allocations to display in the lower window.

The Analysis Data

Both the upper and lower views show data in the same format as that found in the Memory tab's collected data.

Each line has an icon at the left, indicating its type, and has an explanatory tooltip:

The text on each line is the same and indicates:

- datatype (if known)
- size
- allocation address/handle value
- source file and line number (if available) where the allocation occurred
- an optional event sequence id at the beginning of the line

The background colour for each line indicates the status of the data - eg, leaked, damaged, or uninitialized.

To edit the source code, double click on any part of the lines of source code displayed or use  **Edit Source Code...**

The upper window - working data

The upper window will contain any data sent from other tabs or results of queries made via the buttons at the left.

This becomes your working data, and will grow (with a header line between each group) as each set of results is added.

Using the relations option on the popup menu, you can then find related allocations or objects which are displayed as separate results in the lower window.

The lower window - results within results

Having obtained related data in the lower window you can inspect it in the same way, filter it and promote it back up to the top window.

Navigation shortcuts

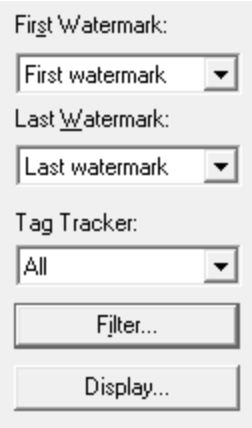
Two navigation keyboard shortcuts are provided. They collapse the current entry, move to the next or previous entry and then expand that entry to show the full callstack.

To move to the previous top level entry press 

To move to the next top level entry press 

Analysis tab options - upper window

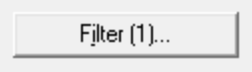
At the far left of the window are the now familiar options for filtering data using the the Watermark and Tracker methods outlined for the Memory tab.



Also familiar, are the filter and display settings:

- **Filter...** › shows the local filters dialog for the memory tab

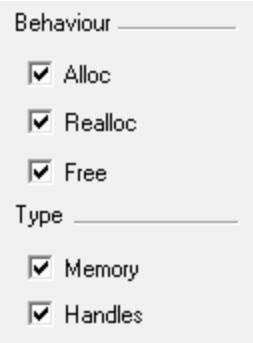
The filter button also indicates the *number* of local filters, although not all of these may be enabled



- **Display...** › shows the Analysis Display Settings dialog

Unlike other tabs, there are no options to update the display here, since the data is always static.

Instead, there are options to change the type of memory allocation you are interested in, and these will only take effect on any new queries added to the display.



- **Behaviour** › choose to show memory that has reached one or all of the allocation, reallocation and deallocation stages
- **Type** › choose to show query results for handles, or memory as per the above behaviour, or both
- **Clear Results** › removes all working data from the upper window only as the lower window has its own clear button

Running data queries

The data in the analysis tab is not limited to that sent from other tabs.

From the buttons at the left you can run some common or very targeted queries to search for allocations, reallocations, and deallocations of memory and resource handles.

Custom queries

Several of the main tabs have some comprehensive methods of querying memory or functions, and which are accessible from the main query menu and query toolbar.

The Analysis tab has dedicated buttons for two of these queries:

- **Memory...** › Shows the Find Memory dialog to use a wide range of search criteria to find memory
- **Functions...** › Shows the Find Functions dialog but displays the results here in the analysis tab

Predefined queries

There are a selection of common predefined queries available at a single click:

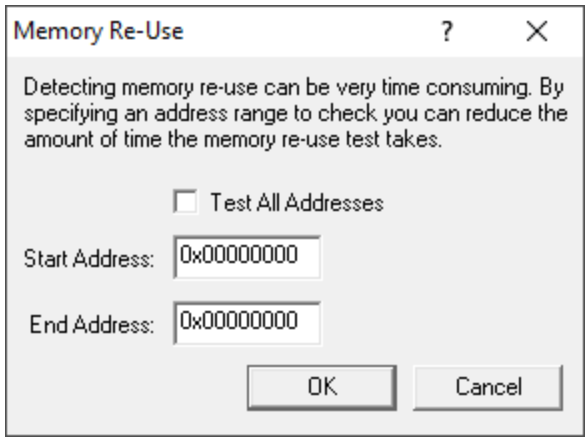
- **Damaged** › finds all damaged memory allocations, such as overwrite, underwrite, double delete, etc
- **Uninitialised** › shows any detected uninitialised memory if you've switched these hooks and settings on
- **Leaked** › finds all leaked locations, memory and handles still in use after the application has exited
- **Invalid Handles** › finds all invalid or NULL handles
- **Trace Messages** › shows any trace messages if you've switched the trace hook setting on

Memory reuse

Memory re-use is not necessarily an error, in fact it almost certainly isn't, as the allocator will provide previously freed memory addresses as memory allocation addresses.

For certain bugs, memory corruption being an example, you may be interested in knowing which locations allocated memory allocations at a particular address,
and if that address has been re-used. These are targets for further investigation.

- **Memory Reuse** › searches for reused memory in the application



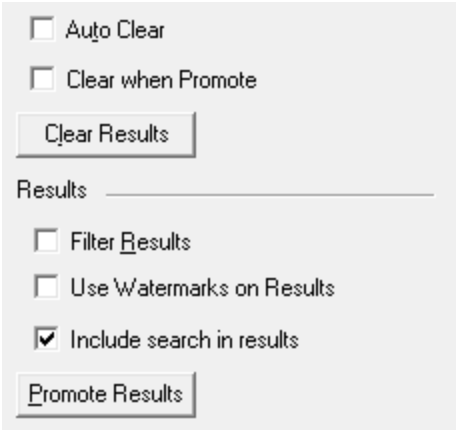
Because search through all memory may take a long time in a large and active application, you can instead opt to search within a range of memory:

The start and end addresses for the memory range can be entered in decimal or hexadecimal format with the leading `0x`

Note that some libraries outside your control may reuse memory, so be aware that not all results found are necessarily errors in your application.

Analysis tab options - lower window

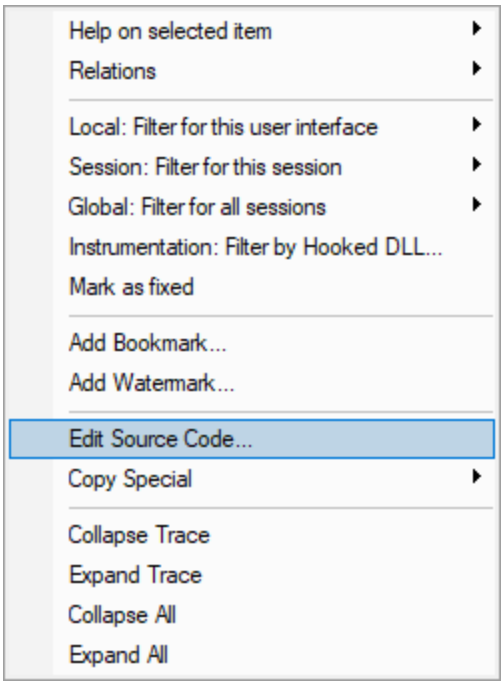
The lower window has it's own comparatively simple set of options:



- **Auto Clear** › clears the lower window before adding a new relations search from the upper window
- **Clear when Promote** › clears the lower window when promoting these results to the upper window
- **Clear Results** › simply empties the lower window
- **Filter Results** › optionally filters the lower window data using the same filters as the upper window
- **Use Watermarks on Results** › applies the upper window's watermark settings to the lower window
- **Include search in results** › brings the upper window item that initiated the relations search along with the results
- **Promote Results** › pushes all or selected lower window results into the upper window, optionally adding to or replacing what's there already

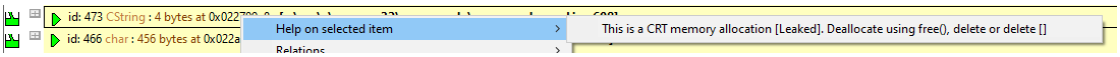
Analysis view popup menu

The following popup menu is available over the upper window.



Menu option: data item summary

- **Help on selected item** > the sub-menu shows a simple one line description of the selected entry:



Menu option: relations

The relations menu has a large sub-menu with many different options for choosing a set of related data to display in the lower analysis window.

Think of this as a sub-query on the working data - like searching for friends of friends on a social network!

Given an entry in the upper window, available relations are as follows, with *allocations* generally meaning any allocation, reallocation or deallocation

• **Same address** • **Same size** • **Smaller** • **Larger**	Finds any other allocations on the same memory address, for example previous allocations or frees Allocations on any memory objects of identical size or on smaller or larger objects
• **Same handle**	Finds any other allocations of the same resource handle
• **Same object/handle type**	Finds any other allocations of the same type
• **Same location, same callstack** **different callstack** **all callstacks**	Finds other allocations made at the source code location: via the same callstack different callstacks or any callstack
• **Same function** • **Same source file** • **Same DLL**	All allocations from the same function... or the same file or the same DLL
• **Class allocations**	All allocations, reallocations or deallocations from the same C++ class
• **Relations to 'this'**	Finds various other events relating to the selected object: **Allocator of this** - only for reallocated objects **Reallocation of this** **Reallocation of this address at same address** **Reallocation of this address at different address** **Deallocation of this** **Allocations, reallocations, deallocations** **Referenced by this** - these two need the target application to still be running

• **Same address** • **Same size** • **Smaller** • **Larger**	Finds any other allocations on the same memory address, for example previous allocations or frees Allocations on any memory objects of identical size or on smaller or larger objects
	Referencing this
• **Allocations within**	For memory allocations, finds all other allocations within a range of 32 bytes up to 4Kb of this one
• **Allocations prior** • **Allocations after**	For memory allocations, finds the previous 5, 10 or 20 allocation events or the next 5, 10 or 20 events
• **Errors**	Shows any known damaged memory allocation information relating to this entry

Menu option: filters

The filter options here are identical to that of the Memory tab menu options.

The three filter options on the menu let you create different types of filters at different scopes, using various attributes of the selected data item.

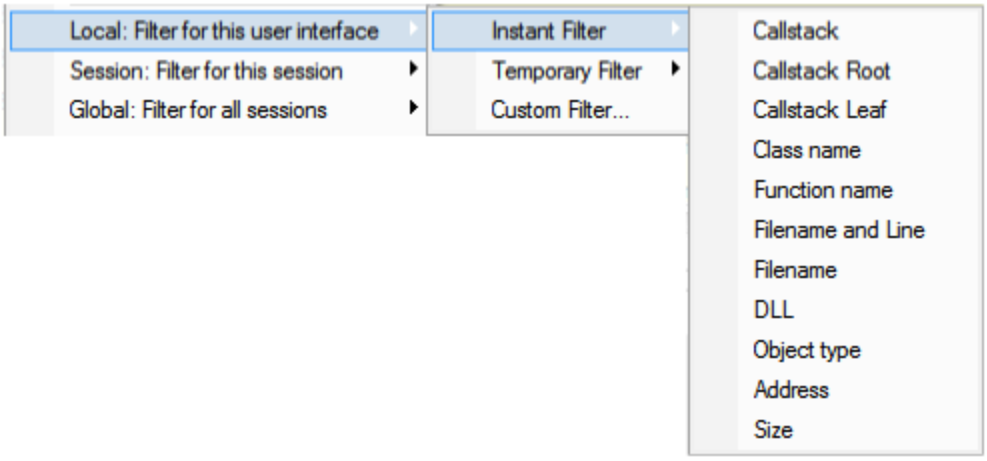
The types and scopes of filters in Memory Validator are described in detail elsewhere, but here's a quick summary:

- **Local filters** › affect the current tab only and are managed locally via the filters button
- **Session filters** › affect all tabs until the end of the session, and are saved with sessions
- **Global filters** › affect all tabs, and are persistent between sessions until removed from the global filters

For each scope, you can create instant, temporary or custom filters:

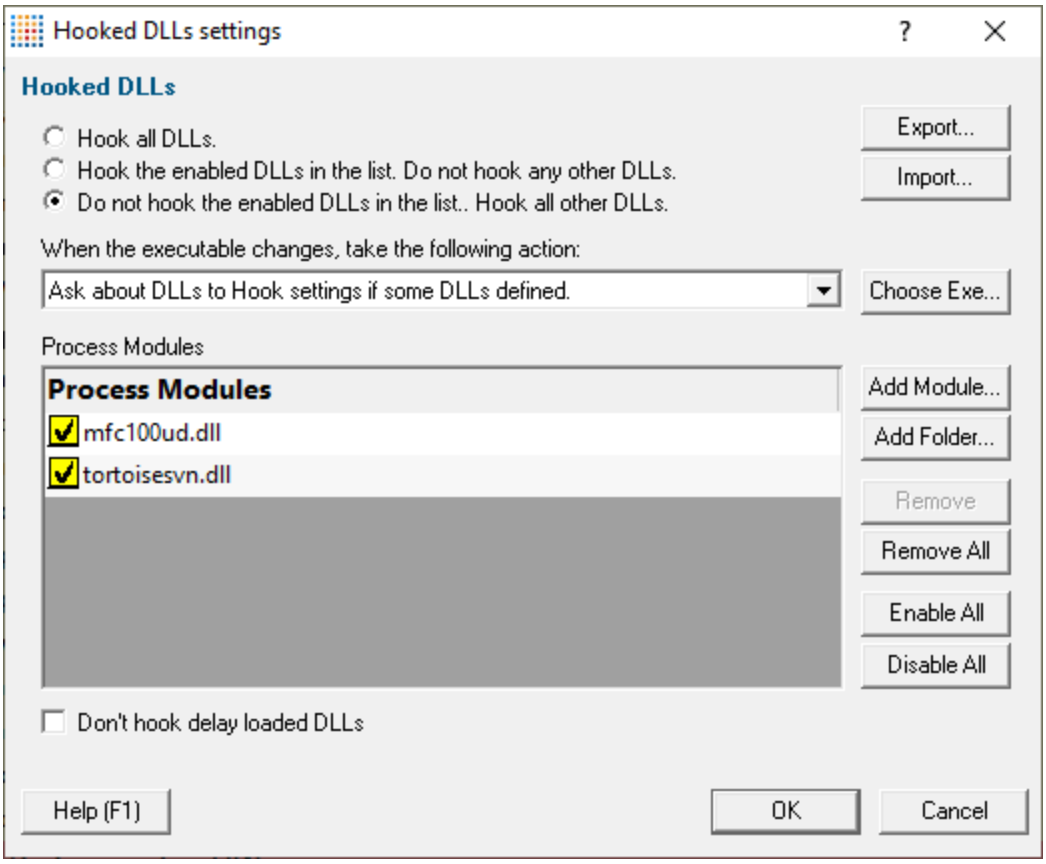
- **Instant** › quick and easy - this uses the selected data item and requires no further input from you
- **Temporary** › like instant filters but not saved with any session data
- **Custom** › allows you to define the characteristics of the filter

Finally - for each scope and type of filter, you can use any of the following elements of the selected data item as the filtering characteristic:



A fourth option allows you to filter the next run by DLL, either excluding that DLL, or including that DLL, in the list of DLLs that will be monitored.

The Hooked DLLs settings dialog is displayed.

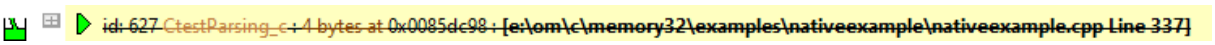


Menu option: mark as fixed

- **Mark as fixed** › marks the selected item as "I have fixed this"

To remove the marking from the event press the shift key at the same time you choose **Mark as fixed** on the menu.

Items that have been marked as fixed are shown with a line struck through them.



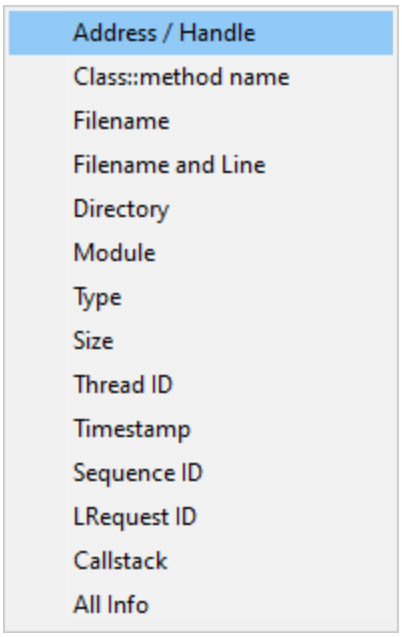
This allows you to easily identify items that you've worked on and items that have yet to be worked on.

Menu option: editing source code

- **Edit Source Code...** > opens the default or preferred editor to edit the source code

Menu option: copy special

The copy special sub-menu lets you copy to the clipboard any of the following attributes, (or all the information):



See the same option on the Memory view menu for more information.

Menu option: bookmarks and watermarks

Bookmarks allow you to find a data item easily at a later date, while watermarks are used above to show only those items between two points in time

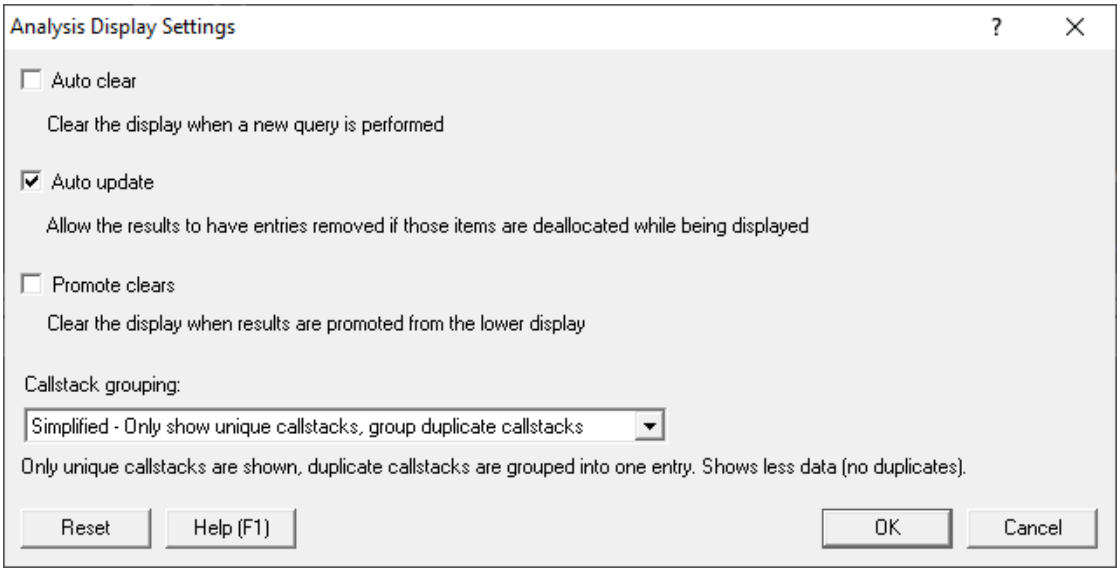
- **Add Bookmark...** › adds a bookmark for the selected item
- **Add Watermark...** › adds a watermark for the selected item

Menu options: collapse / expand trace

- **Collapse** or **Expand Trace** › simply shows and hides data item information, the same as using the or buttons
- **Collapse All** › completely collapses all data items in the upper window, including any source code views that were open
- **Expand All** › expands all data items down to but not including the source code snapshots

3.7.6.3.1 Analysis Display Settings

The Analysis Display Settings dialog controls the grouping and removal of data shown on the Analysis view.



Clearing the display with new data

The display can be cleared automatically when new data is added.

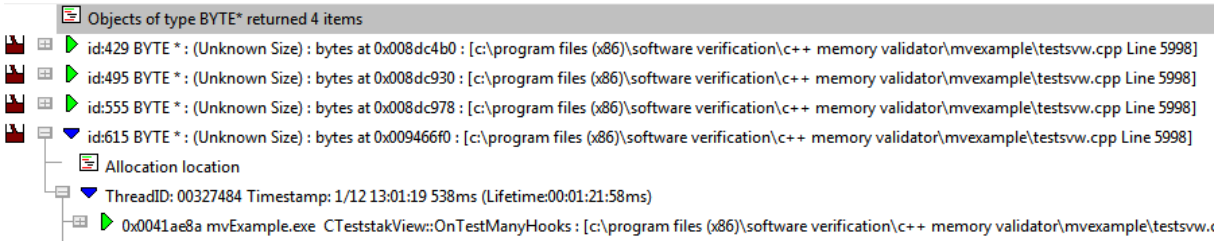
- **Auto Clear** › clears the upper window on each new query or when new results are sent from another tab
- **Promote Clears** › clears the upper window when data is promoted from the lower window

Grouping by callstack

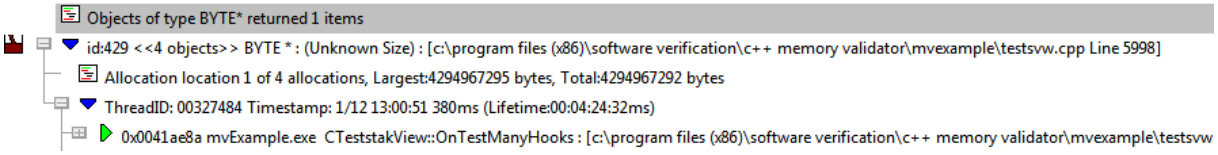
You can opt to show allocations from similar callstacks as individual entries or as single summary entries.

- **Group by Callstack** › displays all allocations having the same callstack as a single entry

For example, in the mvExmple application, choosing the menu option **Allocation** › **Test Many Hooks at once** four times would normally show the following



But if entries were set to be grouped by callstack, you'd see this instead:



The summary indicates the number of allocations along with the largest allocation and the total size for all items.

Removal of purged data

Using the allocation history global settings, you can set how much data Memory Validator keeps for reallocated or freed memory.

If you are keeping your analysis results around for a while, or you have a large and active application, you may want to ensure that your analysis results are not affected when older data is purged in this way.

- **Auto Update** › allows older callstacks to be removed if their historical data is purged

  The default is to auto update the results to remove purged data, but note that if you disable this and have limited memory resources, you should re-enable it when possible, to ensure memory consumption does not build up unnecessarily.

Reset

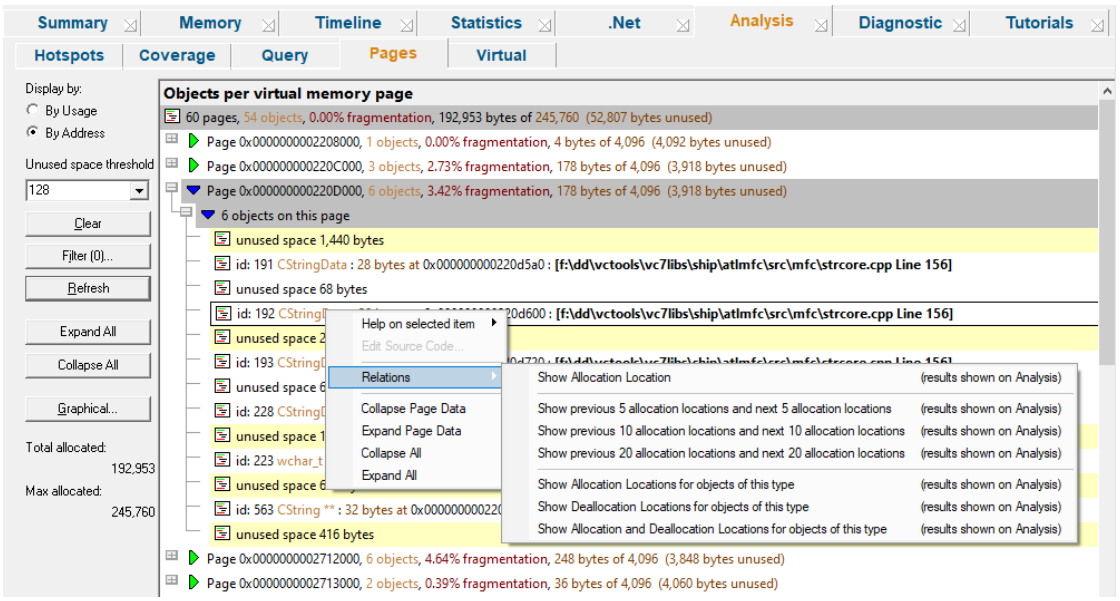
- **Reset** › resets the options to the default options shown in the screenshot at the top of this section

3.7.6.4 Pages

The **Pages** tab displays all known memory allocations according to the virtual memory page in which they were allocated.

If the Pages tab isn't visible, use the Data Views menu to set which views are shown.

Click a part of the image below to jump straight to the help for that area.



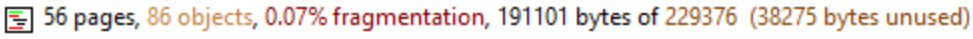
This tab provides a snapshot of how the memory in use is distributed across virtual memory pages (4K in size).

By looking at the data, especially the graphical view, you can see the memory fragmentation, and find out which parts of the application are using each memory page.

The Pages Data

The data lists all virtual memory pages used by the target application, omitting those pages that have no allocations within or overlapping them.

The first line of data gives an overall summary, for example:



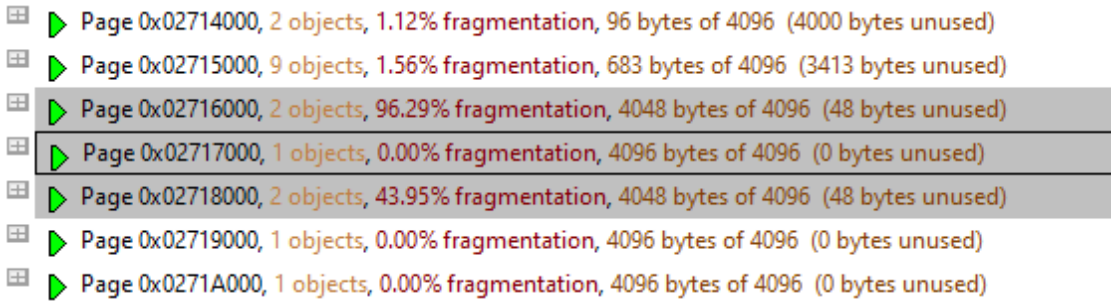
The remaining expandable items are shown in page order:

Each page item shows:

- the page address
- the number of allocated objects completely or partially in the page
- the percentage fragmentation within the page
- number of bytes used
- unused bytes

Expanding an item shows details about allocated objects and free space in each page.

Selecting one page in the list may highlight others either side in gray as below. This indicates that those pages are linked by memory allocations spanning between or across those pages.

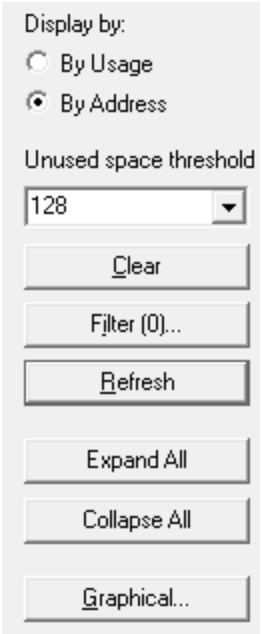


Display options allow you to focus either on the objects within each page or on the *order* of the objects and free space.

Note that unlike many of the other data tabs, the pages data is not automatically updated at regular intervals. You'll have to click Refresh to get an update.

Page tab options

At the far left of the window are some options for updating and viewing the data.



Display settings

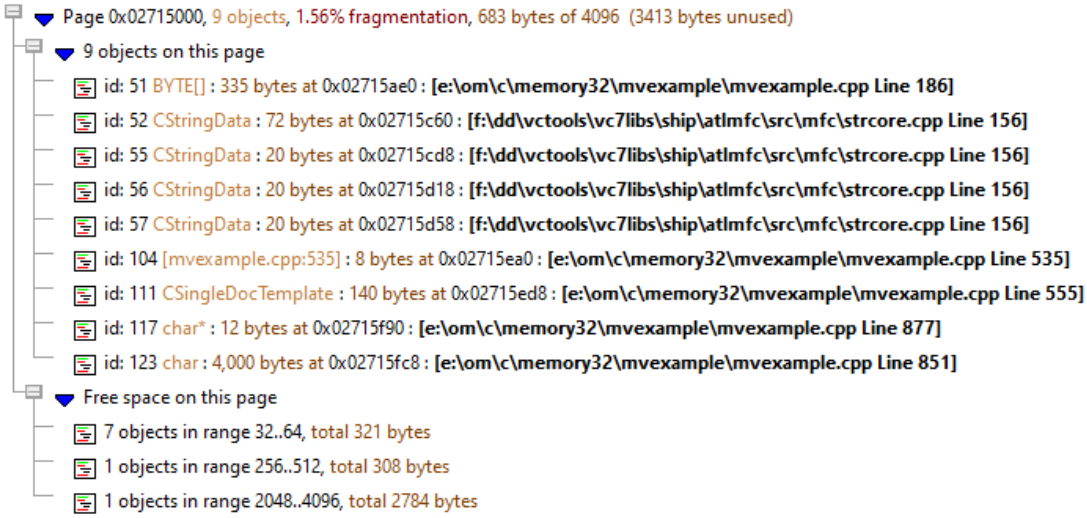
The display option at the top changes how the data is displayed when you expand each page data item:

- **Display by Usage** › orders the information within each page by the objects using it, and then by the chunks of free space, grouped by size
- **Display by Address** › shows each page's information by the start address of each object or chunks of free space within that page
    - **Unused space threshold** › when displaying by address, free space blocks greater than this threshold are highlighted as below

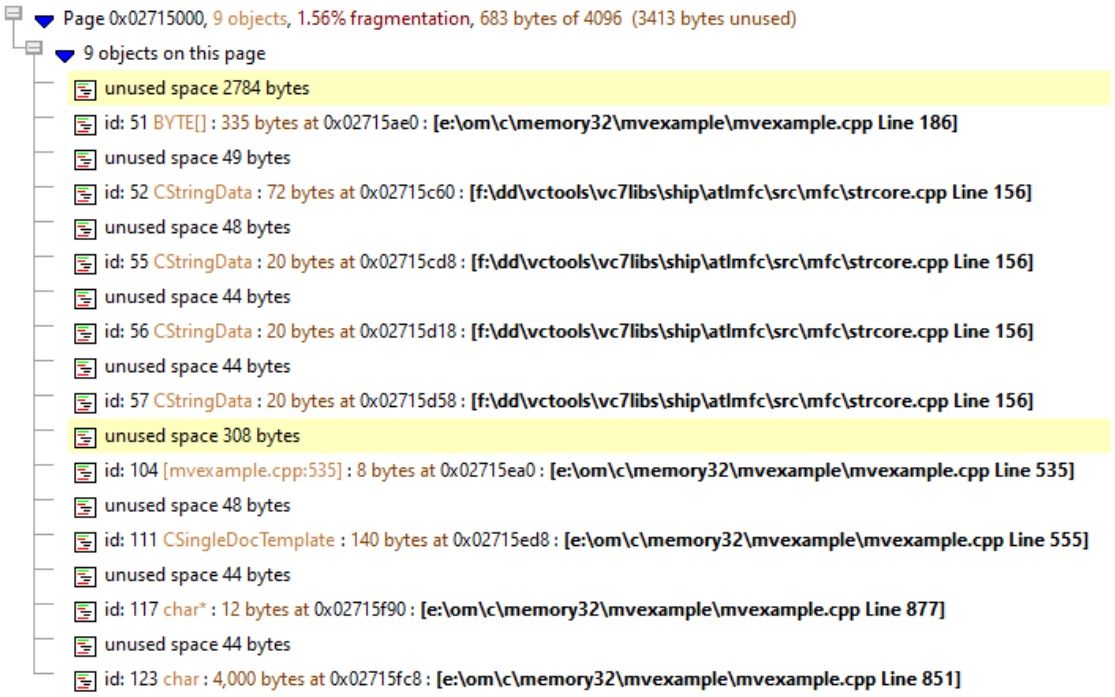
Each allocated object in the expanded view shows data type, size, address, and source file with line number if known.

Below are examples of a page's details expanded for each display method:

By usage:



By address, highlighting free space blocks over the set threshold:



Updating the display

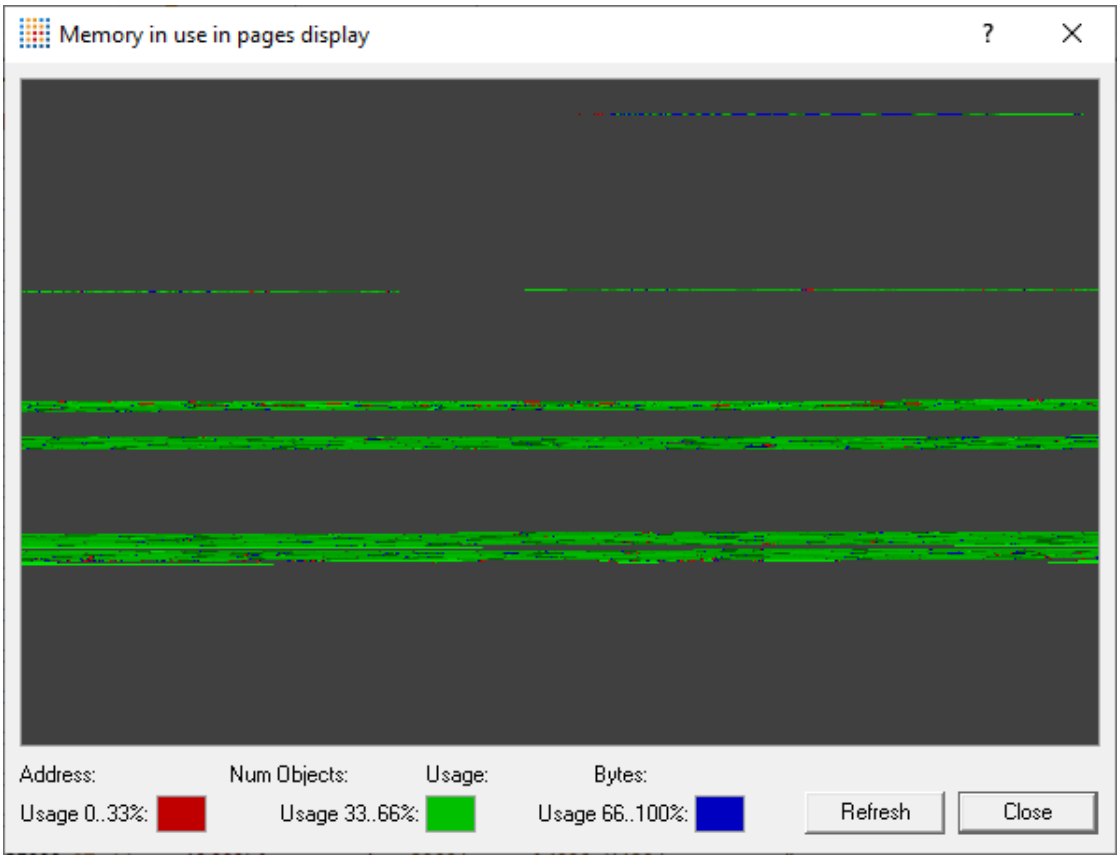
- **Clear** › removes all data from the display

  This can be useful if you have limited RAM and inspecting large applications as it frees up resources used by Memory Validator so that you can use another tab view.

- **Refresh** › updates the display - as does the button on the Tools menu and toolbar
- **Expand All** › expand all entries on the display
- **Collapse All** › collapse all entries on the display

Graphical view

- **Graphical** › shows the following dialog visualizing the page data in memory



This view uses a single coloured pixel for each page of memory, wrapping from left to right and top to bottom:

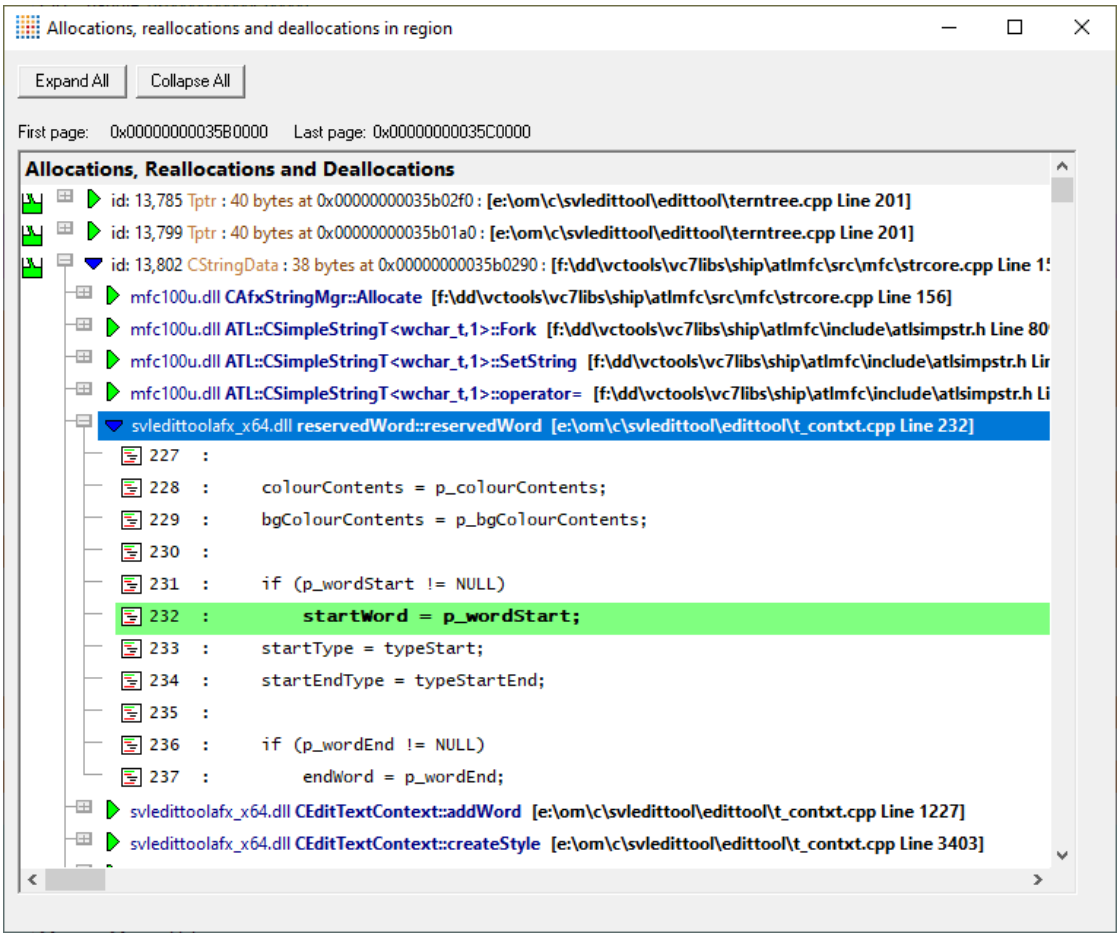
- Red › very fragmented - 0 to 33% usage
- Blue › fragmented - 33 to 66% usage
- Green › least fragmented - 66 to 100% usage

Hovering the mouse over any coloured area in the graph shows the address, number of objects, and the usage as a percentage and number of bytes just below the graph.

The graph has a popup menu giving access to allocations, reallocations and deallocations as follows:

- **Page information...** › allocations in the selected page (pixel)
- **Region information...** › allocations in the same contiguous block of colour (i.e. with the same fragmentation level)

An example of the page information dialog is below. The region information shows the first and last page address accordingly.

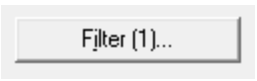


The allocations in the dialog can be expanded, to show the local source code, and double clicking launches your preferred editor with the source file.

Local filters and settings

- **Filter...** > shows the local filters dialog for the Pages tab

The filter button also indicates the *number* of local filters, although not all of these may be enabled

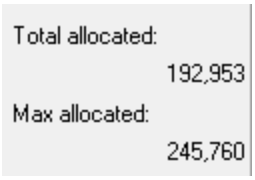


Searching for memory in pages

You can search for items in the displayed data using the  button in the toolbar, and the **Search** option in the Query menu.

Totals

At the bottom of the options on the left are two totals:

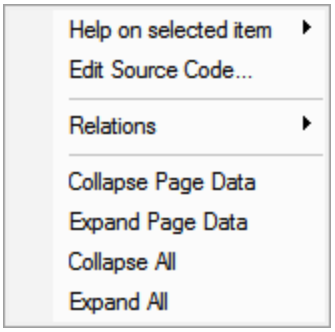


- **Total allocated** › gives the total memory consumption for all allocated objects
- **Max allocated** › shows the total allocation capacity for all the virtual memory pages currently in use

  This would simply be the number of pages multiplied by 4K, so in the example above 57 pages * 4096 = a capacity of 233,472 bytes

Pages view popup menu

The following popup menu is available over the data area



Menu option: Help on selected item

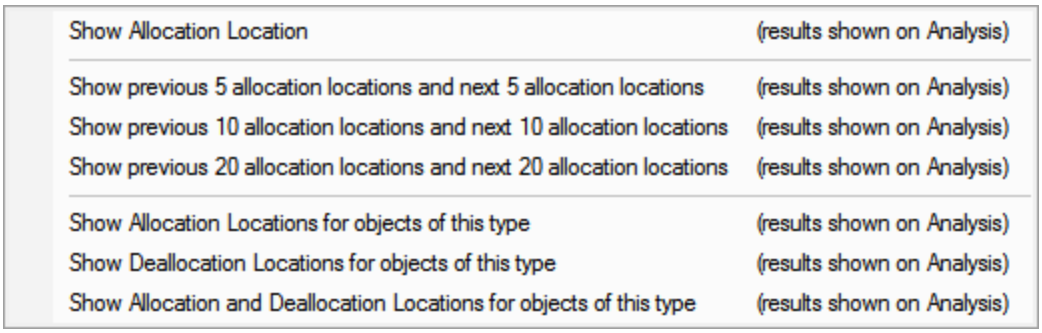
- **Help on selected item** › the sub-menu shows a simple one line description of the type of data that has been selected:

Menu option: editing source code

- **Edit Source Code...** › opens the default or preferred editor to edit the source code

Menu option: relations

When used over an allocation within a page, the relations sub menu lets you show some related allocations in the Analysis tab to inspect in more detail.



- **Show Allocation Location** › adds only the allocation for the selected item in the page to the Analysis tab
- **Show previous and next...** › adds the available allocations for the allocations in the specified event sequence range, relative to the selected item in the page

For example, choosing the previous and next 10 allocations, shows allocations in the range id-10 to id+10, but only those that are still accessible.

- **Show Allocations/Deallocations for type** › adds all the allocations/deallocations (or both) to the Analysis tab that match the object type of the selected item in the page

For example, if the allocation in the page is of type `BYTE[]`, this will show every allocation that is of the same type, including the one you selected.

Menu options: collapse / expand page data

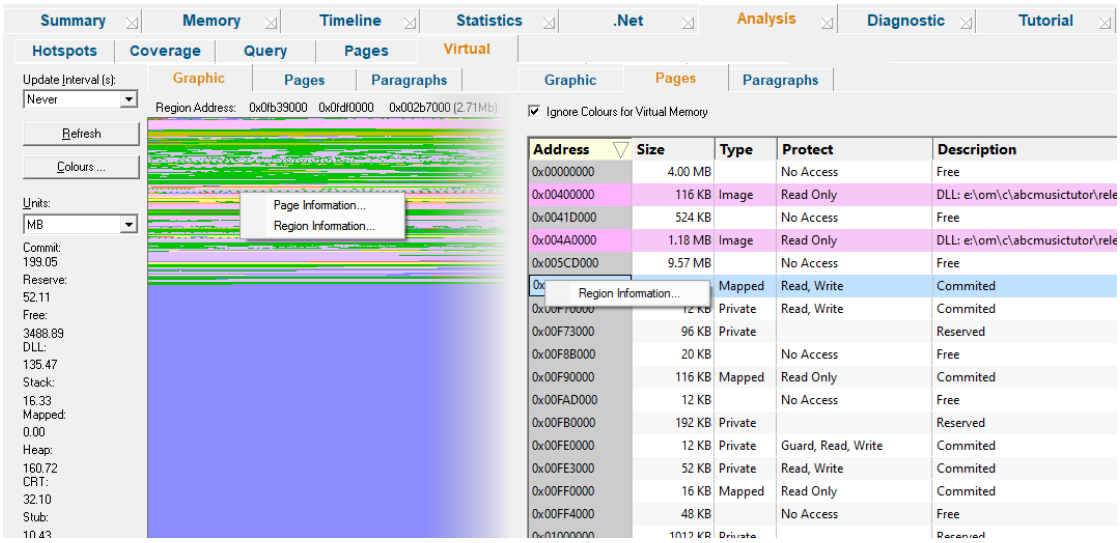
- **Collapse** or **Expand Page Data** › simply shows and hides the information for the selected page, the same as using the ⊞ or ⊟ buttons
- **Collapse** or **Expand All** › completely collapses or expands all the pages information in the display

3.7.6.5 Virtual

The **Virtual** tab shows a graphical or tabulated view of all the memory in the target program, and how it's being used.

If the Virtual tab isn't visible, use the Data Views menu to set which views are shown.

Click a part of the image below to jump straight to the help for that area.



The virtual tabs

At the top of the display are three tabbed data views: one graphical, and two tabulated:

- **Graphic** › a graphical visualisation of virtual memory pages, similar to that used on the main Pages tab
- **Pages** › a tabulated view of each 4K virtual memory page used by the target application
- **Paragraphs** › a tabulated view of each virtual memory paragraph

Each of the three views includes data from Win32 Heaps, CRT heaps, DLLs, Memory mapped files and thread stacks.

Note that the view supports an address range of 2Gb, or 3Gb on suitable operating systems if the target application has been built with the /3GB and /LARGEADDRESAWARE parameters.

Memory usage types

The virtual tab views show the memory type as being one of the following. Default colours are shown here, but can be changed.

Commit › Commited memory, allocated with VirtualAlloc()

Reserve › Reserved memory, allocated with VirtualAlloc()

Free › Free memory, whether unused or deallocated

DLL › used by EXEs, DLLs and assemblies in the program

Stack › stack space used by threads

Mapped › memory mapped files

Heap › allocated by HeapAlloc() etc

CRT › allocated by the C runtime: malloc, calloc, new etc

Stub › workspace memory used by Memory Validator

out of address range

Normally, the virtual memory manager treats memory only as free, committed, or reserved, since it doesn't care that a given region of memory is stack space, DLL storage space, a memory heap etc. Memory Validator adds the additional memory types to make the results more useful.

Graphic view

The graphic view uses a single pixel, coloured as above, to visualise the state of each 4K page of memory, wrapping from left to right and top to bottom.

If there is no target program being monitored, or the program has closed, the display will appear black.

Hovering the mouse over the display shows information at the top of the view about the region and page under the pointer.

- **Region Address:** › shows the start and end address of the contiguous coloured region of memory, and its size (hex and decimal)
- **Page Address:** › shows the page address
- **<Memory usage information>** › shows the memory type (as above) at the end of the line and optional extra information

  For a heap, its handle value is also displayed.
  If the region is a process module (DLL, EXE, etc), then its filename is shown.

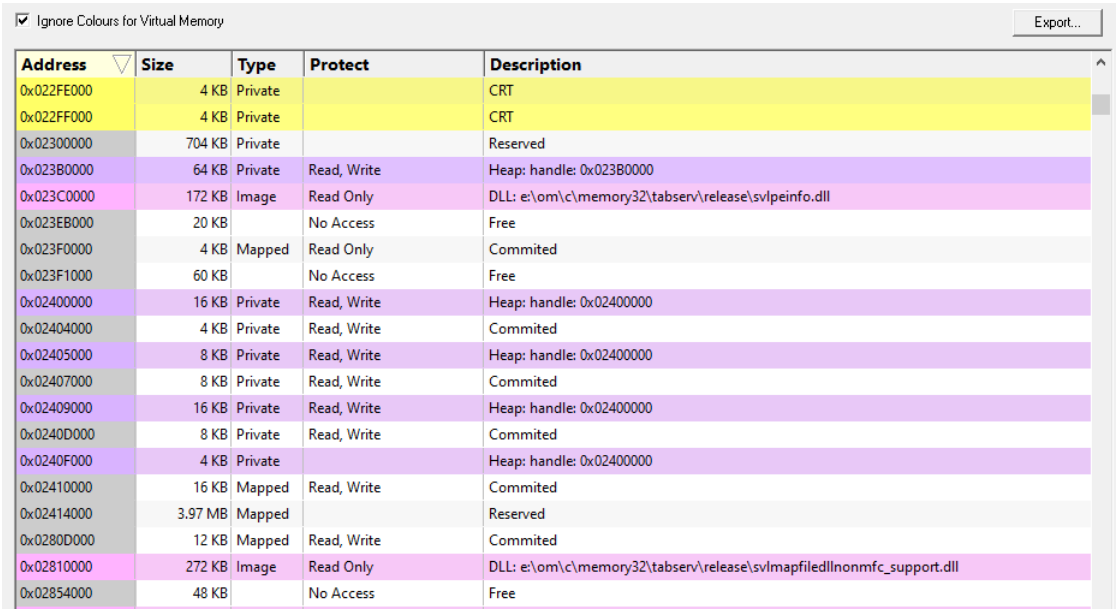
If you're testing this display using the supplied example program, you might want to allocate quite a large block of memory, eg 10Mb or more, to make it significant in the display.

Pages view

The Pages view lists similar data as the graphic view, showing the allocation state of each page and region of memory in the program.

For each area, the address, size, type, protected state and description of the area are given.

Unlike the graphical view, if the target program has exited, this view keeps the last known state



Address	Size	Type	Protect	Description
0x022FE000	4 KB	Private		CRT
0x022FF000	4 KB	Private		CRT
0x02300000	704 KB	Private		Reserved
0x023B0000	64 KB	Private	Read, Write	Heap: handle: 0x023B0000
0x023C0000	172 KB	Image	Read Only	DLL: e:\om\c\memory32\tabserv\release\svlpeinfo.dll
0x023EB000	20 KB		No Access	Free
0x023F0000	4 KB	Mapped	Read Only	Commited
0x023F1000	60 KB		No Access	Free
0x02400000	16 KB	Private	Read, Write	Heap: handle: 0x02400000
0x02404000	4 KB	Private	Read, Write	Commited
0x02405000	8 KB	Private	Read, Write	Heap: handle: 0x02400000
0x02407000	8 KB	Private	Read, Write	Commited
0x02409000	16 KB	Private	Read, Write	Heap: handle: 0x02400000
0x0240D000	8 KB	Private	Read, Write	Commited
0x0240F000	4 KB	Private		Heap: handle: 0x02400000
0x02410000	16 KB	Mapped	Read, Write	Commited
0x02414000	3.97 MB	Mapped		Reserved
0x0280D000	12 KB	Mapped	Read, Write	Commited
0x02810000	272 KB	Image	Read Only	DLL: e:\om\c\memory32\tabserv\release\svlmapfiledllnonmfc_support.dll
0x02854000	48 KB		No Access	Free

A few options are available above the list:

- **Sort** › chooses which column to sort by, the default being to sort by ascending address

  This is exactly equivalent to clicking in the header row of columns in the table itself.

- **Descending** › swaps the ascending or descending ordering of the hotspots
- **Ignore Colours for Virtual Memory** › toggles whether to highlight the areas of memory in the list that are marked as commited, reserved or free
- **Export** › enables the export of virtual memory data in HTML, XML or CSV file formats

  ➡ Read more about exporting virtual memory data and the format of the exported data.

Paragraphs view

The Paragraph view displays very similar data to the previous Pages view, but using larger chunks of memory.

A paragraph is defined by dwAllocationGranularity in the SYSTEM_INFO data structure returned from GetSystemInfo, but is typically 64K.

Virtual tab options

At the far left of the window are the usual options for updating the refresh interval and setting display colours.

Updating the display

- **Update Interval (s)** › automatically updates the display at your choice of interval between 0.1 and 60 seconds, or the default of Never!

  Choose an interval suitable for the size of your application, as lots of data may take longer to draw than the interval itself !

- **Refresh** › updates the display - as does the button on the Tools menu and toolbar

  With an update interval set to Never, you'll need to use this Refresh button to update the display.

Display settings - colours

- **Colours** › shows the Virtual Memory Colours dialog to change the colours used in the views for each type of memory usage above

  Each colour can be changed via the ... button, or all colours can be reset to their defaults.

  Apply › updates the colours in the virtual view without closing the dialog.

Memory usage totals

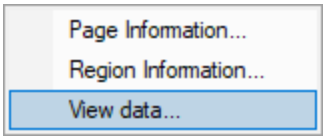
Below the options are the total accumulated amounts for each type of memory usage above.

- **Units** › sets the units for displaying the total values in MB, KB or the number of pages

Virtual view popup menu

Each virtual view has a popup menu giving access to allocations, reallocations and deallocations in a page or region:

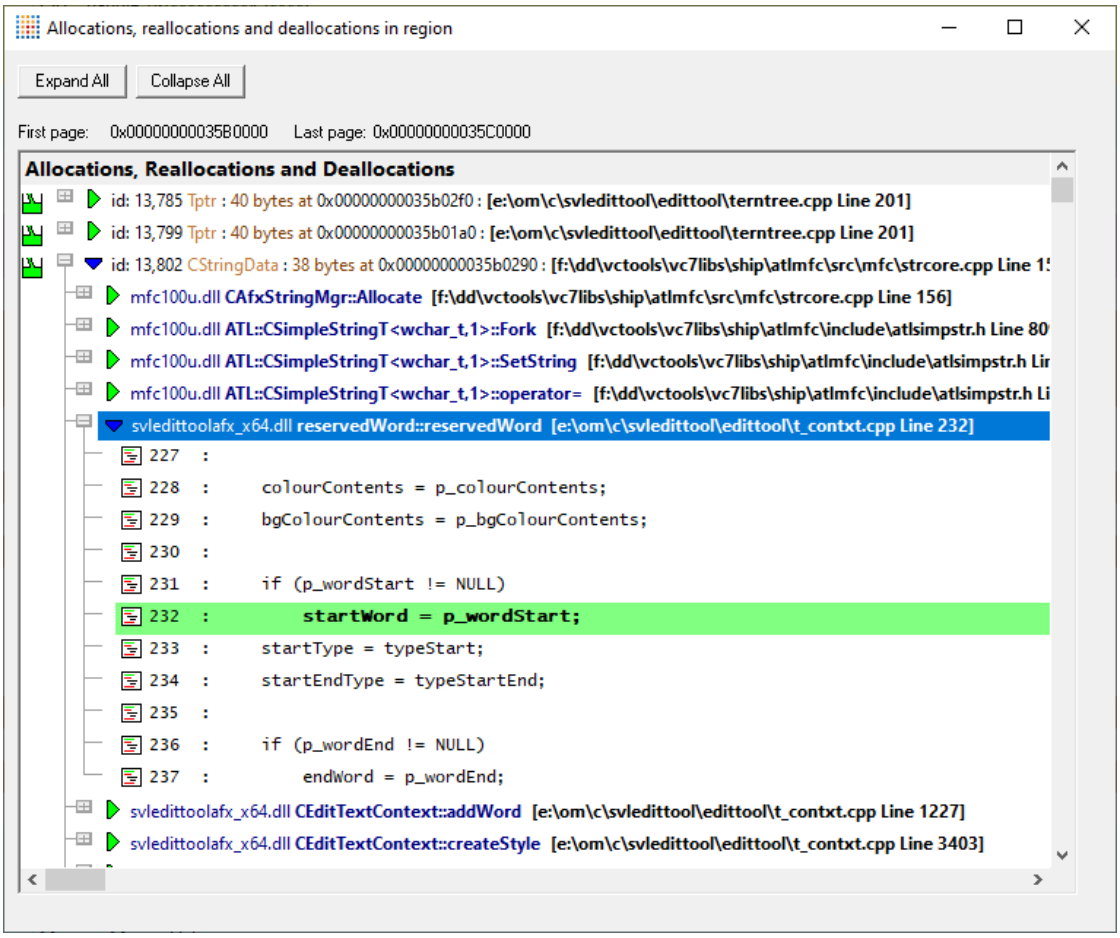
All views give region information and view data, but only the graphical view also gives page information for the point under the cursor.



- **Page information...** › allocations in the selected page (pixel)
- **Region information...** › allocations in the same selected contiguous region of memory with the same type.

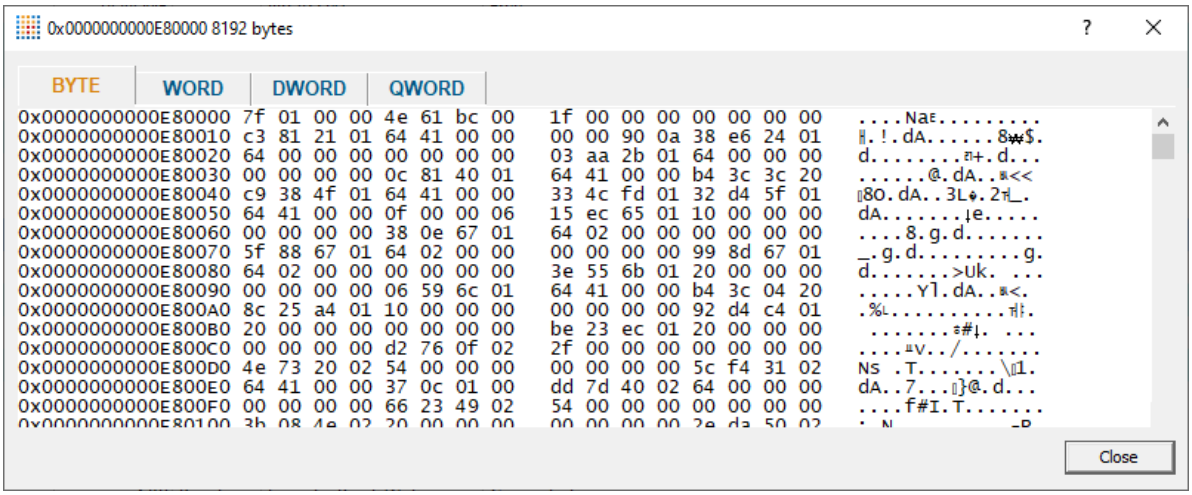
No dialog? Not all pages and regions contain allocations, in which case, no allocation dialog appears, and you'll hear a beep (if you have speakers on!). If your allocations are generally on the CRT heap for example, look for the corresponding coloured areas - the CRT memory usage is yellow by default.

An example of the page information dialog is below. The region information shows the first and last page address accordingly.



The allocations in the dialog can be expanded, to show the local source code, and double clicking launches your preferred editor with the source file.

- **View data... >** shows the bytes for selected memory region, if that region is readable. ***CAUTION, this option can be very slow.***

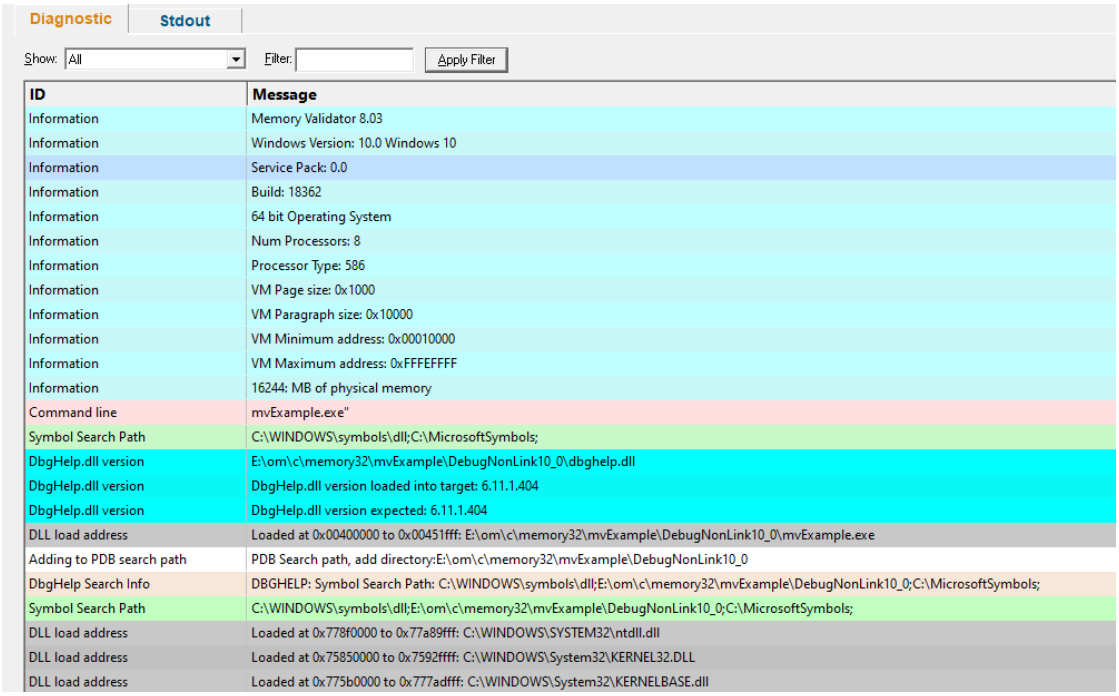


3.7.7 Diagnostic

The **Diagnostic** tab displays information collected by Memory Validator about the target program.

There are two subtabs. One for Diagnostic information and one for displaying any data captured from stdout and stderr.

Diagnostic



Diagnostic information

When Memory Validator's stub is injected into the target program, it logs diagnostic information to the main window for inspection.

Examples of diagnostic data collected are below, and may be displayed with a message, although you may not see some of these if all is well:

Hooking information

- Ordinal hook found
- Hook C++ constructor / destructor
- Function hook success or failure
- Delay loaded function hooked
- Possible hook found
- Function already hooked
- Hook at address

Other information

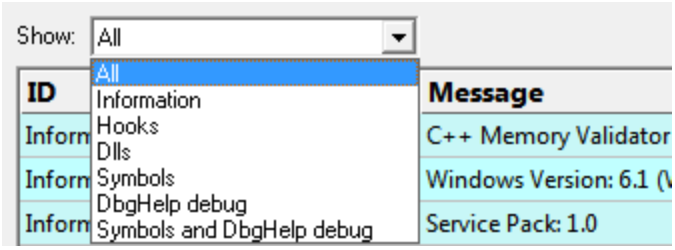
- DLL load address
- DbgHelp searching
- Image source line
- Unknown instruction found
- Disassembly of troublesome code
- Failed to find Release/Debug CRT heap
- Symbol reader status

The locations of loaded DLLs are also displayed in the window for each **LoadLibrary()**, **LoadLibraryEx()** and **FreeLibrary()** in the target program.

If for whatever reason, you don't want to collect diagnostic information, you can switch it off in the Advanced > Symbols Misc page of the settings dialog

Filtering diagnostic information

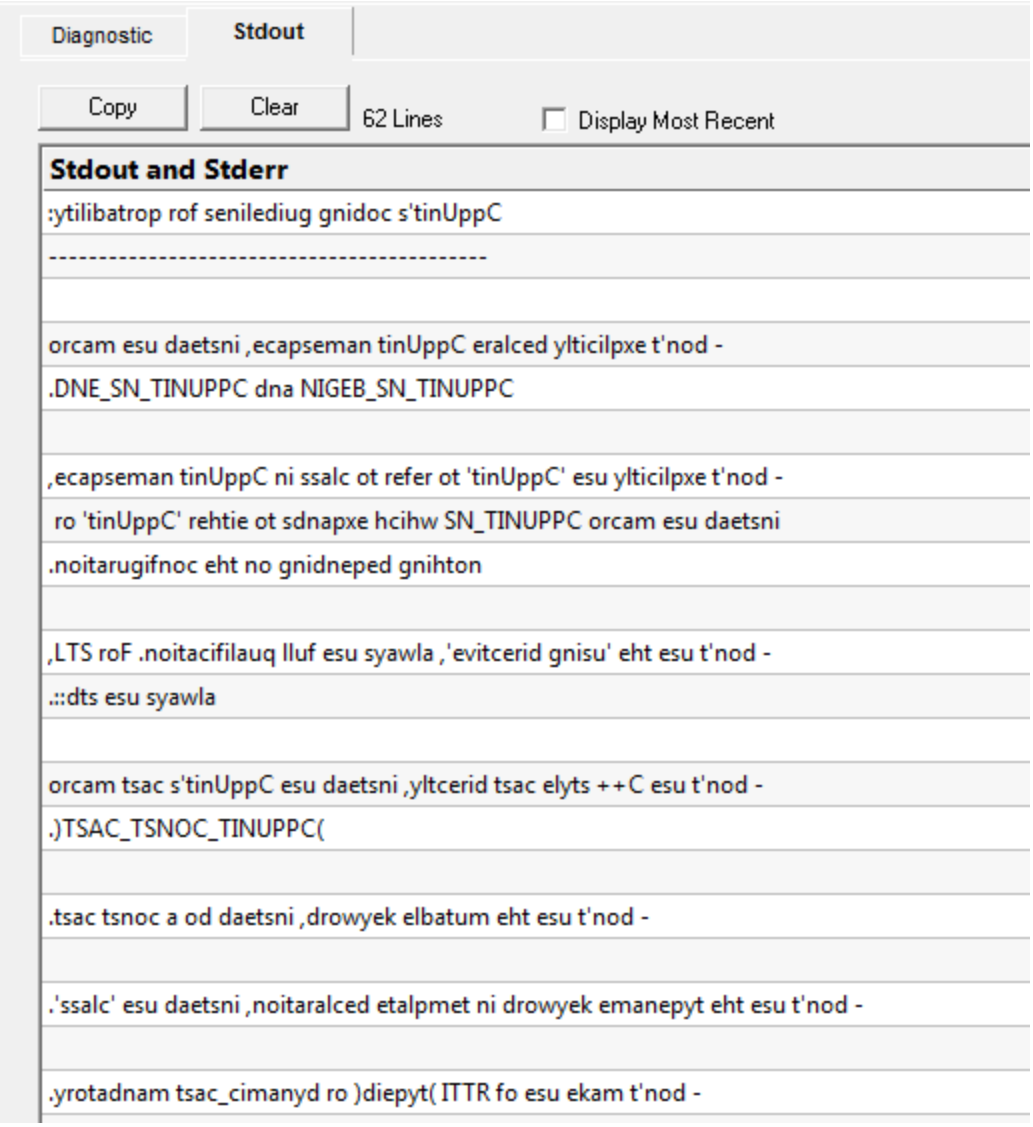
By default, all information is displayed, but you can filter the messages to show only one type:



- **All** > the default
- **Information** > operating system and environment information, etc
- **Hooks** > hooking success and failure messages
- **DLLs** > DLL related information
- **Symbols** > symbol loading status messages
- **DbgHelp debug** > messages from DbgHelp.dll about the DLL symbol search processes
- **Symbols and DbgHelp debug** > both the previous two

When identifying why symbols are not loading, you'll find it's much easier when showing only the DbgHelp debug information.

Stdout and Stderr



The **Stdout** tab displays any data collected from stdout and stderr. ➜ The option to enable this data collection is specified on the launch dialog/wizard.

The above image shows some data collected from a program that reverses the characters in each line passed to it.

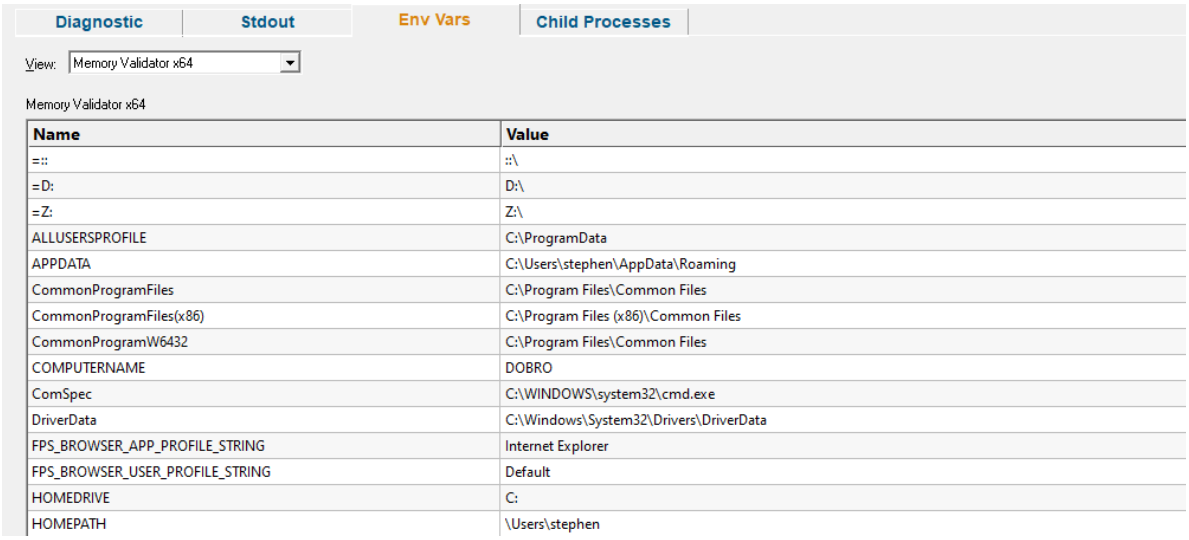
- **Copy** › copy all data from the display on to the clipboard. For large amounts of data this can be time consuming.
- **Clear** › clear the display of any captured data.
- **Display Most Recent** › the display will be scrolled to ensure the most recently captured data is displayed.

Environment Variables

Environment variables tab displays environment variables from Memory Validator, environment variables from the program under test and environment variable substitution errors.

Choose which data you wish to view using the combo box at the top left of the tab.

Memory **Validator environment variables**

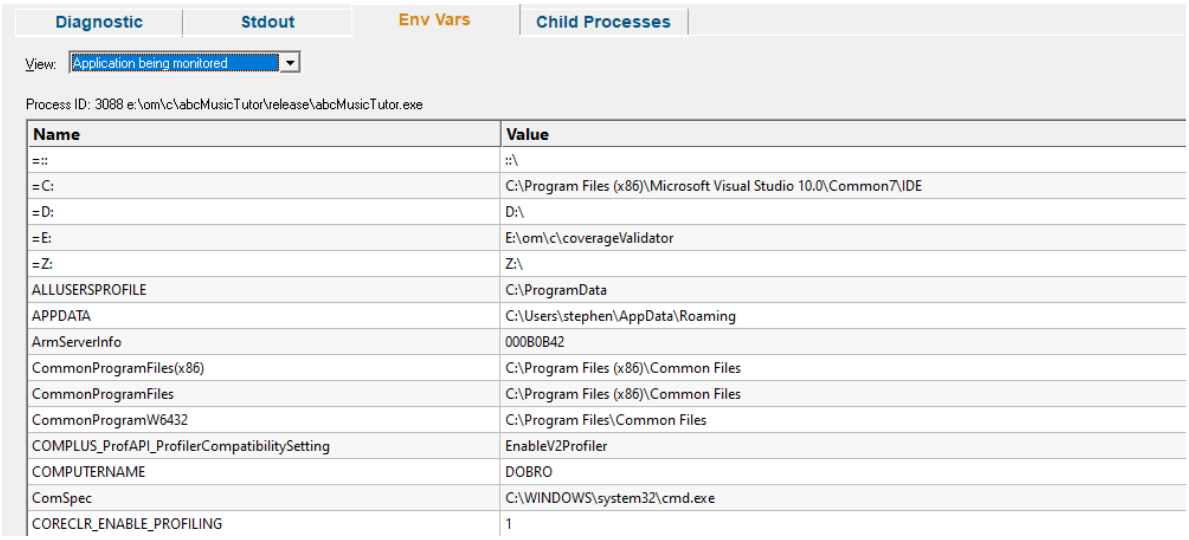


Name	Value
=::	::\
=D:	D:\
=Z:	Z:\
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\stephen\AppData\Roaming
CommonProgramFiles	C:\Program Files\Common Files
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPUTERNAME	DOBRO
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
FPS_BROWSER_APP_PROFILE_STRING	Internet Explorer
FPS_BROWSER_USER_PROFILE_STRING	Default
HOMEDRIVE	C:
HOMEPATH	\Users\stephen

Target application environment variables

If you launched the target application from Memory Validator the target application's environment variables will be similar to those in Memory Validator, but with some additional env vars to control .Net profilers and and some other SVL_ prefixed env vars to communicate various data to Software Verify components that are loaded.

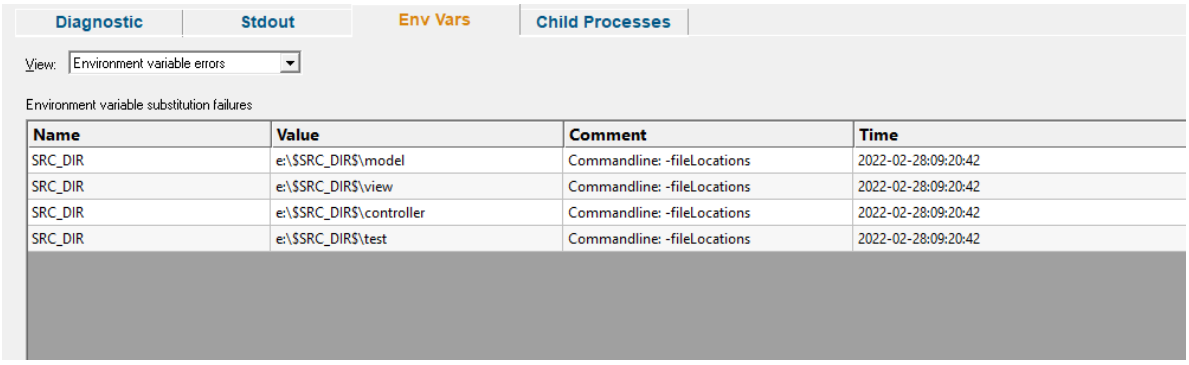
If you launched the target application as a standalone application, or service and used one of our APIs to connect to Memory Validator, the environment variables shown will reflect those in force at the time the application/service was started, and the account that application/service is running on.



Name	Value
=::	::\
=C:	C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE
=D:	D:\
=E:	E:\om\c\coverageValidator
=Z:	Z:\
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\stephen\AppData\Roaming
ArmServerInfo	000B0B42
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files
CommonProgramFiles	C:\Program Files (x86)\Common Files
CommonProgramW6432	C:\Program Files\Common Files
COMPLUS_ProfAPI_ProfilerCompatibilitySetting	EnableV2Profiler
COMPUTERNAME	DOBRO
ComSpec	C:\WINDOWS\system32\cmd.exe
CORECLR_ENABLE_PROFILING	1

Environment variable errors

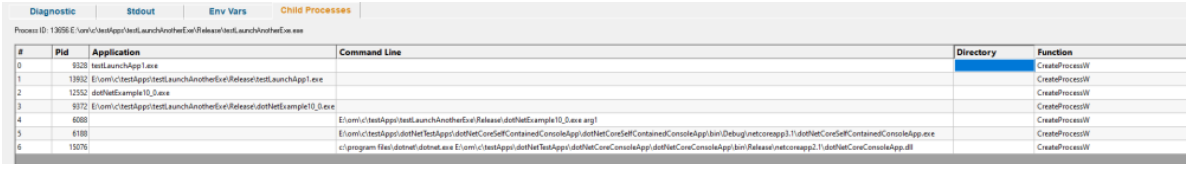
The environment variable errors display shows the name of the environment variable that could not be found, the string containing the environment variable, a comment indicating where the string came from (in this example, the command line), and a timestamp.



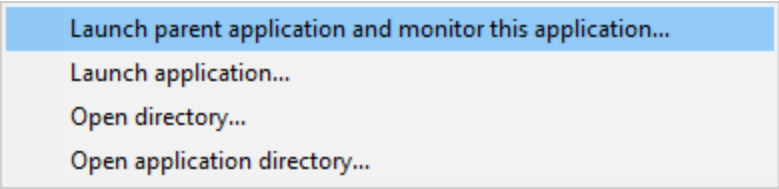
Name	Value	Comment	Time
SRC_DIR	e:\SRC_DIR\model	Commandline: -fileLocations	2022-02-28:09:20:42
SRC_DIR	e:\SRC_DIR\view	Commandline: -fileLocations	2022-02-28:09:20:42
SRC_DIR	e:\SRC_DIR\controller	Commandline: -fileLocations	2022-02-28:09:20:42
SRC_DIR	e:\SRC_DIR\test	Commandline: -fileLocations	2022-02-28:09:20:42

Child Processes

Information about child processes, and the appropriate launch parameters passed to CreateProcess (and related functions) are displayed on this tab.



A context menu is provided to allow you to perform some actions with the launched application data.

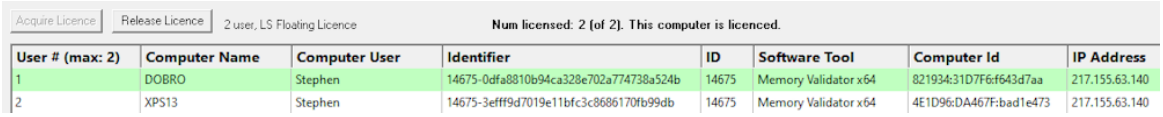


- **Launch parent application and monitor this application...** › the launch application dialog is displayed configured to launch the parent application and monitor this application
- **Launch application...** › the launch application dialog is displayed configured to launch and monitor this application
- **Open directory...** › Windows Explorer is launched to view the contents of the launch directory (the directory field is empty nothing will be shown)
- **Open application directory...** › Windows Explorer is launched to view the directory that contains the application (if the application specification has no path nothing will be shown)

3.7.8 Floating Licence

The **Floating Licence** view displays information about the computers using the floating licence.

This view is only displayed if a floating licence has been purchased. Evaluation users will not see this view.


Acquire Licence | Release Licence | 2 user, LS Floating Licence | Num licensed: 2 (of 2). This computer is licenced.

User # (max: 2)	Computer Name	Computer User	Identifier	ID	Software Tool	Computer Id	IP Address
1	DOBRO	Stephen	14675-0dfa8810b94ca328e702a774738a524b	14675	Memory Validator x64	821934:31D7F6:f643d7aa	217.155.63.140
2	XPS13	Stephen	14675-3efff9d7019e11bfc3c8686170fb99db	14675	Memory Validator x64	4E1D96:DA467F:bad1e473	217.155.63.140

The screenshot above show two computers using the same 2 user floating licence, that has maintenance id 14675. Both computer users are licenced and can use the software.

On startup the software automatically checks to see if a floating licence is available, and acquires the licence if possible. This takes a few seconds to process, after startup of the software.

Global Floating Licences

An internet connection is required for floating licences to work. The licence server is managed and run by Software Verify.

Local Network Floating Licences

An internet connection is not required for local network floating licences to work. No licence server is required. The licences are automatically managed by the computers on the local network.

Licence information

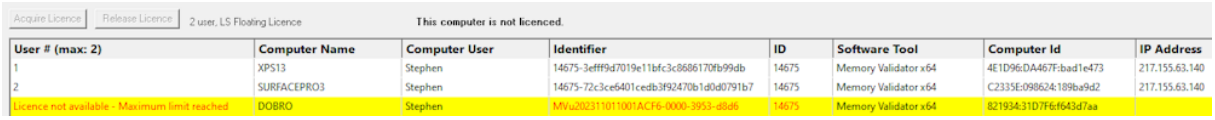
The information show in this display allows you to identify which of your colleagues are using the software and which versions of the software are in use.

- User

The user id (1 to number of licensed users).

- Computer Name
  The name of the computer
- Computer User
  The login name of the user of the computer.
- Identifier
  The unique identifier for this licence, used on the licence server.
- ID
  The maintenance id for the software.
- Software Tool
  The software tool and version of the software that is running on that computer.
- Computer ID
  The unique id for this computer.
- IP Address
  This computer's IP address.

Unlicenced users


Acquire Licence | Release Licence | 2 user, LS Floating Licence | This computer is not licenced.

User # (max: 2)	Computer Name	Computer User	Identifier	ID	Software Tool	Computer Id	IP Address
1	XPS13	Stephen	14675-3efff9d7019e11bfc3c8686170fb99db	14675	Memory Validator x64	4E1D96:DA467F:bad1e473	217.155.63.140
2	SURFACEPRO3	Stephen	14675-72c3ce6401cedb3f92470b1d0d0791b7	14675	Memory Validator x64	C2335E:098624:189ba9d2	217.155.63.140
Licence not available - Maximum limit reached	DOBRO	Stephen	MVu20231101001ACF6-0000-3953-d8d6	14675	Memory Validator x64	821934:31D7F6:f643d7aa	

If any additional users are trying to get a licence for the software, but there are not enough licences, they will also be shown in the display, but with red text on a yellow background.

Please note that on the machine of an unlicensed user the status information will be different.

The software checks to see if a licence has been released on a periodic basis, so that if a licence is released by another user, it can be acquired by the next waiting user.

Releasing a licence

If you have finished using a licence and wish to let a team mate use the software, you have two choices.

You can close Memory Validator, releasing the licence as it closes.

Or you can keep Memory Validator running by manually releasing the licence. Do this by clicking the **Release Licence** button.

Acquiring a licence

If you have released a licence you will need to actively reclaim a licence when you wish to use Memory Validator again. You can start this procedure by clicking the **Acquire Licence** button.

3.8 User Interface Mode

Setting the user interface mode - Wizards or Dialogs?

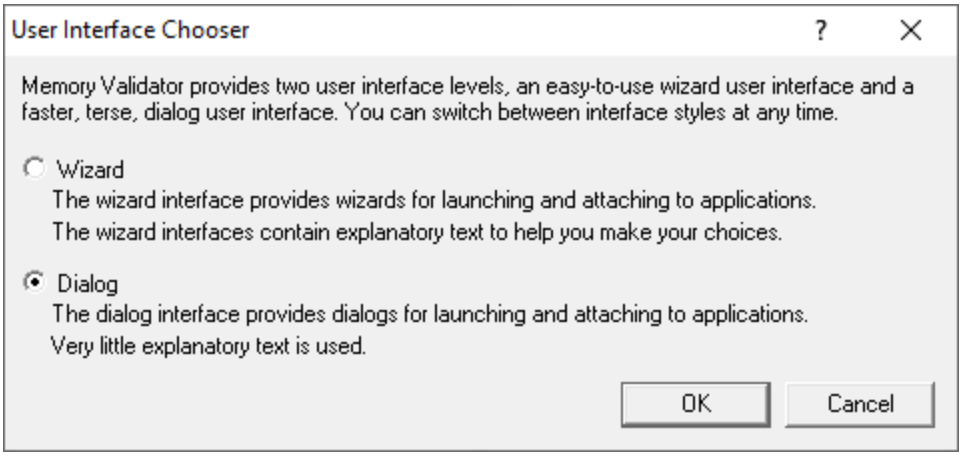
For some key tasks, there are two user interface modes controlling the way in which options are presented to you:

- **Wizard mode** › guides you through the tasks in a linear fashion
- **Dialog mode** › all options are contained in a single dialog

  Experienced users will find this mode quicker to use

To set the user interface mode

Settings menu › **User Interface Mode...** › select the desired mode in the **User Interface Chooser** dialog:



The user interface mode affects the following tasks:

- Attaching to an application (Injection)
- Launching an application
- Wait for application to start

3.9 UX Theme

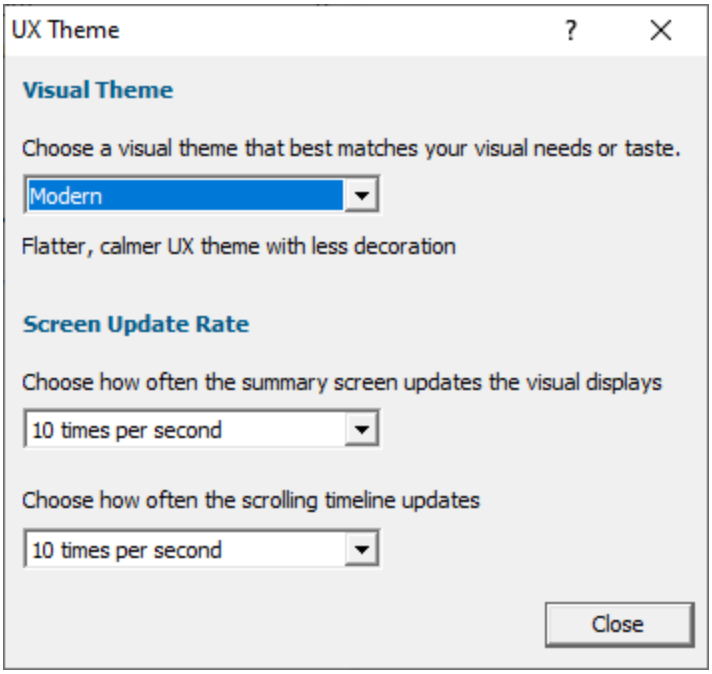
The user interface provides three UX themes.

- Modern. The look and feel of current Software Verify tools.
- Classic. The look and feel of previous Software Verify tools.
- High Contrast. A higher contrast version of the Modern theme.

Setting the UX theme

To set the UX theme

Settings menu › **UX Theme...** › shows the UX Theme chooser dialog



Changing the UX theme will update some of the colours that you can modify with the colour settings dialog.

Setting the Screen Update Rate

The default screen update rate for the Summary view is 10Hz for all graphics and text.

You can change this update to 1Hz, 2Hz, 3Hz, 4Hz, 4Hz, 10Hz using the combo boxes for the statistics panels and the scrolling timelines.

3.10 Summary Display Layout

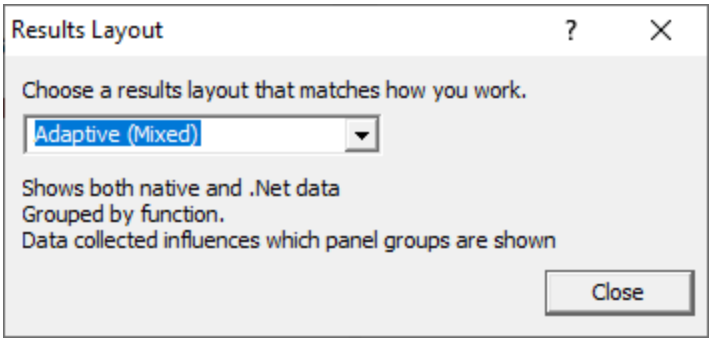
The summary display provides several different layout methods

- Adaptive (Mixed)
- Adaptive (Layered)
- Native and .Net (Mixed)
- Native and .Net (Layered)
- Native only
- .Net only (minimal set of panels)
- .Net only (all .Net related panels)

Setting the summary display layout

To set the Summary display layout

Settings menu › **Summary Display Layout...** › shows the Summary display layout chooser dialog

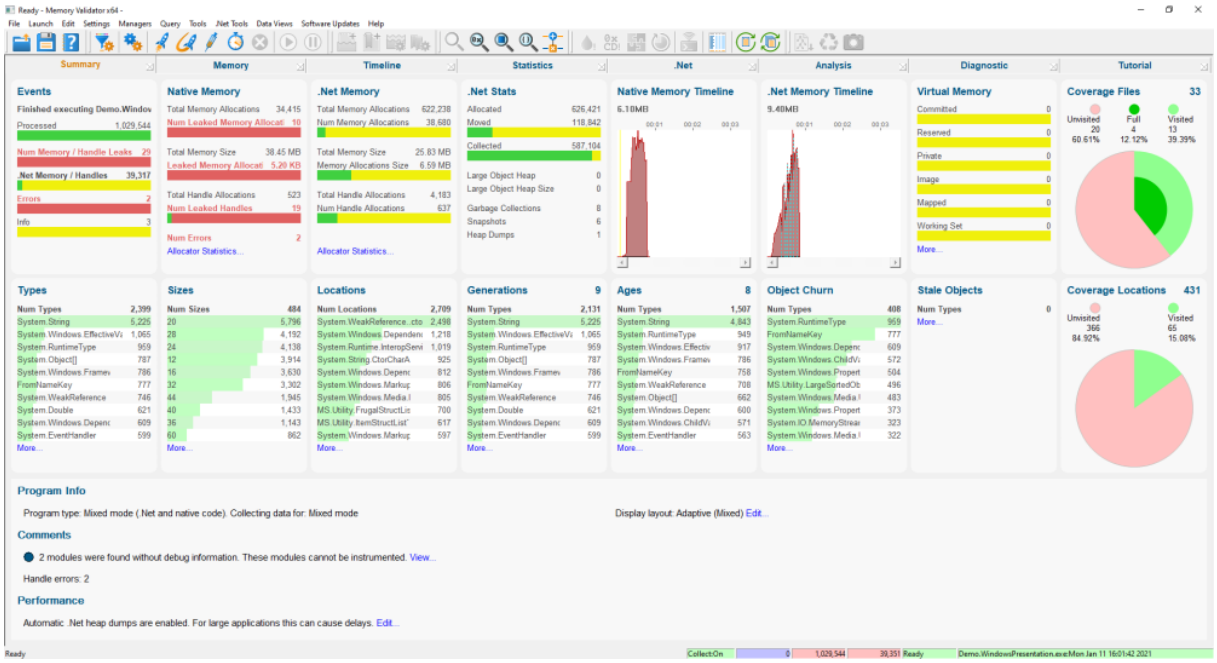


Changing the layout changes how the summary display arranges it's panels.

Two of the layouts are adaptive and change in response to the data collected and also to the data collection settings (when there is no data collected).

Examples of each layout are shown below.

Adaptive (Mixed)

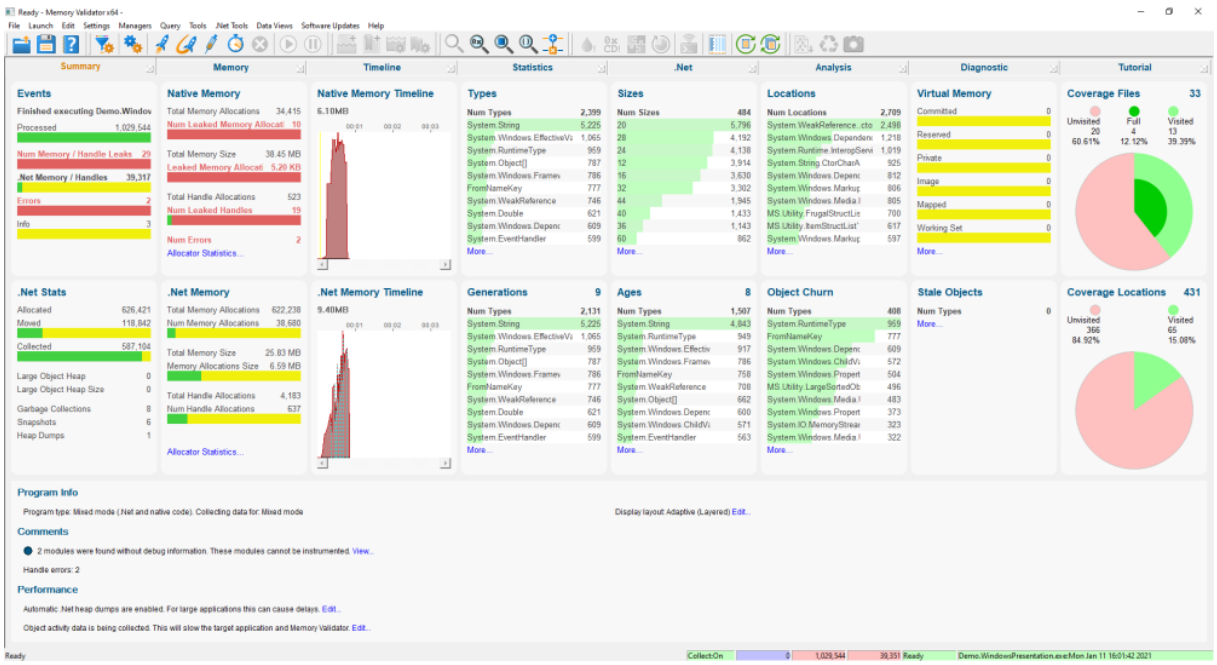


The display shows all native and .net panels, arranged so that all the statistics groups are on the bottom layer and all other statistics are on the top layer.

For mixed mode applications, both native and .Net panels are shown.

For native applications only native panels are shown. For .Net applications only .Net panels are shown.

Adaptive (Layered)

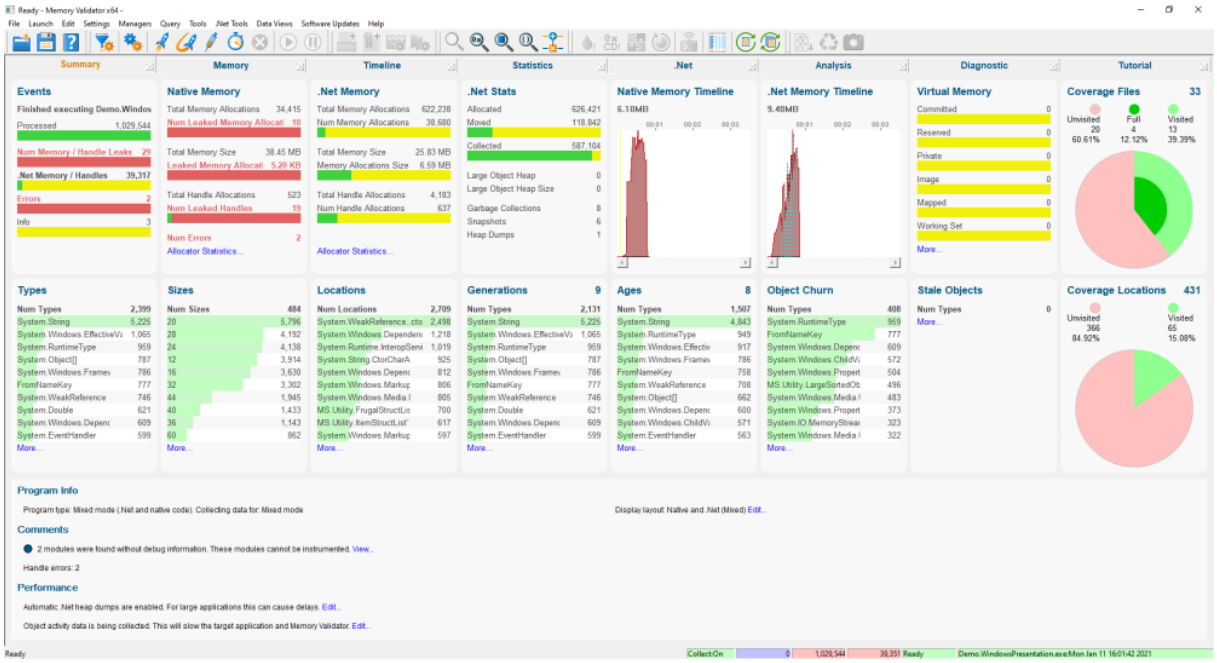


The display shows all native and .net panels, arranged so that all native statistics are on the top layer and all .Net statistics are on the bottom layer.

For mixed mode applications, both native and .Net panels are shown.

For native applications only native panels are shown. For .Net applications only .Net panels are shown.

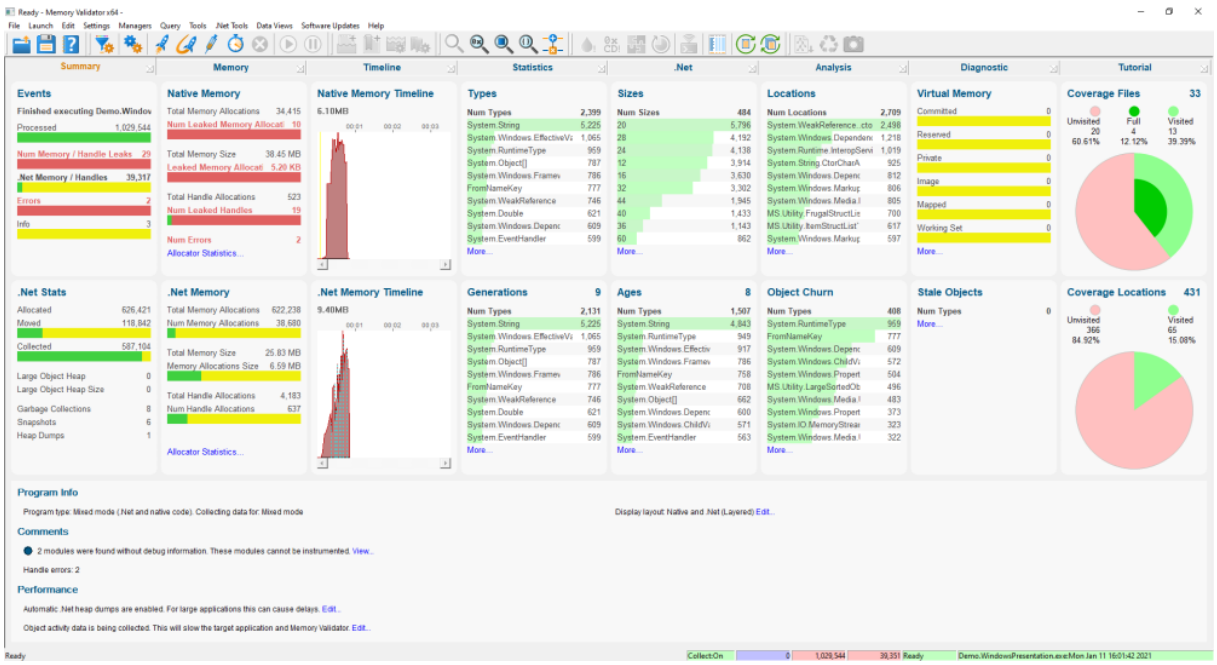
Native and .Net (Mixed)



The display shows all native and .net panels, arranged so that all the statistics groups are on the bottom layer and all other statistics are on the top layer.

Types, Sizes and Locations apply to both native data and .Net data.

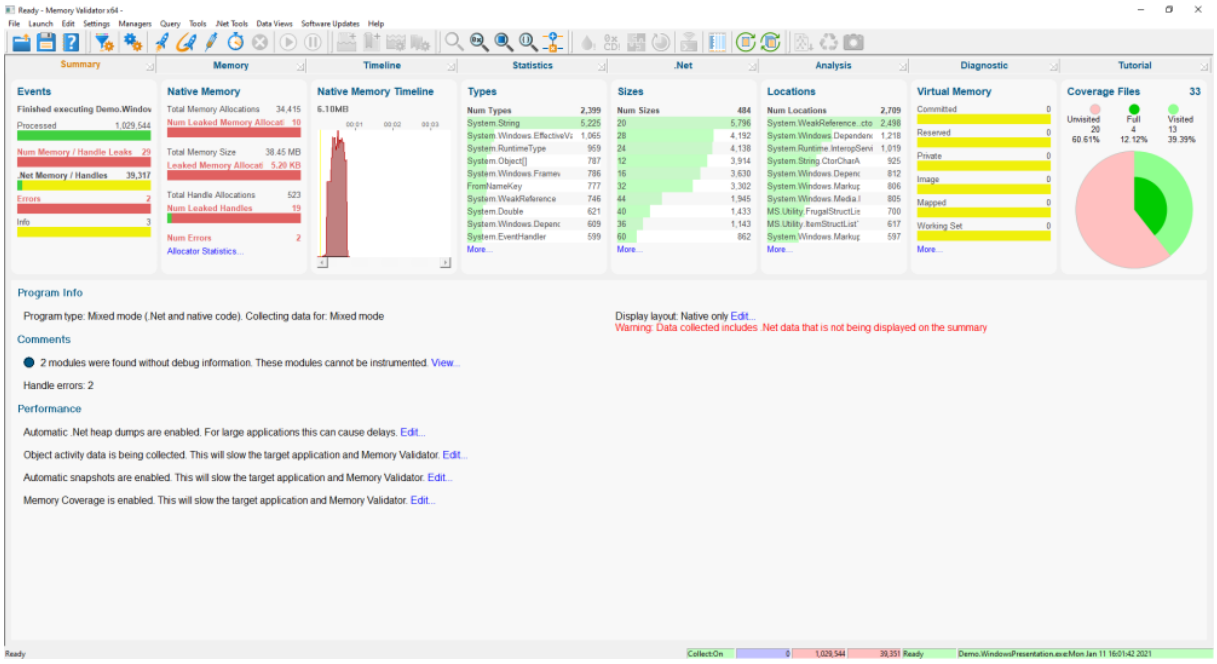
Native and .Net (Layered)



The display shows all native and .net panels, arranged so that all native statistics are on the top layer and all .Net statistics are on the bottom layer.

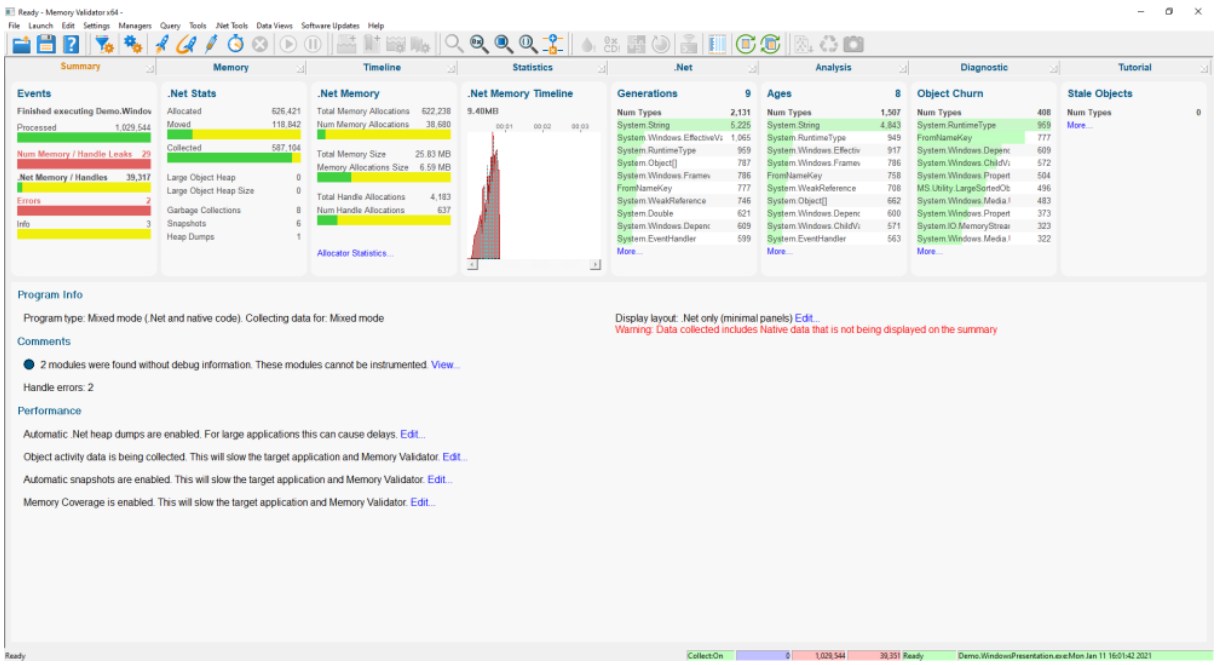
Types, Sizes and Locations apply to both native data and .Net data.

Native only



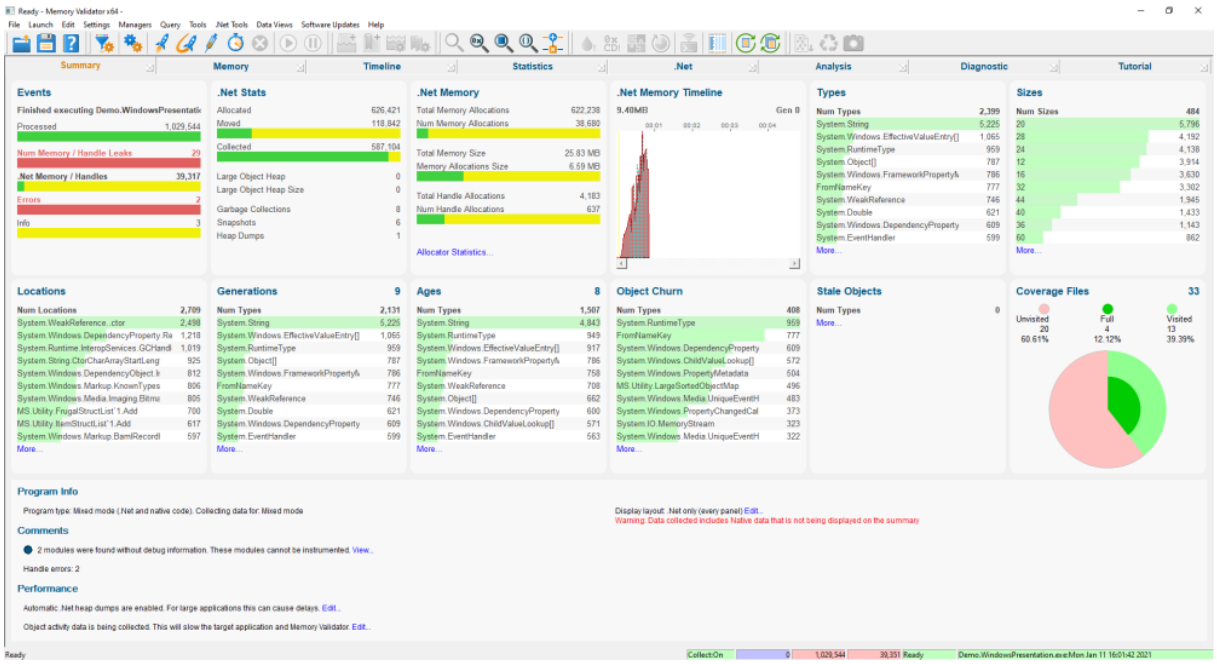
The display shows only native panels.

.Net only (minimal set of panels)



The display shows only .Net panels. A minimal selection of panels has been selected so that they fit on one line.

.Net only (all .Net related panels)



The display shows only .Net panels. All relevant .Net panels are shown, resulting in a two line display.

3.11 Delete Cache Files

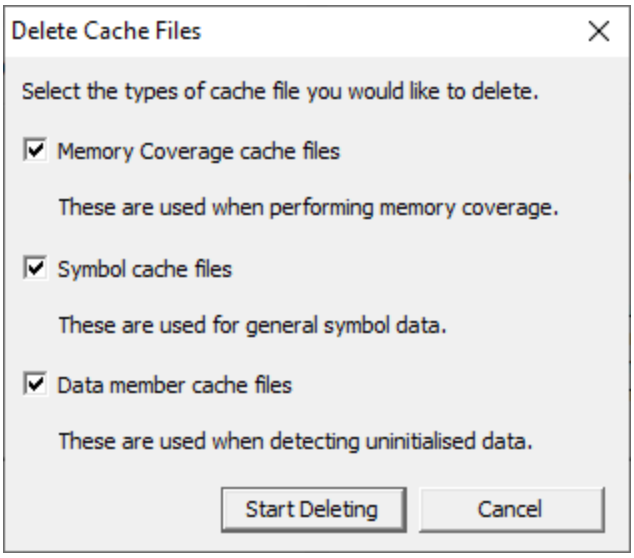
The Delete Cache Files dialog provides a one-stop-shop for deleting cache files used by Memory Validator.

This is simpler and easier to use than finding each of the various cache control settings and deleting each cache from there.

Deleting cache files

To delete the cache files

Settings menu › **Delete Cache Files...** › shows the Delete Cache Files dialog



- **Memory coverage cache files** › Cache files used for memory coverage
- **Symbol cache files** › Cache files used for symbol names
- **Data member cache files** › Cache files used for C++ object data member names and types

3.12 Settings

Memory Validator allows extensive control over which data is collected and how that data is displayed. Additional options control the way the application behaves.

The settings can generally be grouped as either global or local.

Global and local settings

Global settings affect all data collected and its display throughout Memory Validator. Global settings are changed via the Settings Dialog and the following 40 or so pages describe each available group of settings.

Local settings apply to controls and data displayed in each of the main display windows.

Local settings are found on the left side of each relevant tab:

- Memory display settings
- Types display settings
- Sizes display settings
- Hotspots display settings
- Analysis display settings

Other settings

There are a few more settings not included in the global settings dialog such as:

- User permissions warnings
- User interface mode
- Session settings

3.12.1 Global Settings Dialog

The settings dialog allows you to control all the global settings in Memory Validator that affect the way data is collected and displayed. There are also local settings on each main tab.

This page has a warning about use of the **Reset** button.

Opening the settings dialog

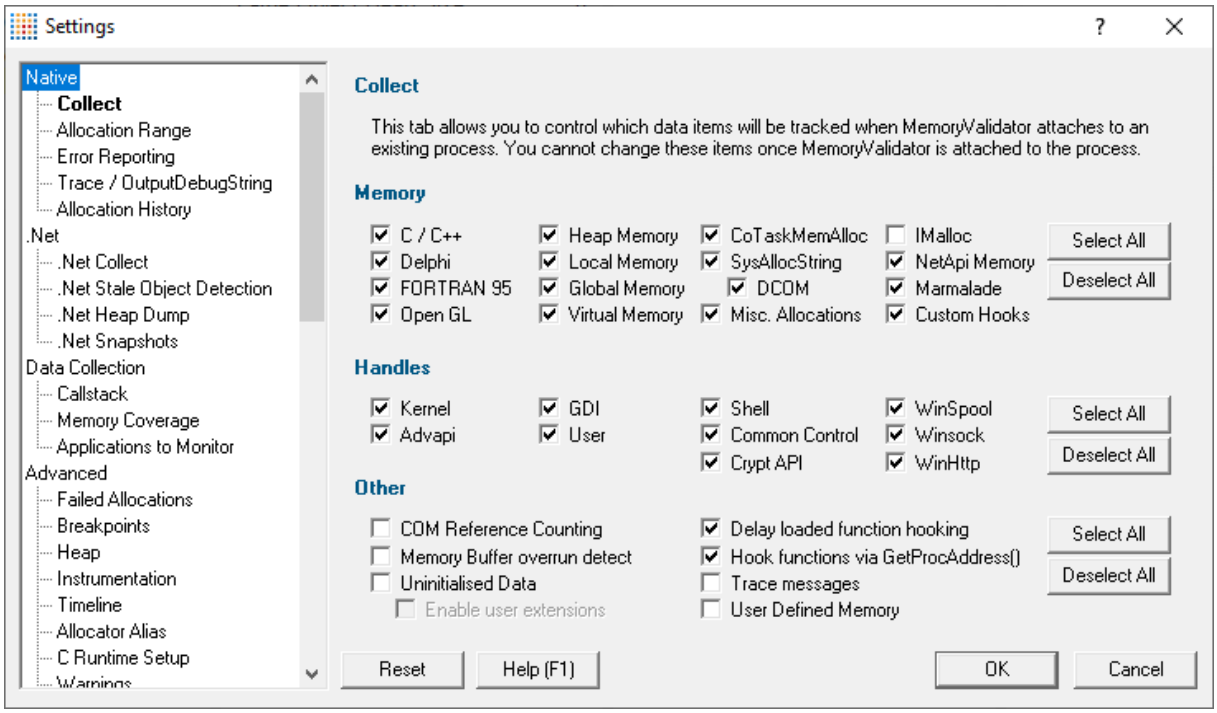
To view the settings dialog, choose **Settings** menu › **Edit Settings...**

Or use the option on the Session Toolbar:

Using the settings dialog

The dialog has a scrolled list on the left hand side, grouping the topics. When a topic is clicked, its related controls are displayed on the right hand side.

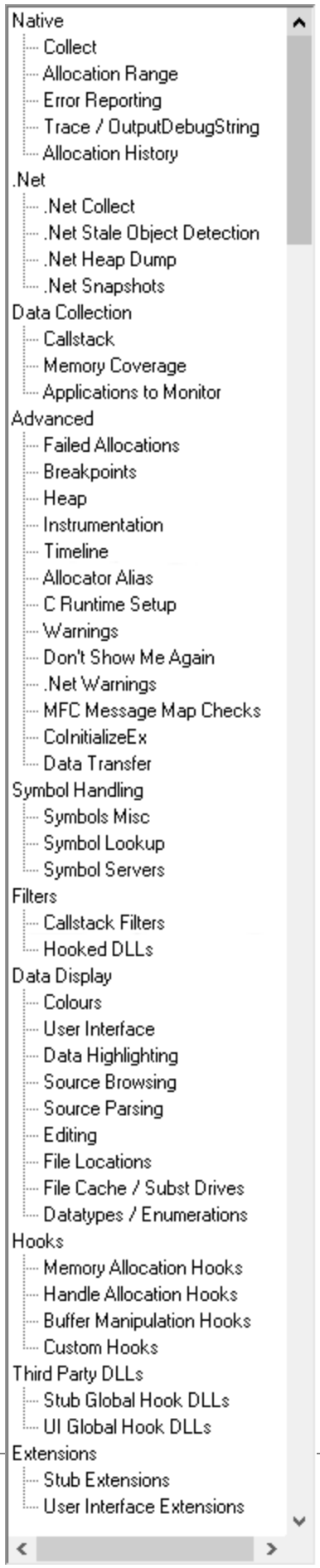
The default display of the dialog is shown below with the first topic selected.



After selecting a topic, you can also use the cursor up and down arrow keys to change the selected item

Too many settings? It may seem that there is an overwhelming number of settings to worry about. Don't panic! The good news is that for new users, very few (if any) settings actually need to be changed to use the application in most cases, and even for experienced users, many groups of settings will not be needed. However, Memory Validator remains flexible for all our users in many different scenarios.

Click on any item in the picture below to find out more about the settings for that group.



Restoring the default settings

The settings dialog has **Reset All** and **Reset** buttons near the bottom left of the dialog which you can use to reset all global settings back to their default values.

The **Reset All** button resets **all global settings** in Memory Validator, not just the settings visible on the current tab of the dialog.

The **Reset** button resets just the settings visible on the current tab of the dialog.

3.12.1.1 Native

Enter topic text here.

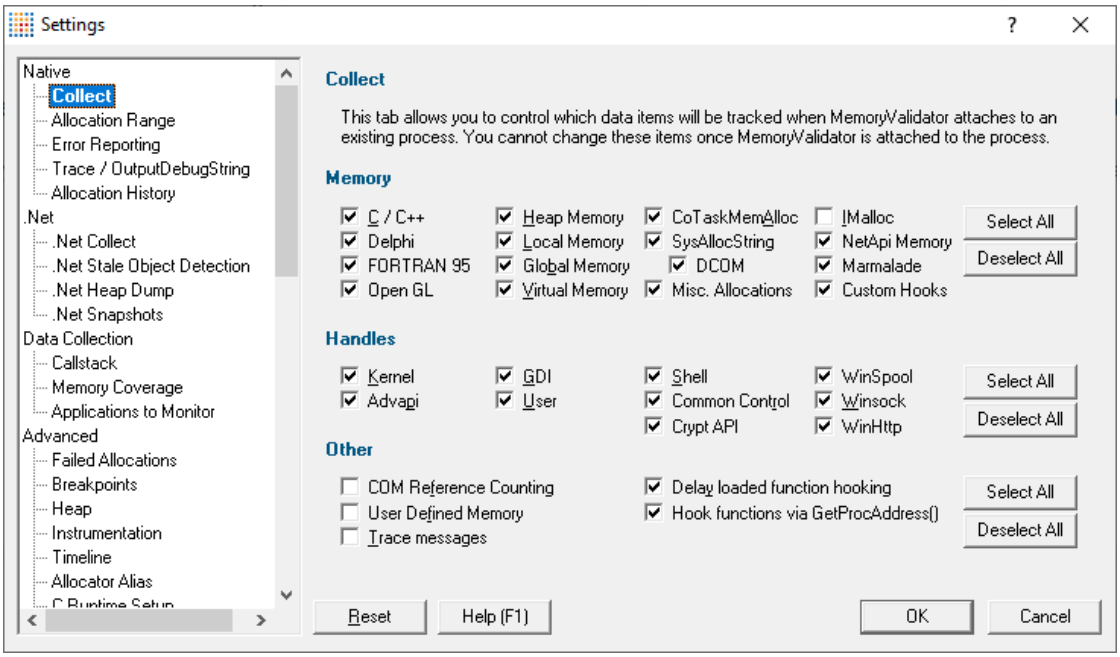
3.12.1.1.1 Collect

The **Collect** tab allows you to specify which groups of hooks are installed in the target program when a session starts.

Changing the hooks that will be installed

Once installed, the group of installed hooks cannot be changed, so changing these settings is only effective on the *next* session with the target program.

The following image shows all the *default* options:



Hooks specified here will be overridden by inclusion in the process modules list of the Hooked DLLs settings.

Each of the **Memory**, **Handles** and **Other** checkboxes includes or excludes the relevant hooks for tracking the following items

Memory hooks

- **C / C++** › install hooks for C runtime heap
- **Delphi** › Delphi runtime heap
- **Fortran 95** › Fortran runtime heap
- **Open GL** › Open GL handles

- **Heap Memory** › install hooks for `HeapAlloc, HeapRealloc, HeapFree` functions
- **Local Memory** › `LocalAlloc, LocalReAlloc, LocalFree` functions
- **Global Memory** › `GlobalAlloc, GlobalReAlloc, GlobalFree` functions
- **Virtual Memory** › `VirtualAlloc, VirtualFree, VirtualAllocEx, VirtualFreeEx` groups of allocation functions

⊟ Using VirtualAlloc(Ex)?

Note that your call to VirtualAlloc/VirtualAllocEx should include the flag MEM_RESERVE or MEM_COMMIT.

If including MEM_COMMIT without MEM_RESERVE, either:

- the first page of the proposed address range should be non-reserved

  or

- your proposed address should be NULL allowing the operating system to choose the address for you

- **CoTaskMemAlloc** › `CoTaskMemAlloc` group of allocation functions

  The preferred way of allocating memory in COM objects.

- **SysAllocString** › `SysAllocString` group of allocation functions

  Allocates, reallocates and deallocates BSTR objects (OLE Strings).

- **DCOM** › an additional group of `SysAllocString` hooks are inserted for use in DCOM transactions

  Only available when `SysAllocString` is selected above, and not required for most applications.

- **Misc. Allocations** › miscellaneous allocation functions

- **IMalloc** › `IMalloc` functions

  The old way of allocating memory in COM objects, prior to the introduction of `CoTaskMemAlloc`.

  See also, notes about IMallocSpy below.

- **NetApi Memory** › `NetApi` allocation functions

- **Marmalade** › `Marmalade` game SDK allocation functions

  `s3eMalloc, s3eRealloc, s3eFree, s3eBaseMalloc, s3eBaseRealloc, s3eBaseFree`

- **Custom Hooks** › hooks for allocation functions tracking custom memory

Default settings are with all memory hooks selected, except for IMalloc.

Marmalade?

For more information about working with Marmalade read Working with Marmalade game SDK.

IMallocSpy?

We do not recommend using the IMallocSpy option.

We have noticed that some functions (`SysAllocString`) cause many allocations to be reported by IMallocSpy, and yet the equivalent deallocation function `SysFreeString` does not cause the same allocations to be reported as deallocated by the IMallocSpy interface. This results in Memory

Validator displaying misleading information. We recommend that the memory hooks for **CoTaskMemAlloc**, **SysAllocString** and **Misc Allocations** are used rather than **IMallocSpy**.

Handle hooks

- **Kernel** › install hooks for handles allocated in `Kernel32.dll`
- **Advapi** › handles allocated in `Advapi32.dll`

- **GDI** › handles allocated in `gdi32.dll`
- **User** › handles allocated in `user32.dll`

- **Shell** › handles allocated in `shell32.dll`
- **Common Control** › handles allocated in `comctl32.dll`

- **Winspool** › printer spool handles
- **Winsock** › socket handles
- **WinHttp** › http handles

Default settings are with all handle hooks selected.

Other hooks

- **COM Reference Counting** › install hooks for tracking COM object creation and reference count tracking.

  See section below regarding possible erratic behaviour when using COM reference counting hooks.

- **Trace Messages** › tracking `TRACE()` messages and `OutputDebugString()`

- **Delay loaded function hooking** › functions which are delay loaded will be hooked as the functions are resolved by the delay loading process

  ▣ What is 'delay loading'?

  Delay loading a DLL is when it it is implicitly linked, but not actually loaded until your code references a symbol contained in the DLL.

  Delay loading can speed up startup time, but unhandled exceptions may cause your program to terminate if the DLL can't be found when needed during the run time.

  Not all DLLs are delay loaded, but typical examples are `COMCTL32.DLL` and some COM libraries.

  If this option is not selected then any functions that would *normally* be hooked, but which have been delay loaded, will end up not being hooked.

- **Hook functions via GetProcAddress()** › calls that are made to functions via a pointer obtained from `GetProcAddress()` will be hooked.

- **User Defined Memory** › calls to the Memory Validator API to track user defined calls will be allowed to send data to the user interface

Default settings are with *only* Delay loaded function hooking enabled.

GetProcAddress() Hooking?

If a function that is looked up via `GetProcAddress()` is also a function that would normally be hooked via Import Address Tables, turning this option on will mean that that function will be hooked even if it is called via pointer returned from `GetProcAddress()`. This is typical source of unreported leaks in many memory leak detection tools, they don't detect calls via pointers from `GetProcAddress()`.

COM Reference Counting?

Selecting **COM Reference Counting** can cause erratic program behaviour if a hook gets inserted incorrectly.

This is because the COM hooks alter the machine code of the program and can get confused by the program structure. This is not typical of *user written* COM objects, but is typical of Microsoft® COM objects which appear to share entry and/or exit points between common functions.

Random crashes: If a program crashes randomly at start up, or seems to get stuck in an infinite loop (cpu at 99%) try disabling COM Reference Counting and check to see if the program behaves normally.

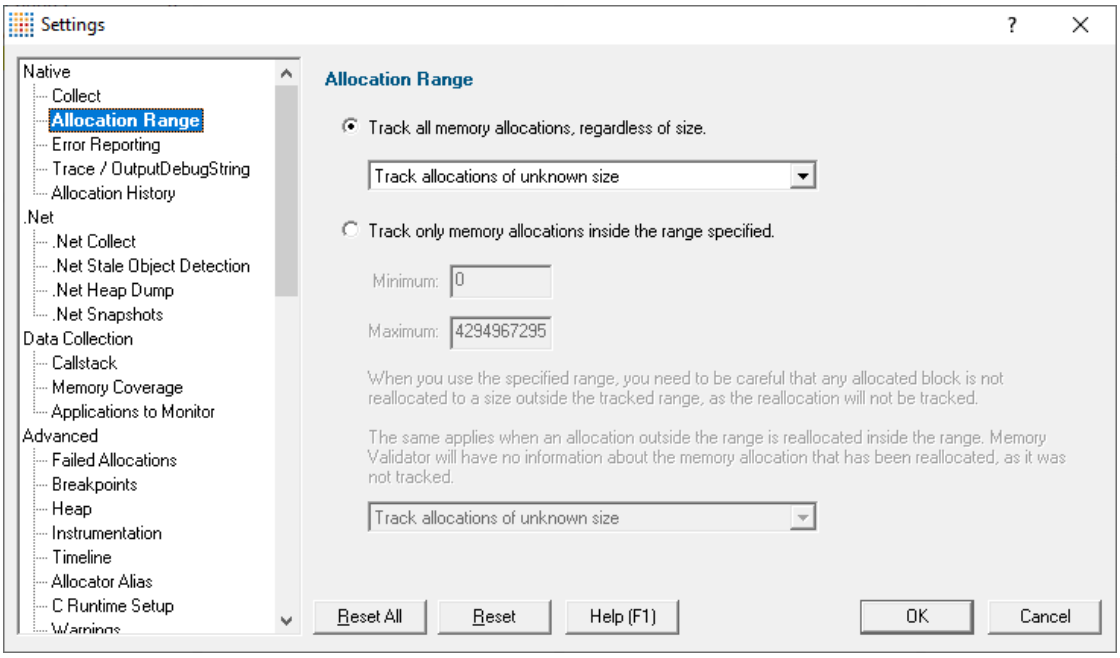
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.1.2 Allocation Range

The **Allocation Range** tab allows you to restrict Memory Validator to only tracking allocations in a specified size range.

This can be very useful for identifying specific leaks in an application that otherwise makes a large number of allocations.



Allocation size range

The default option is **Track all memory allocations, regardless of size.**

Alternatively you can restrict collection according to the size of the allocation:

- Choose **Track only memory allocations inside the range specified** › Enter **Minimum** and **Maximum** sizes › enables the collection of memory allocation data based on allocation size

Caveats of restricting collection of data by size range

If you enable collection of data for a range of data sizes, it is possible for memory allocations that are reallocated, that the allocation, reallocation or deallocation of that memory size may not be recorded by Memory Validator. This is because the size of the allocation may be too small, even though a previous or subsequent allocation/reallocation *is* in range.

This can cause incorrect memory leak reports, please bear this in mind if you choose to use this option.

If your program never using realloc(), _expand(), HeapReAlloc() and so forth, this should not be an issue.

Allocations of unknown size

Whether you are collecting all allocations, or only allocations in a specific size range you need to decide how to handle allocations that have an unknown size.

Some allocations have an unknown size because they are returned from allocators where the size of the object is not disclosed (it's not an input parameter to the function), or where the function hook is unable

to acquire the size data. The are not many functions that fall into this category, but it is important to mention them as ignoring such allocations may result in allocation data not being recorded.

- **Don't Track allocations of unknown size** › memory allocation data will not be collected when the allocation size is not known
- **Track allocations of unknown size** › memory allocation data will be collected when the allocation size is not known
- **Convert allocations of unknown size to a known size** › memory allocations that have an unknown size will have a synthetic size calculated for them

  This option is provided so that unknown size allocations can be visualized in the memory timeline, even if the visualization isn't correct in the sense that the sizes are unknown. Without a valid size these allocation won't show in the summary status or the allocation timeline. The synthetic size is calculated from the length of the filename of the allocation location. For example, if allocated in file `e:\om\c\test\main.cpp` the synthetic length is 21.

This option is provided for both the collect all allocations and collect size restricted options.

The default for both options is that allocations of unknown size will be recorded.

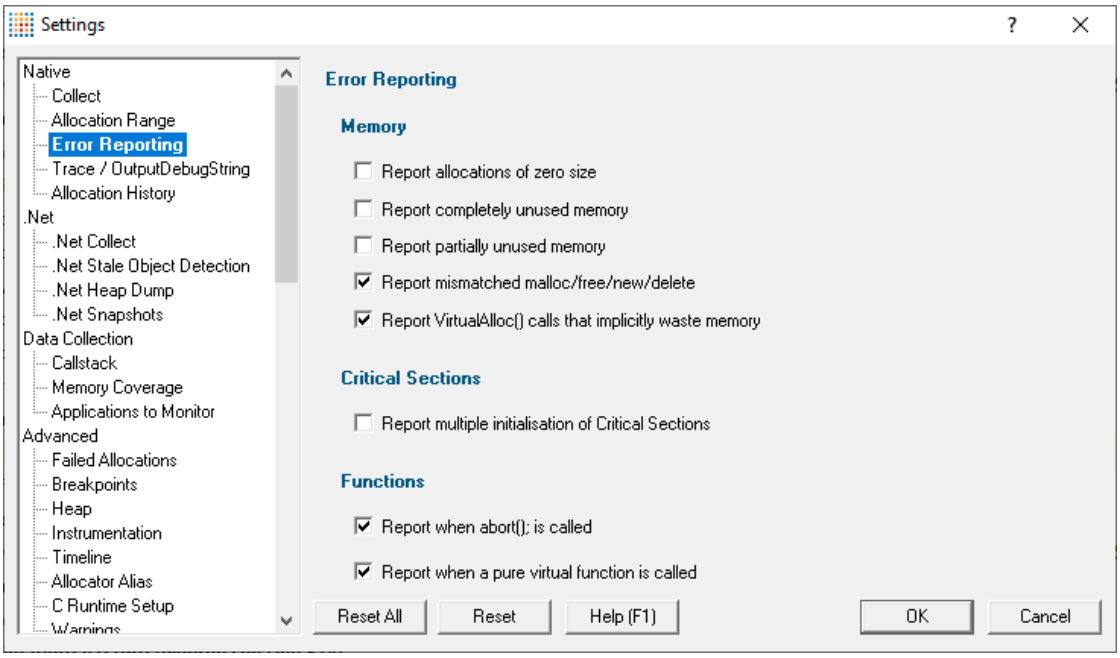
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.1.3 Error Reporting

The **Error Reporting** tab allows you to check for a few special conditions that can stop the target program using a warning dialog or breakpoint


Read on, or click on a setting in the picture below to find out more



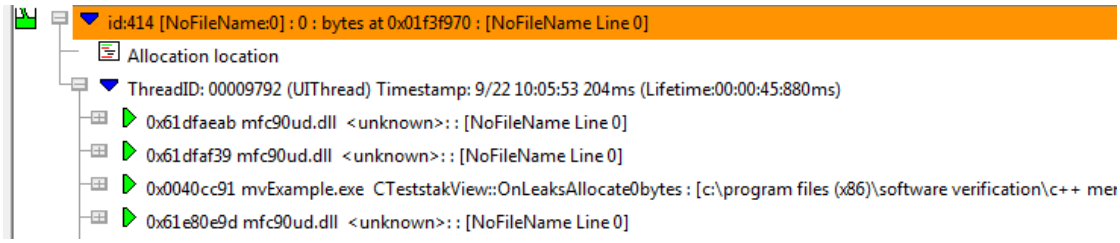
Memory Errors

When allocating CRT memory Memory Validator looks for two conditions that may indicate a programming error, or programming oversight.

- **Report allocations of zero size** › enable the detection of zero size allocations showing the warning dialog in the breakpoints topic

  Zero byte allocations are valid, but not normally something that is done. Bugs come about for example because memory blocks of zero length are allocated and then used, or perhaps not deallocated because the logic looks for a byte *count* rather than a non NULL pointer.

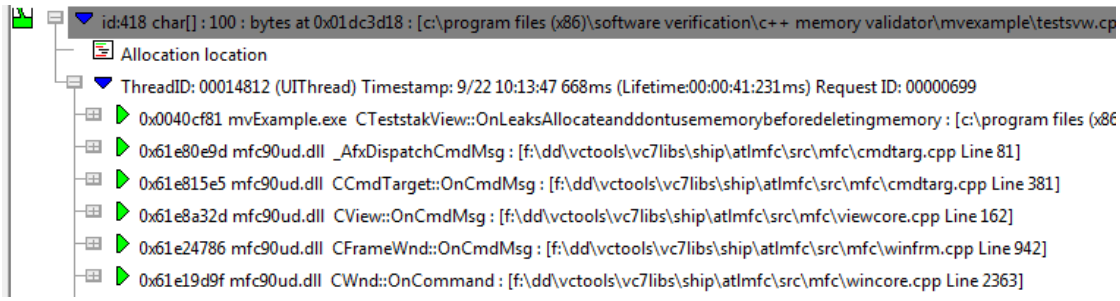
  An example report in the Memory tab:



- **Report unused memory** › enable the detection of memory which is allocated, but later deallocated without actually having been used

  This sometimes happens due to the logic of the function in which it occurs. Or maybe the memory was once required but the source code has changed and the person that changed the code has forgotten to remove the allocation and deallocation code.

  An example of what you might see in the Memory tab:



Improving application performance: for each of the above cases, if this either of these conditions is detected it may be possible to modify the code to remove the allocation and deallocation and thus improve program performance by reducing the use of the heap manager and reducing memory consumption.

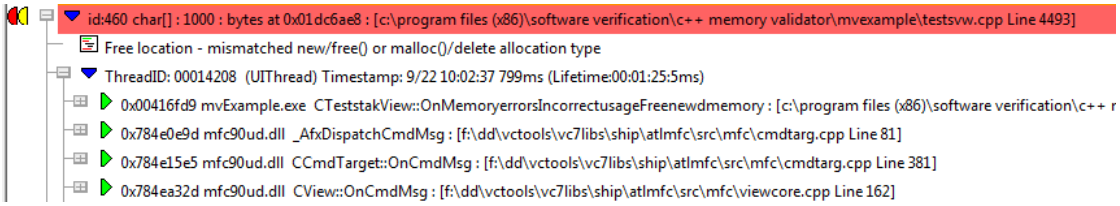
- **Report partially unused memory** › enable the detection of memory allocated, but which is then later deallocated without all of it having been used

- **Report mismatched malloc / free / new / delete** › enables detection of an inappropriate deallocation method compared to that used for allocation

  Allocated memory must be freed with the correctly paired deallocator:

  - new › delete
  - new [ ] › delete [ ]
  - malloc › free

  On detection of mixed use, a report will appear in the Memory tab as below:



- **Report VirtualAlloc() calls that implicitly waste memory** › enables detection of VirtualAlloc() calls that implicitly waste memory because the allocation size requested is smaller than the minimum allocatable size for VirtualAlloc().

  On detection of implicitly wasted memory the wasted memory will be reported in the Memory tab when you refresh the display.

  There are two entries. The first entry (469 below) is for the VirtualAlloc() allocation. It also describes any implicitly leaked memory in the allocation - memory up to the next page boundary. The second entry (470 below) describes the free memory that follows the allocation (starts on the page boundary following the allocated memory).

False positives: note that Memory Validator detects unused blocks of memory by looking for an entire memory block with the uninitialized data signature. There may be rare occasions when the software has filled the entire memory block with the same signature. This will result in a false error report, but inspection of the source code should allow this to be validated.

Other conditions which are checked

In addition to the above conditions, the following conditions are also checked for:

- Zero size allocation
- Negative size allocation
- Bad reallocation
- Bad deallocation
- Bad pointer to C heap
- Bad start of C heap
- Bad node in C heap
- Damage in C++ heap
- Damage in Release CRT heap

InitializeCriticalSection

- **Report multiple initialisation of Critical Sections** › checks if CRITICAL_SECTION objects are initialized more than once

This error typically does not cause a problem in any program execution, but you may wish to find and remove the cause of these errors.

You may also find that some third party DLLs cause these errors. DLLs by nVidia often seem to generate these errors.

Functions

- **Report when abort is called** › sets a signal handler will be called when `raise(SIGABRT)` is called, and if found reports a callstack showing where this happens
- **Report when a pure virtual function is called** › sets a purecall handler that will be called if a pure virtual call is made, and if found reports a callstack showing where this happens

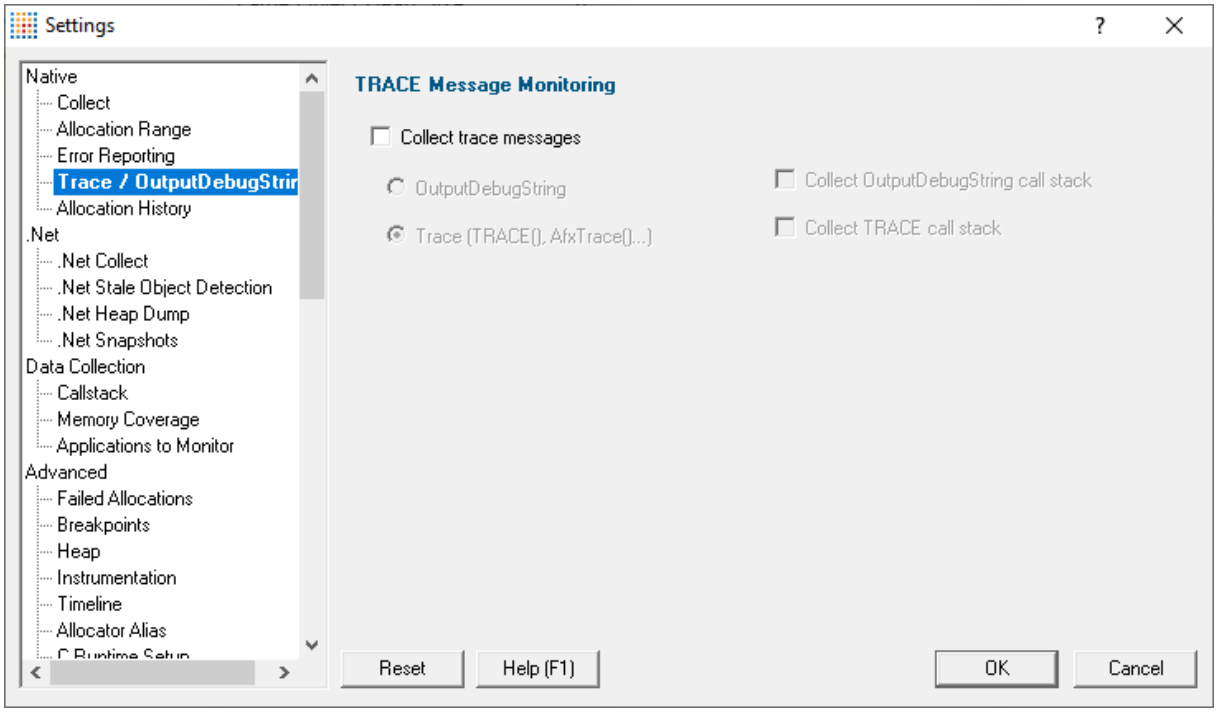
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.1.4 Trace Hooks

The **Trace Hooks** tab provides controls for:

- how TRACE hooks are installed
- tracking duplicate or invalid handles
- controlling maximum duration of the memory timeline



Trace message monitoring

Memory Validator can optionally collect the output from `AfxTrace()`, `TRACE` macros and the `OutputDebugString()` function.

`OutputDebugString()` sends a string to the debugger for display:

```
OutputDebugString(msgbuf);
```

`TRACE` messages are macros, for example:

```
TRACE( "This is a TRACE statement\n" );
```

The TRACE macros ultimately get routed via `OutputDebugString()`, so to prevent duplicate messages, we don't allow both to be collected at the same time.

- **Collect trace messages** › enables collection of either `OutputDebugString()` or `TRACE` messages

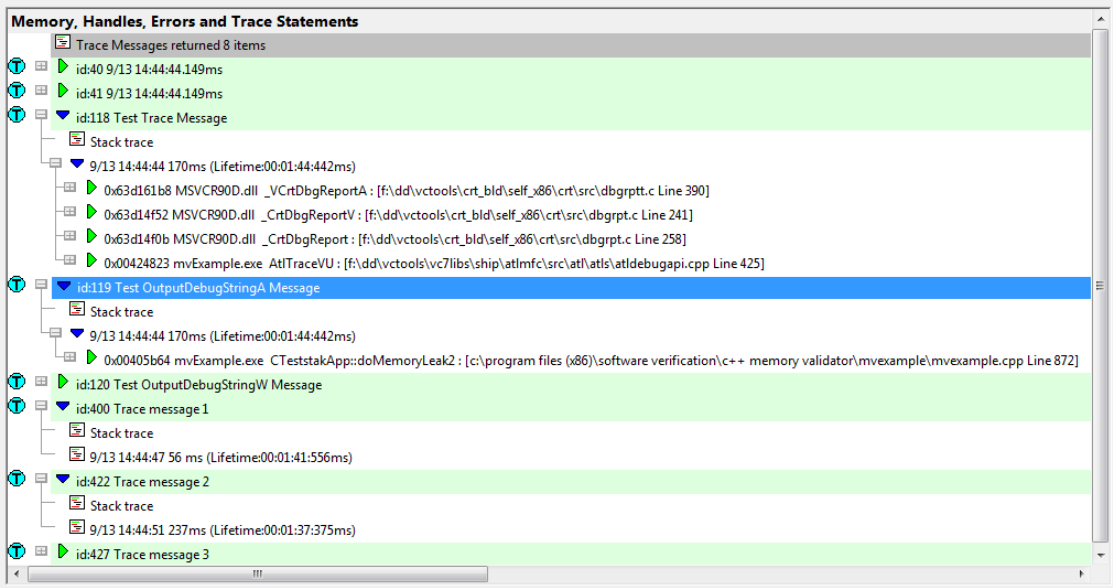
  OutputDebugString › collect `OutputDebugString()` messages

  To collect the relevant callstacks, select **Collect OutputDebugString() call stack**

  Trace (TRACE(), AfxTrace()...) › collect TRACE messages

  To collect the callstacks, select the **Collect TRACE call stack**

Example of trace statements collected in the Analysis tab:



Reset All - Resets **all** global settings, not just those on the current page.

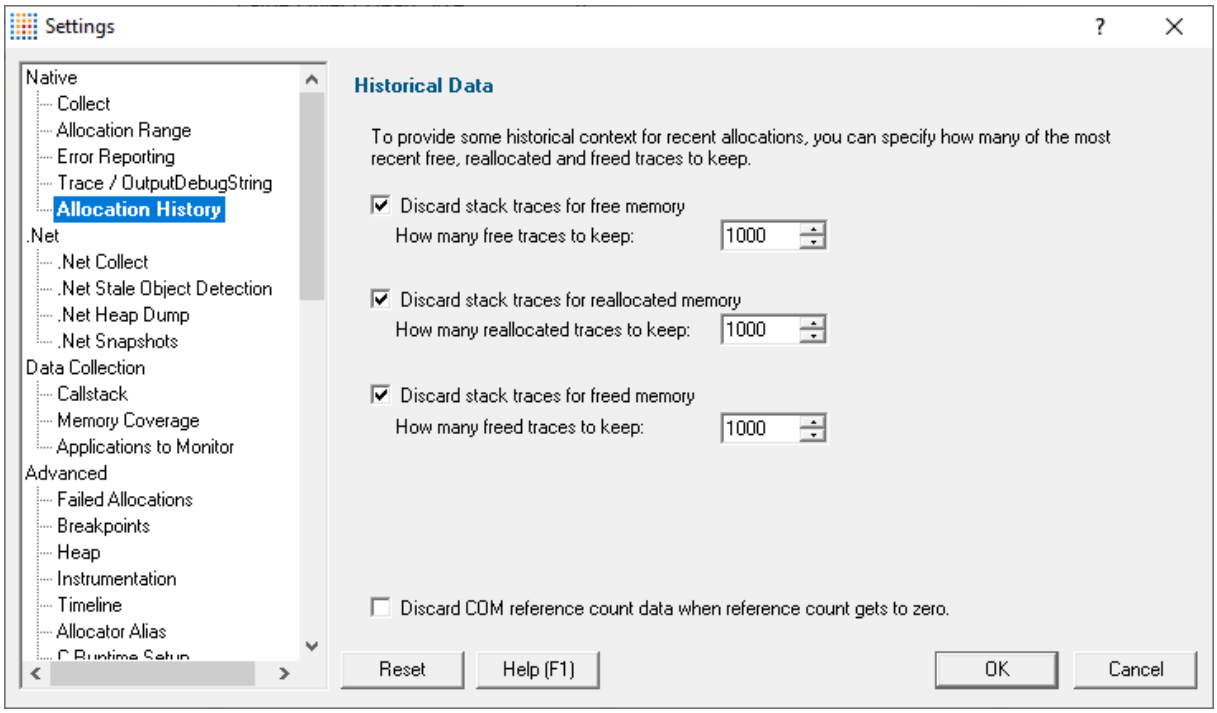
Reset - Resets the settings on the current page.

3.12.1.1.5 Allocation History

The **Allocation History** tab allows you to control how much memory Memory Validator uses to perform its work.

Depending on the task you are performing you may want to allow Memory Validator to use more (or less) memory than for other tasks.

The less information on deallocations, reallocations and freed data that Memory Validator keeps, the smaller the memory requirements placed on your computer.



Historical Data - discarding older deallocated data

Depending on the following:

- the task you are trying to complete using Memory Validator
- your computer's RAM capacity
- virtual memory storage
- the target program being inspected

...you may want to discard some types of data rather than keep it all.

You can optionally discard stack traces for deallocations, reallocations and freed memory, and in each case specify just how much information is kept, with oldest deallocated data being discarded first.

- **Discard stack traces for free memory** › discards information about memory and handle *deallocations*

- **Discard stack traces for reallocated memory** › discard information about *reallocated* memory and handle allocations
- **Discard stack traces for freed memory** › discard information about *deallocated* memory and handle allocations

The default is to keep only the most recent 1000 stack traces in each case, but deselecting each option will keep all the data instead.

COM reference counts

Normally Memory Validator keeps information about COM Reference counts even after the reference count for a particular object reaches zero. This can be very useful for examining the reference count history when a COM object is deleted because its reference count is `Release`'d too many times or not `AddRef`'d enough times.

- **Discard COM reference count data when reference count gets to zero** › discards the relevant data to the *freed* list (see option above)

  Discarding this data is useful when your application is correctly `AddRef`-ing and `Release`-ing large numbers of COM objects and you are only concerned with *leaking* COM objects rather than COM objects being deleted too soon.

Reset All - Resets **all** global settings, not just those on the current page.

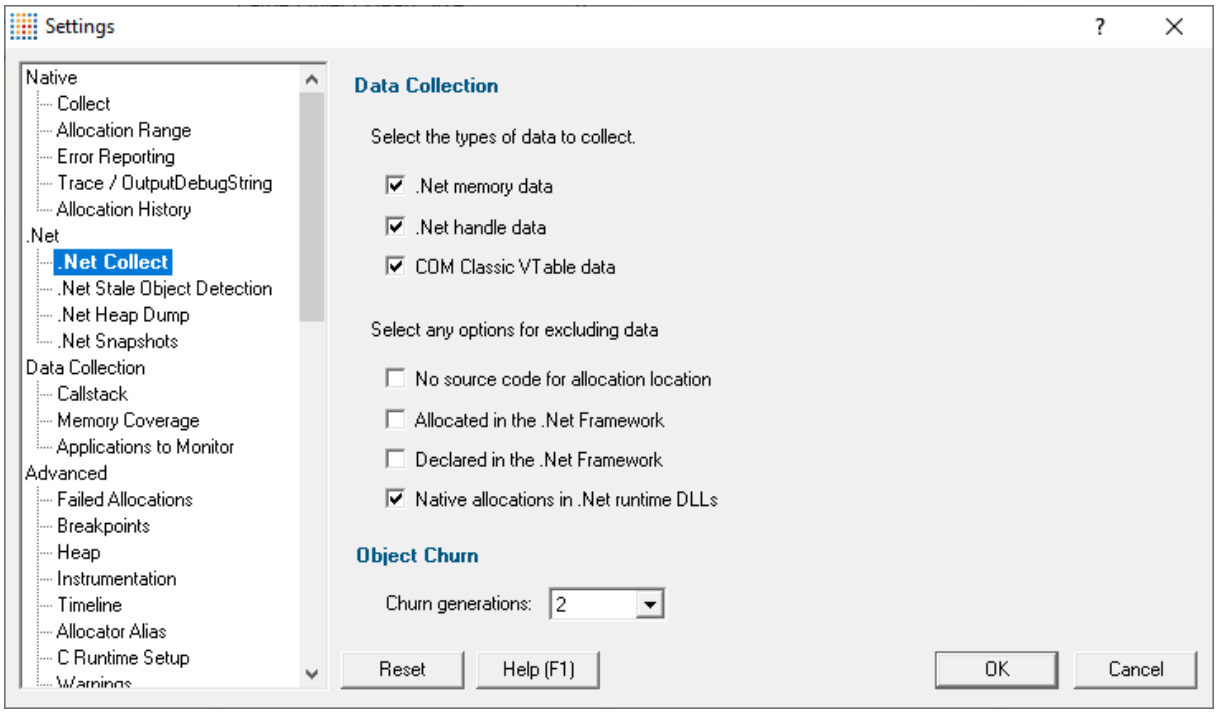
Reset - Resets the settings on the current page.

3.12.1.2 .Net

Enter topic text here.

3.12.1.2.1 .Net Collect

The **.Net Collect** tab allows you to specify which .Net data is monitored and which is ignored when a .Net or .Net mixed mode program is run.



Data Collection

There are three categories of data that can be monitored. By default they are all enabled.

- **.Net memory data** › monitor all .Net memory allocations
- **.Net handle data** › monitor all .Net handle allocations
- **COM Classic VTable data** › monitor COM object creation

A lot of objects are created and destroyed in a .Net application. Some of these objects may not be of interest. We've put them into four categories so that you can easily exclude them if you want to.

- **No source code for allocation location** › If there is no source code for the allocation location, ignore these allocations. Disabled by default.
- **Allocated in the .Net Framework** › If the object is allocated in the .Net Framework, ignore the allocation. Disabled by default.
- **Declared in the .Net Framework** › If the object is declared in the .Net Framework, ignore the allocation. Disabled by default.
- **Native allocations in .Net runtime DLLs** › Ignore native allocations in the .Net runtime DLLs. Enabled by default.

Object Churn

Object churn is a measure of how frequently objects of a specific type are garbage collected after being created. Objects with low churn rates may be leaked objects.

When calculating object churn the results can be influenced by how many of the recent generations of objects you inspect.

- **Churn generations** › How many recent generations to consider when calculating churn.

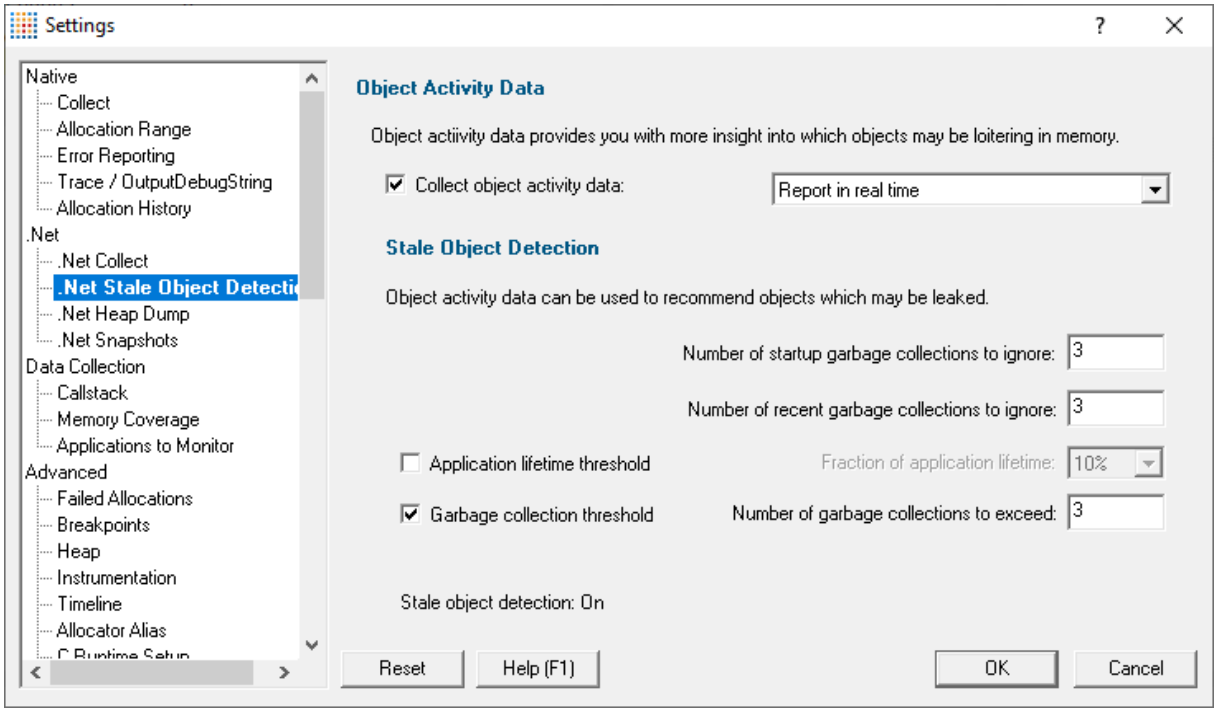
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.2.2 .Net Stale Object Detection

The **.Net Stale Object Detection** tab allows you to specify how Memory Validator detects .Net objects that haven't been used recently.

Objects that haven't been used may be leaked objects.



Object Activity Data

Object activity data describes which objects have had methods called on them in a given timespan. Objects in use will get called often, whereas leaked objects will not get called at all.

There is a performance penalty for this monitoring. It can be quite expensive - how expensive depends on your application's behaviour.

- **Collect object activity data** › monitor all calls to .Net objects.

The data is can be reported to the GUI with each garbage collection, or in real time. Real time provides more info, but is more expensive in terms of performance hit.

Stale Object Detection

Once we have object activity data we can then interpret that data to provide stale object detection - detecting objects that haven't been called recently, where recently can be defined as a fraction of the application lifetime, or within the most recent N garbage detections.

There are four controls that influence which objects are identified as possibly stale. Depending upon your application behaviour and the settings you select you can make all objects appears stale or no objects appear stale, and many states between. How you set these values will affect what values you see in the **Stale Objects** subtab of the Ages view. The values are suggestions only, they are not guaranteed to be stale objects.

Before choosing your settings you need to be aware of a two things about garbage collected applications.

- Objects allocated early in the application lifetime often survive the entire application lifetime.
- Most objects do not survive many garbage collections.

Using these two facts and the knowledge of the application being monitored you can set the stale object detection settings. The settings come as two preconditions and two optional conditions (thresholds). The preconditions are described first.

Startup garbage collections

Typically you don't want to perform stale object detection during the starting up your application - specify the **number of startup garbage collections to ignore**.

Set this value to the number of garbage collections that you notice happen during your application startup. You can see this by watching the Generations tab and counting the number of generations that are present once your application has started but before it has commenced any work.

GUI application: this would be the point at which the GUI had opened and the menu system was ready to accept commands.

Webserver: this would be the point at which the webserver was ready to accept connections from clients.

Recent garbage collections

You may also want to ignore contemporary garbage collections - if so specify the **number of recent garbage collections to ignore**.

Set this value to the number of garbage collections that you think most objects will survive for. For example if you think most objects will be created, used, finalized in 3 garbage collections set this value to 3.

Thresholds

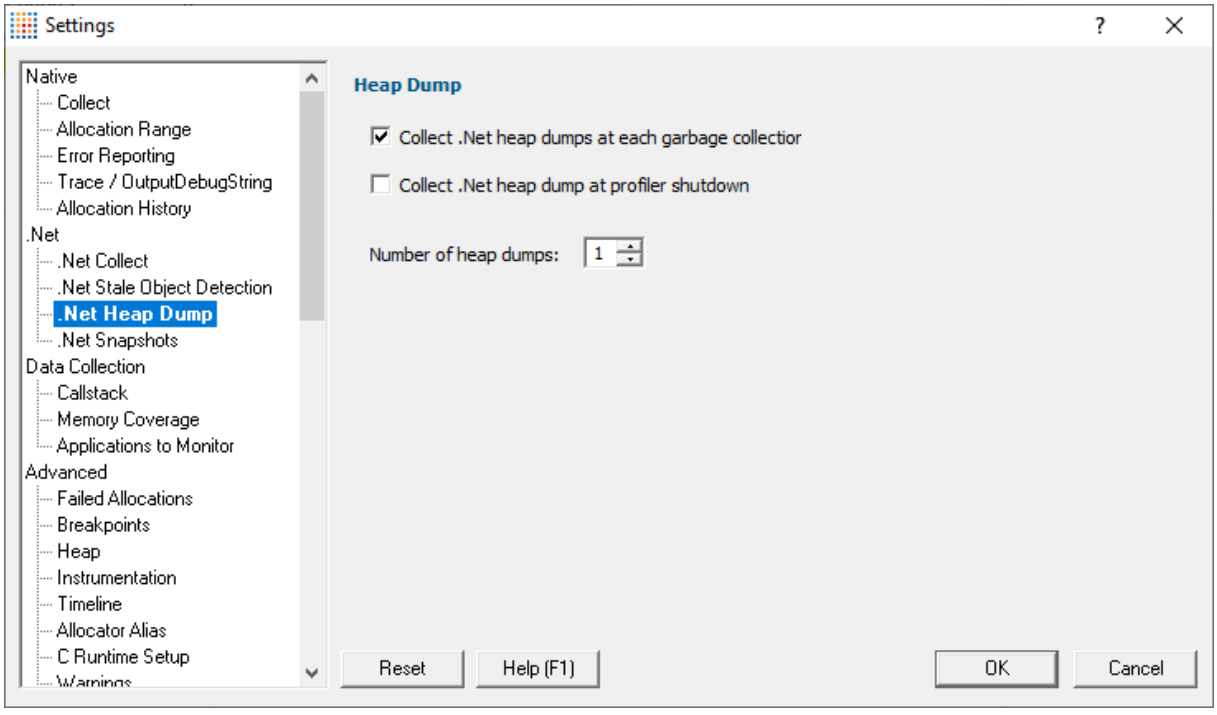
- **Application lifetime threshold** › Consider objects that have not had activity for longer than the specified fraction of the application lifetime to be stale.
- **Garbage collection threshold** › Consider objects that have no had activity for more garbage collections than specified to be stale.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.2.3 .Net Heap Dump

The **.Net Heap Dump** tab allows you to specify how Memory Validator collects and manages .Net Heap Dumps.



Heap Dump

Everytime the .Net runtime performs a garbage collection the CLR provides a heap dump.

Memory Validator can't record every heap dump that happens - eventually we'd run out of memory to store them.

- **Collect .Net heap dumps at each garbage collection** › collect .Net Heap dumps when a garbage collection happens

Caution, enabling this option may cause large programs to run slowly. You may be better served by creating heap dumps manually, when you want them.

- **Collect .Net heap dump at profiler shutdown** › collect .Net Heap dump when the profiler shutsdown

Caution, enabling this option may cause large programs to close slowly. You may be better served by creating heap dumps manually, when you want them.

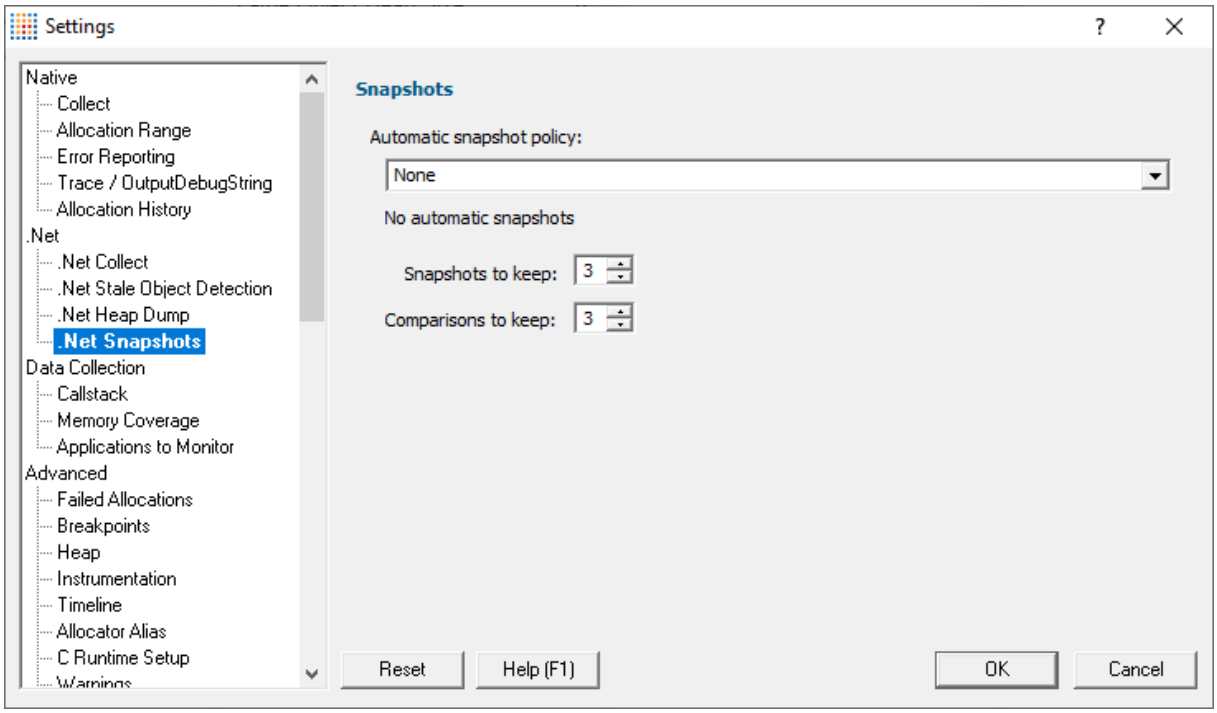
- **Number of heap dumps** › the maximum number of heap dumps that will be kept

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.2.4 .Net Snapshots

The **.Net Snapshots** tab allows you to specify how Memory Validator collects and manages .Net snapshots.



Snapshots

.Net memory snapshots can be performed manually and automatically. These settings control how the automatic snapshots are managed.

Each snapshot uses memory. If you keep too many snapshots you will run out of memory.

- **Automatic snapshot policy** › when snapshots are created
  - **None** › No snapshots will be created
  - **Snapshot before GC** › A snapshot will be created before each garbage collection
  - **Snapshot after GC** › A snapshot will be created after each garbage collection
  - **Snapshot before and after GC** › A snapshot will be created before and after each garbage collection
  - **Snapshot before and after GC, compare before and after** › A snapshot will be created before and after each garbage collection, then they will be compared with each other
- **Snapshots to keep** › maximum number of automatic snapshots that are kept.
- **Comparisons to keep** › maximum number of automatic snapshot comparisons that are kept.

Snapshot disposal

Snapshots and comparisons are disposed oldest first.

For example if you had a threshold of 3 snapshots and you created 5 snapshots A, B, C, D, E, the snapshots remaining would be C, D, E.

A
A B
A B C
B C D (a disposed)
C D E (b disposed)

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

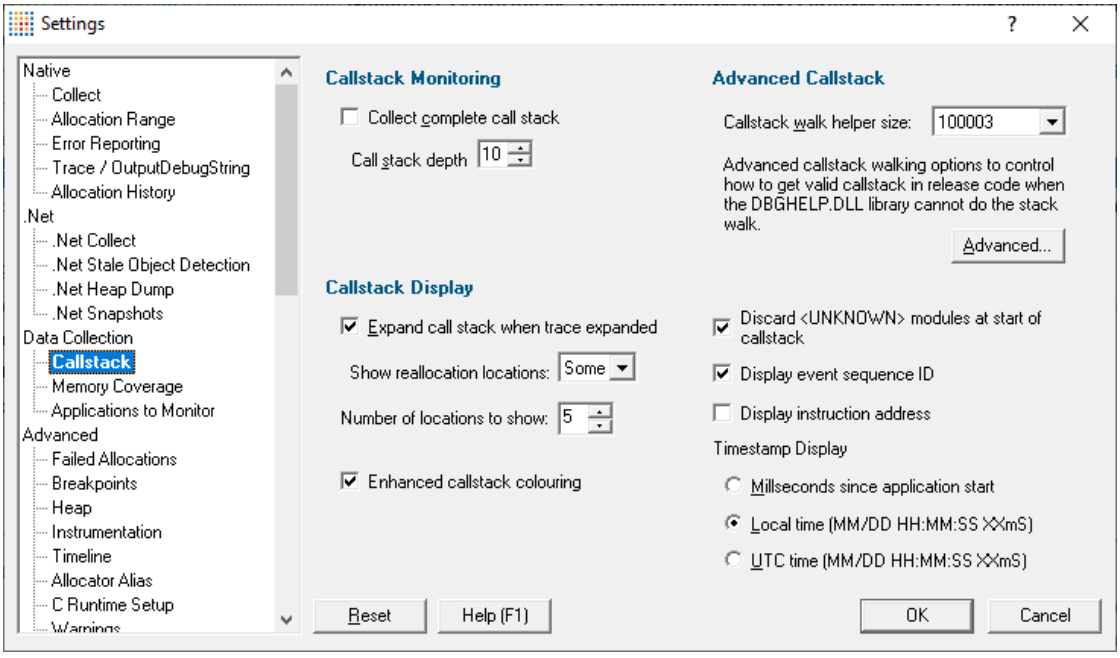
3.12.1.3 Data Collection

3.12.1.3.1 Callstack

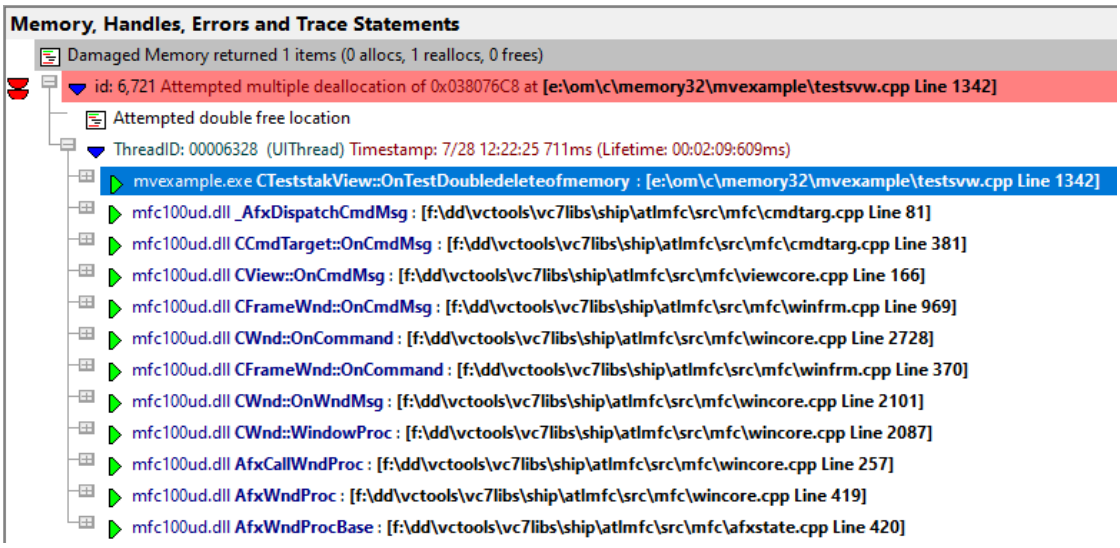
The **Callstack** tab allows you to specify how the callstack is collected, and how information about it is displayed, via four groups of settings:

- callstack monitoring
- callstack display
- timestamp display
- advanced callstack optimization

The following image shows all the *default* settings:



An example of a callstack on the Analysis page:



Callstack monitoring - full or partial callstacks

The Memory Validator stub collects callstacks for each hooked function to display on some of the main tabs such as Memory, Analysis and Pages.

Callstacks can be very long and consume more resources:

- collecting stack traces and converting all the addresses in the stack trace to symbol names takes more *time*
- collecting longer stack traces makes the target program *slower*
- longer stack traces consume more *memory* in the user interface

Opting to collect only part of the stack trace helps reduce that load:

For partial callstacks, deselect **Collect complete call stack** and set the depth of the callstack as required, using **Call stack depth.** The default depth is 10.

Otherwise, select **Collect complete call stack** to collect the entire callstack.

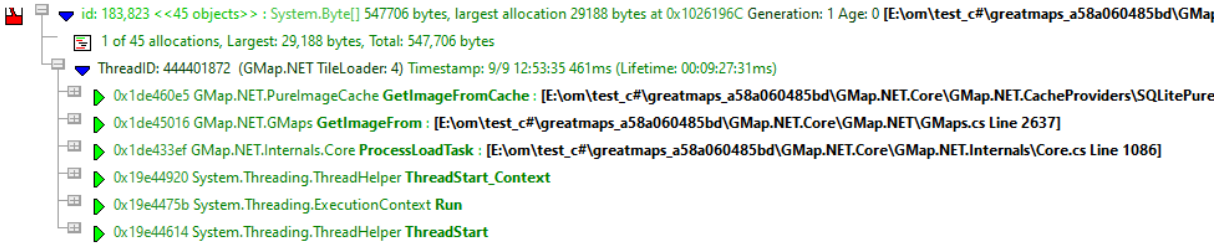
Callstack shorter than expected?

Enhanced callstack colouring

The callstack can optionally be coloured in shades of green to indicate if objects previously allocated (or reallocated) on this callstack have been deallocated or garbage collected.

This colour coding can be used as a hint that memory/handles allocated at this location may not leak.

Here is an example callstack showing the enhanced callstack colouring.



- **Enhanced callstack colouring** › display callstacks in shades of green when this allocation callstack is known to have been used to create memory/handles that have been deallocated or garbage collected.

Callstack display

The callstack can be displayed with optional parameter names and may be auto expanded when a data item is opened

- **Expand call stack when trace expanded** › When a data item is expanded, the whole recorded callstack will also be expanded
- **Show reallocation locations** › When displaying a stack trace for a reallocation, you can show a sequence of stack traces for previous related allocation locations

  This allows you to 'walk' the memory reallocation chain.

  The options are **Some** (with a maximum set below), **All** or **None** (disables the feature).

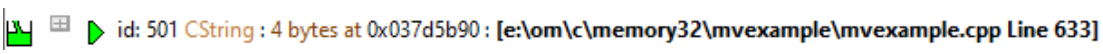
  Choosing *All*, can result in unwieldy data if the reallocation chain is very long.
- **Number of locations to show** › Sets a maximum number of locations when choosing *Some* above (the default is 5)
- **Discard <UNKNOWN> modules at start of callstack** › hides those unknown items where no useful data can be displayed

Event sequence ID

- **Display event sequence ID** › Shows each data collection event with an ID that is incremented each time data is collected

Displaying this ID for each data item allows you to see the order in which allocations, reallocations and deallocations are made.

Example of a callstack showing the event sequence id:501:

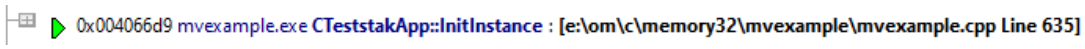


Changing the display of the event sequence id immediately updates existing displays in the current session.

Instruction address

- **Display instruction address** › Shows or hides the address in memory of each callstack item

Example of a callstack entry showing the address 0x004066d9:



Timestamp display

Each callstack has a timestamp as seen in the screenshot above. You can choose how to display that timestamp:

- **Milliseconds since application start** › Time is displayed in milliseconds only. For example: 3897ms
- **Local time (MM/DD HH:MM:SS XXms)** › Time is displayed as Month/Day Hour/Minute:Second Milliseconds. For example: 11/21 13:16:21 897ms
- **UTC time (MM/DD HH:MM:SS XXms)** › Shows Coordinated Universal Time (Essentially GMT) displayed in the same format as Local time

Reset All - Resets **all** global settings, including those on the Advanced Stack Walk Dialog (below), not just those on the current page.

Reset - Resets the settings on the current page, including those on the Advanced Stack Walk Dialog (below).

As the title below suggests, the remaining options are for *advanced* use! An example might be if you find your program has a problem that can be identified, but for which the callstack cannot be collected properly.

Advanced callstack settings

- **Callstack walk helper size** › Specify the size of the cache used to optimize callstack walking

  Memory Validator uses a cache to help it optimize the callstack walking process. The cache is used to avoid calling operating system functions to walk the callstack when the result has been previously calculated.

  For applications generating many unique callstacks, this size may need to be increased.

All sizes are prime numbers, and the default size is 100003 which is large enough for most applications. **If in doubt leave it at the default value of 100003**

Callstack walking

Memory Validator provides three different methods of collecting callstacks for functions which you can choose to tailor callstack collection to the task at hand.

1) The standard Microsoft® DbgHelp StackWalk() function

This function is optimised for walking standard Intel i386 stack frames where the EBP register is pushed on to the stack at function entry and popped from the stack when the function exits. This is the typical stack frame for a program built in debug mode.

The DbgHelp `StackWalk()` function is also capable of reading frame pointer omission data (FPO_DATA) included in a PE file. FPO_DATA is included in optimised binaries that do not use the EBP register to identify the stack frame - this is typical of a program built in release mode.

Missing data in your callstack?Although Microsoft have provided a very capable stack walking function, there are occasions when the StackWalk() function cannot continue walking along the stack, from one frame pointer to the next. When this happens collected stack traces often appear to have data missing, or look "too short". You may have noticed this when debugging release mode programs in Visual Studio®.

2) Alternative (custom) StackWalk() function

This method, although slower than Microsoft's stack walker, does not use stack frames to walk the stack, and so enables the stack walker to walk callstacks that DbgHelp StackWalk() cannot.

This is a proprietary method invented by Object Media Limited and licensed to Software Verify Limited.

What's different about this method?A detailed technical discussion of how this algorithm works is not appropriate here, but suffice to say that all addresses found on the stack are checked for validity, both for code sections, likelihood of CALL instruction taking place, target and source addresses of CALL instructions, removal of duplicate data, and so forth. The resulting callstacks often contain some bogus stack entries, which are obvious to the end user, but not possible to detect by the callstack verification algorithm (this is often due to CALL instructions relying on indirect indexes held in registers which have been changed by the time the stack walker has walked to this point in the callstack - such entries must be taken at face value because they may be valid).

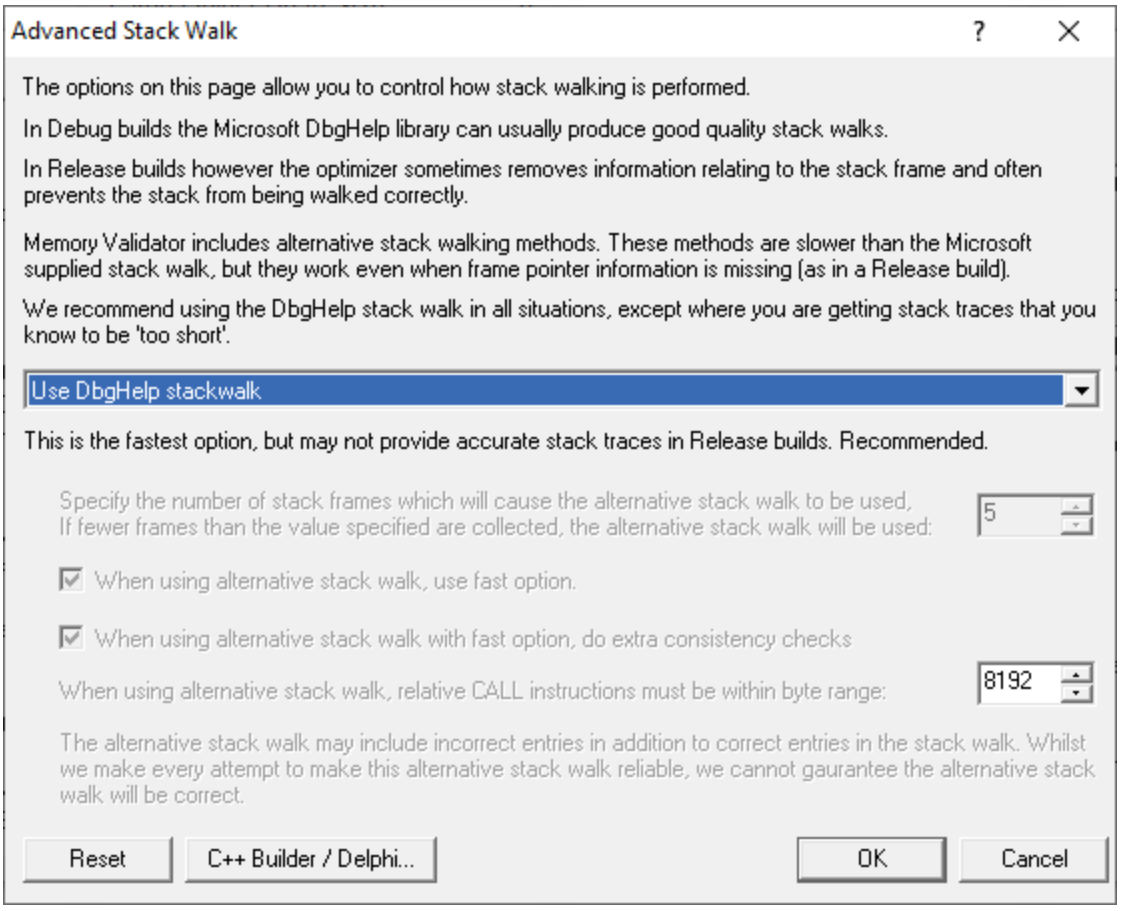
3) Hybrid of the two

The third stack walk type is a hybrid of the other two.

The first method is used to collect all callstacks. Any callstacks that are too short (defined by a callstack length threshold) then have the callstack collected by the second method.

This provides the speed and power of the standard Microsoft stack walker, with the flexibility to collect callstacks that would otherwise be uncollectible when DbgHelp StackWalk() fails to collect the callstack.

The Advanced Stack Walk dialog shown below is accessed via the **Advanced...** button and is used to choose one of the three callstack collection options above:



- **DbgHelp StackWalk** › use the DbgHelp.dll `StackWalk()` function to walk all callstacks

  You may have problems collecting some callstacks in release mode programs, and in some special cases in debug programs depending on your program.

- **Alternative StackWalk** › use the alternative stack walking function to walk all callstacks

  All callstacks will be collected in both debug and release mode programs, but you may find that some callstacks contain incorrect entries.

- **Hybrid StackWalk** › use the hybrid method outline above

  When the alternative stack walk is used you may find that some callstacks contain incorrect entries.

To specify when to use the alternative stack walking function, set a **callstack depth**.

Any standard callstacks that are *shorter* than the depth specified, will be collected using the alternative stack walking function.

➡ See also - recommended usage below

Alternative stack walk method - fast or slow?

- **When using alternative stack walk, use fast option** › uses a faster address verification scheme (recommended) or a slower one

  Do extra consistency checks › When using the *fast* option, consistency checks can optionally be performed (recommended) to reduce the amount of incorrect addresses included in the callstack.

Alternative stack walk method - range of relative addresses

When the alternative stack walk is used, relative address CALL instructions have their target address computed to test if the target address is within a threshold of the previous callstack address.

This provides a form of source address to target address integrity to prevent invalid addresses be placed in the callstack.

- **Relative CALL instruction byte range** › set the threshold which the target address must be within, to fine tune the stack walk

  A *larger* threshold reduces the accuracy of the stack walk by allowing too many invalid addresses into the stack.

  A *smaller* threshold reduces the accuracy of the stack walk by rejecting valid addresses from being placed in the callstack.

  The default is reasonably large 8192 bytes.

Alternative stack walk method - caveats

When the alternative stack walk is used you *may* notice some unusual data on the display:

- **<UNKNOWN> symbols in the middle of call stacks**

  This happens rarely, because the address is not valid but for some reason was not rejected by the alternative stack walk.

- **Symbols in the middle of callstacks that you know cannot be correct**

  This may happen because the address is valid, but not for this position in the callstack, and the address passed the alternative stack walk address verification tests - this address was most

probably the target of an indirect CALL instruction, and as such, could not be verified.

- **Callstacks for data that make no sense**

  This again is rare, but occurs due to Memory Validator monitoring its own behaviour (which can happen in a few limited circumstances). These callstacks are filtered in both stack walk methods, correctly in the standard one, but not perfectly in the alternative method!

Recommend usage

We recommend that in all situations the stack walking method used is either **DbgHelp StackWalk** or **Hybrid StackWalk.**

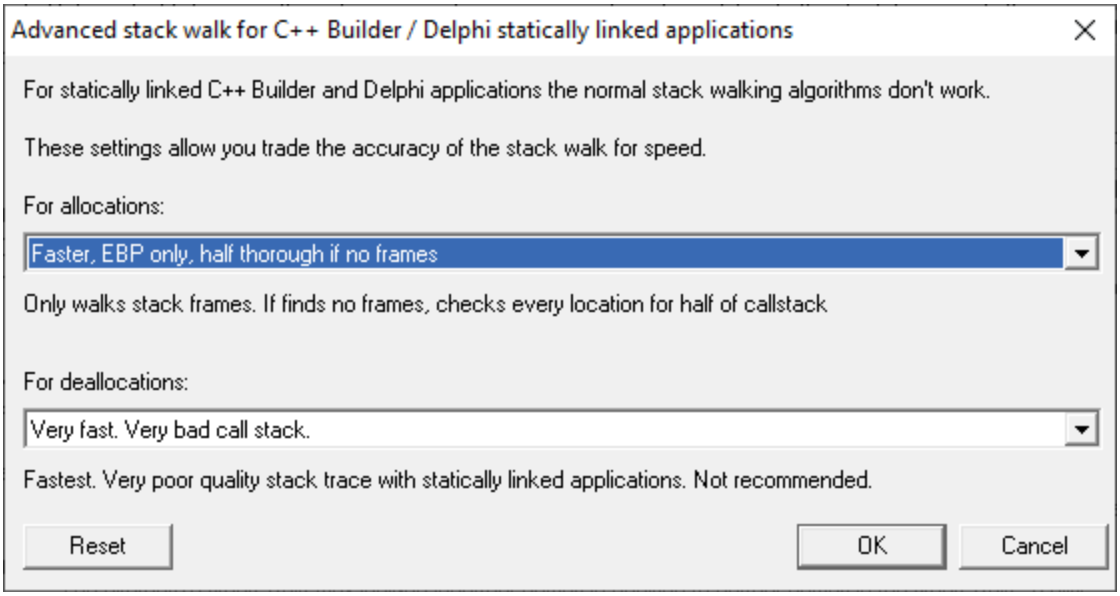
Only if you find your program has a problem that can be identified, but for which the callstack cannot be collected, do we recommend using **Hybrid StackWalk** or **Alternative StackWalk** as appropriate.

C++ Builder / Delphi

If you are using C++ Builder or Delphi to build your applications and you are statically linking the normal method for walking callstacks returns very poor results.

To handle this and allow you to choose between callstack accuracy and speed of execution we've provided the ability to choose the type of callstack walk for both allocations and deallocations. Because deallocation callstacks are less likely to be inspected we think you may wish to choose a less accurate call stack walk for deallocations in return for speed.

The C++ Builder / Delphi Advanced Stack Walk dialog shown below is accessed via the **C++ Builder / Delphi...** button and is used to choose the stack walk options for allocations and deallocations in statically linked C++ Builder and Delphi applications.



- **For allocations** › choose the callstack walking method for allocations
- **For deallocations** › choose the callstack walking method for deallocations

The values that can be choosen for the callstack walking method are:

- **Very fast.** › very fast stack walks, but mainly return callstacks that are incomplete and inaccurate
- **Slow but thorough** › slow stack walks, checks every location on the stack
- **Fast** › slow stacks walks until a stack frame is found, then walks stack frames, continues checking every location when no stack frames found
- **Fast, thorough if no frames** › same as Fast, but if no frames found also does a Slow but thorough stack walk
- **Fast, half thorough if no frames** › same as Fast, but if no frames found also does a Slow but thorough stack walk for half the stack
- **Faster, EBP only** › slow stacks walks until a stack frame is found, then walks stack frames until no more stack frames
- **Faster, EBP only, thorough if no frames** › same as Fast EBP only, but if no frames found also does a Slow but thorough stack walk
- **Faster, EBP only, half thorough if no frames** › same as Fast EBP only, but if no frames found also does a Slow but thorough stack walk for half the stack

 You may wish to choose a not very accurate callstack method for deallocations as deallocation callstacks are not often inspected when monitoring memory issues.

Reset

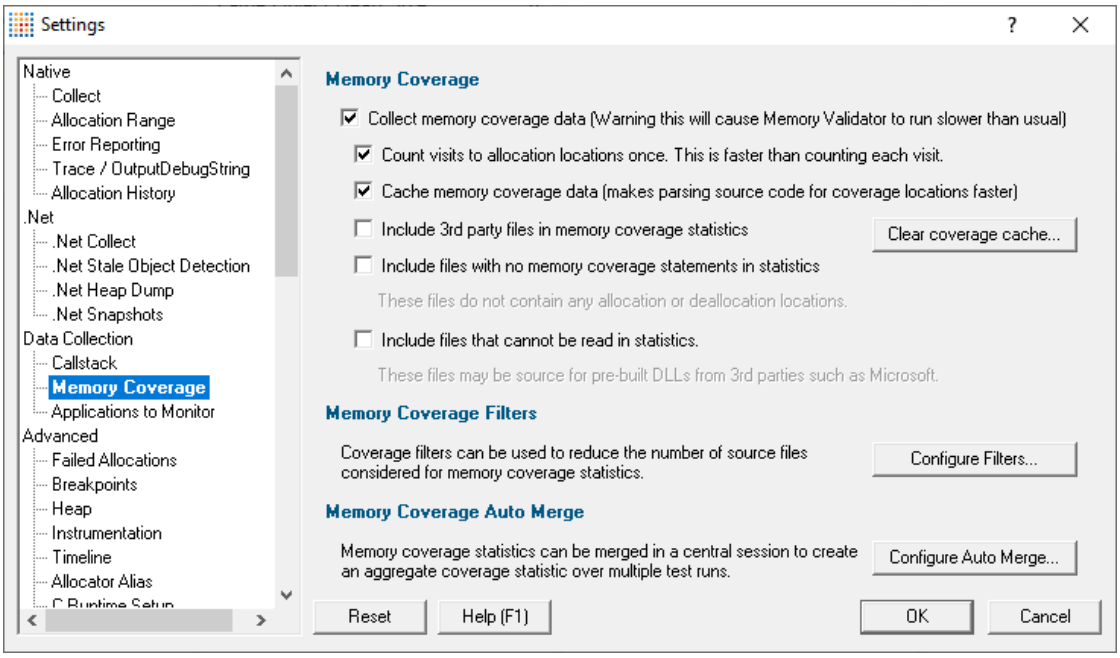
The Delphi Advanced Stack Walk dialog has a button labeled **Reset** at the bottom left of the dialog. This resets *only* the settings on this dialog back to their default values.

The Advanced Stack Walk dialog has a button labeled **Reset** at the bottom left of the dialog. This resets *only* the settings on this dialog back to their default values.

3.12.1.3.2 Memory Coverage

The **Memory Coverage** tab allows you to control how Memory Validator gathers coverage statistics for memory and handle allocations.

The default options are shown below:



What is memory coverage?

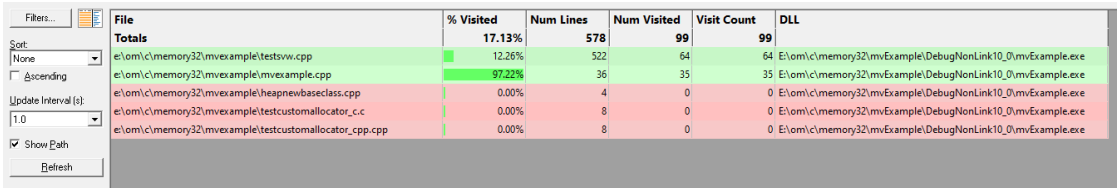
Calculating memory coverage involves parsing your source code to determine all allocation and deallocation locations and matching that information with information from the callstacks of each allocation in your session.

Some of these activities are time consuming, so the default is not to collect memory coverage data, but you can control the behaviour as below.

Controlling memory coverage data collection

- **Collect memory coverage data** › enables the collection of memory coverage data

The Coverage tab (example shown below) shows the memory coverage statistics for each file that contains memory or handle allocation statements. This allows you to check how well you are testing your memory allocation/deallocation code.



When the collection of memory coverage data is enabled, the are some options to tune the performance.

- **Count visits to allocation locations once** › only count line visits once per line when this is selected (default)

Normally each line is counted just once (faster). Although they can be counted for all visits to each line, this will be noticeably slower for large applications.

- **Cache memory coverage data** › enable caching of coverage statement information (default)

  Calculating the coverage statements for each file can be time consuming, especially for large applications as it requires parsing the source code each time the application is run to determine where the allocation and deallocation locations are.

  Caching the coverage statement information improves this by only recalculating it when the source code is modified.

- **Clear Coverage Cache...** › delete any existing memory coverage cache files stored on your computer

- **Include 3rd party files in memory coverage statistics** › include 3rd party files as specified in File Locations

  For most activities you will want to keep this option deselected

- **Include files with no memory coverage statements in statistics** › includes relevant source code files

  Some source code files do not allocate or deallocate memory or handles. As such these files may be viewed as unwanted data on the coverage report, so the default is not to include these

- **Include files that cannot be read in statistics** › includes source code filenames referencing files that do not exist on your computer

  Typically these are third party files in pre-built DLLs. By default they are not included in the statistics.

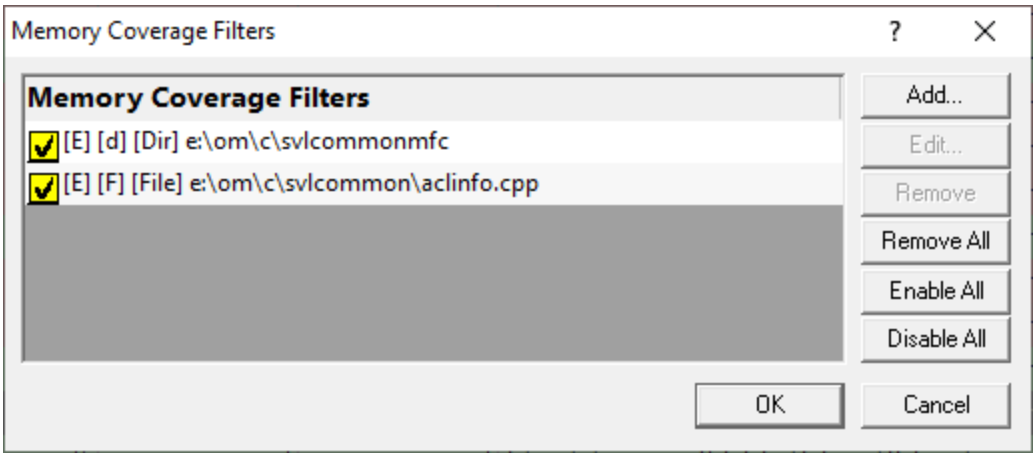
Memory coverage filters

Memory coverage calculation is quite likely to include some third party source files or header files that are out of your control, e.g. from STL, third party files in pre-built DLLs, etc.

If you don't want results to include such files, you can specify a number of filters to exclude them.

Coverage filters need to be set *before* the session starts in order to affect results.

- **Configure Filters...** › shows the Memory Coverage Filters dialog that lets you manage the list of filters...



This dialog is the same one accessed via the local Filters button on the Coverage tab.

The Filters dialog has some basic controls for managing the filters:

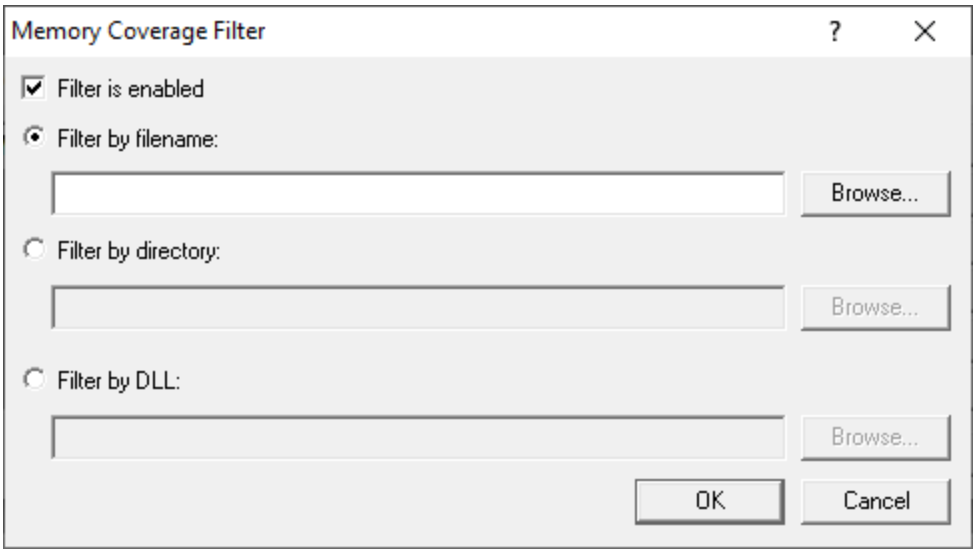
- **Add...** › adds a new filter, as in the example above
- **Edit...** or double click an item in the list › opens the Memory Coverage Filter dialog to modify the enabled state, type or target of the selected filter item

  You can also enable/disable an item via the yellow checkbox on each item in the list
- **Remove** › deletes all *selected* filters in the list, or press *Delete*
- **Remove All** › clears the list of filters
- **Enable All** › sets all filters active
- **Disable All** › sets all filters inactive (but does not remove them from the list)

Each filter can only be one of the three types: filename, directory or DLL. However, you can mix and match multiple filters of any type.

For example, to add a new filter:

- **Configure Filters...** › **Add...** › choose a filter type **filename**, **directory** or **DLL** › **Browse...** (or enter a path directly) › choose a relevant item to add › **OK** › **OK**



To help recognise types of filters in the list, each item is prefixed by some bracketed flags as follows

[E] - enabled
[X] - disabled

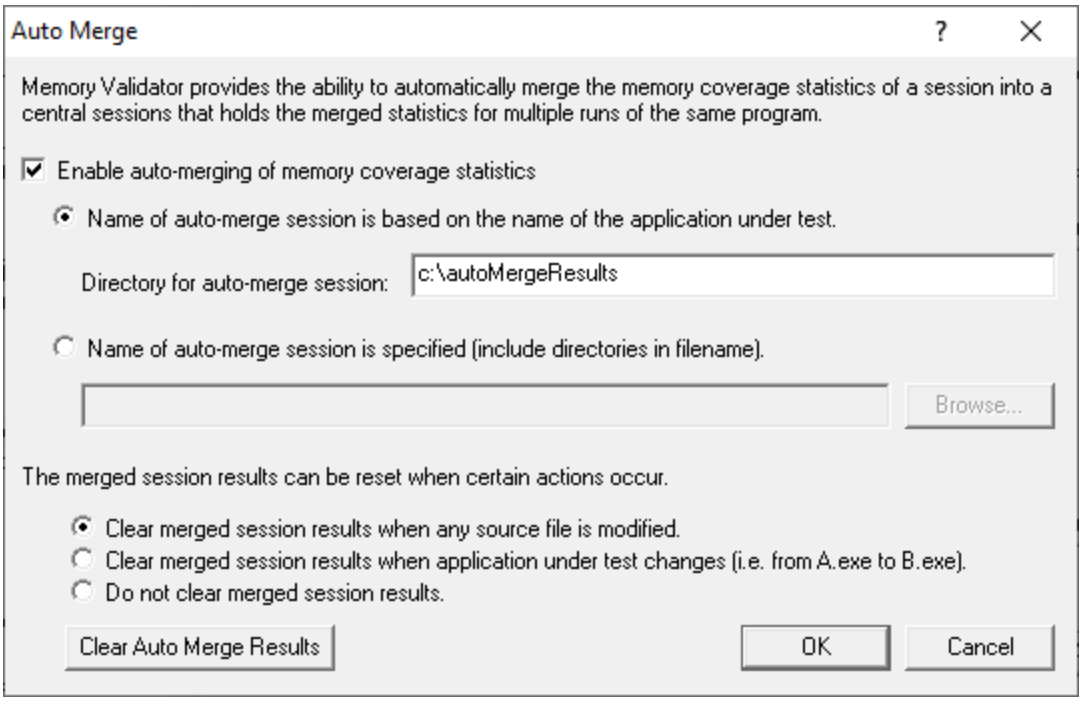
[F] - file
[d] - directory
[D] - DLL

[File] - file
[Dir] - directory
[DLL] - DLL

Memory coverage auto merge

Different runs of your application may execute different parts of your application, in which case you might want to merge the results of one run with the results of another.

- **Configure Auto Merge...** › shows the Auto Merge dialog that lets you configure merging of memory coverage results



The automatic merging works by merging the results of each individual memory coverage session into a central session.

The central session is stored on disk in a file you specify, or in a file using the name of the session, e.g. **TestThis.exe** would get saved in **TestThis.mvm** in the same directory as Memory Validator resides.

- **Enable auto-merging of memory coverage statistics** › switches the merging feature on (default is off)

Depending on how many applications you are performing memory coverage on, you may want your memory coverage data to go to one central location or to a different location for each application under test.

- **Name of auto-merge session is based on the name of the application under test** › saves the central session in a file named according to the application under test in this session (the default)

  By default, the auto-merge session will be stored in the same directory as Memory Validator, but you can change this:

  Directory for auto-merge session › saves the auto-merge session in the specified directory

  For example, if you run the application **nativeExample.exe**, and specify a central session directory of **e:\memoryCoverageResults**, the central session will be saved in a file named **e:\memoryCoverageResults\nativeExample.mvm**.

- **Name of central session is specified** › save the auto-merge session in a filename and path of your choice (enter or **Browse...** to a file)

Auto-merge session reset

The auto-merge results can be automatically cleared by certain triggers, or not cleared at all.

When performing memory coverage analysis sometimes you will uncover a bug in your software and need to modify the software, and/or run different executables. When this happens, line numbers and/or files often change, and you usually wouldn't want to merge memory coverage data from the modified software with existing coverage data.

The triggers for clearing the merged session results are:

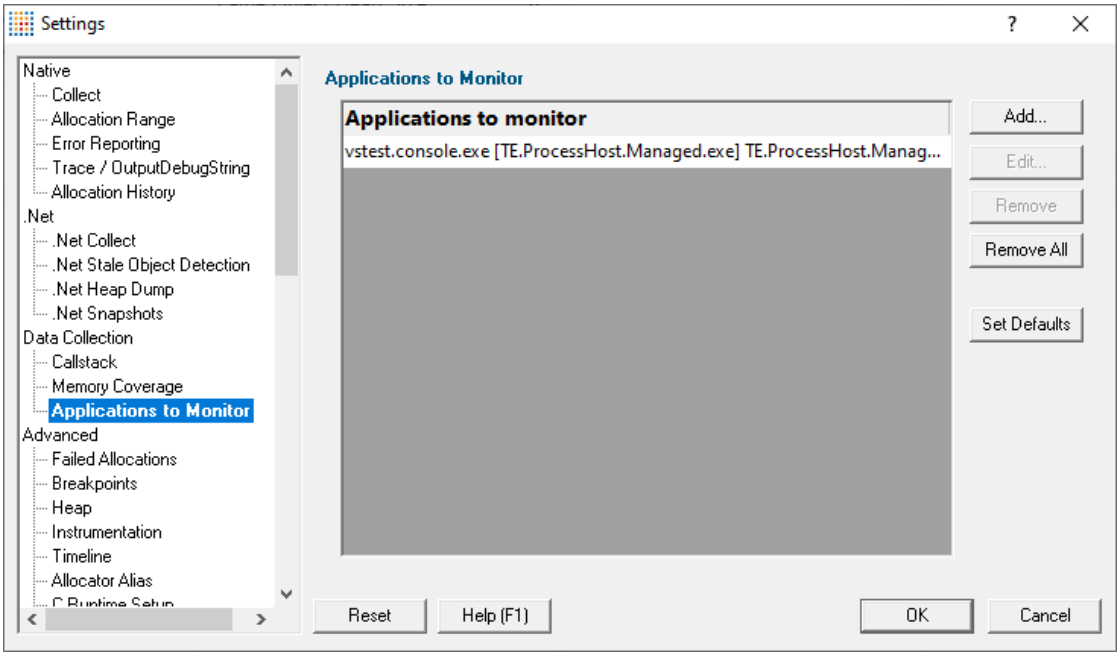
- When any **source file** is modified (the default)
- When the **application** under test changes
- **No clearing** of merged session results occurs under any circumstance

Reset All - Resets **all** global settings, not just those on the current page. Any memory coverage filters added as above are also removed.

Reset - Resets the settings on the current page. Any memory coverage filters added as above are also removed.

3.12.1.3.3 Applications to Monitor

If your target program launches other child applications then the **Applications to Monitor** page lets you choose which ones to monitor.



Monitoring child applications

You may have a case where the program you need to start is not the one you are interested in.

Your program may launch child applications and it may be one of *those* that you want to monitor with Memory Validator.

An example might be for unit testing where a test program spawns one or more child applications, or it might launch the same application multiple times.

The applications to monitor

The main list of **Applications to monitor** shows programs you may want to launch and the child applications they subsequently start - i.e. the you may be interested in monitoring.

Once a definition has been added, you can then use the Application to Monitor setting on the Launch Dialog or wizard to choose which of these child applications you actually want to monitor in a given session.

Managing the applications to monitor

The list contains a set of definitions - each one being for a different launch program.

For each launch program you can set the child applications you might want to monitor later.

An application is defined by its type (native and .Net, or .Net Core), the application executable name, and for .Net Core applications an additional application DLL that is used to identify the application.

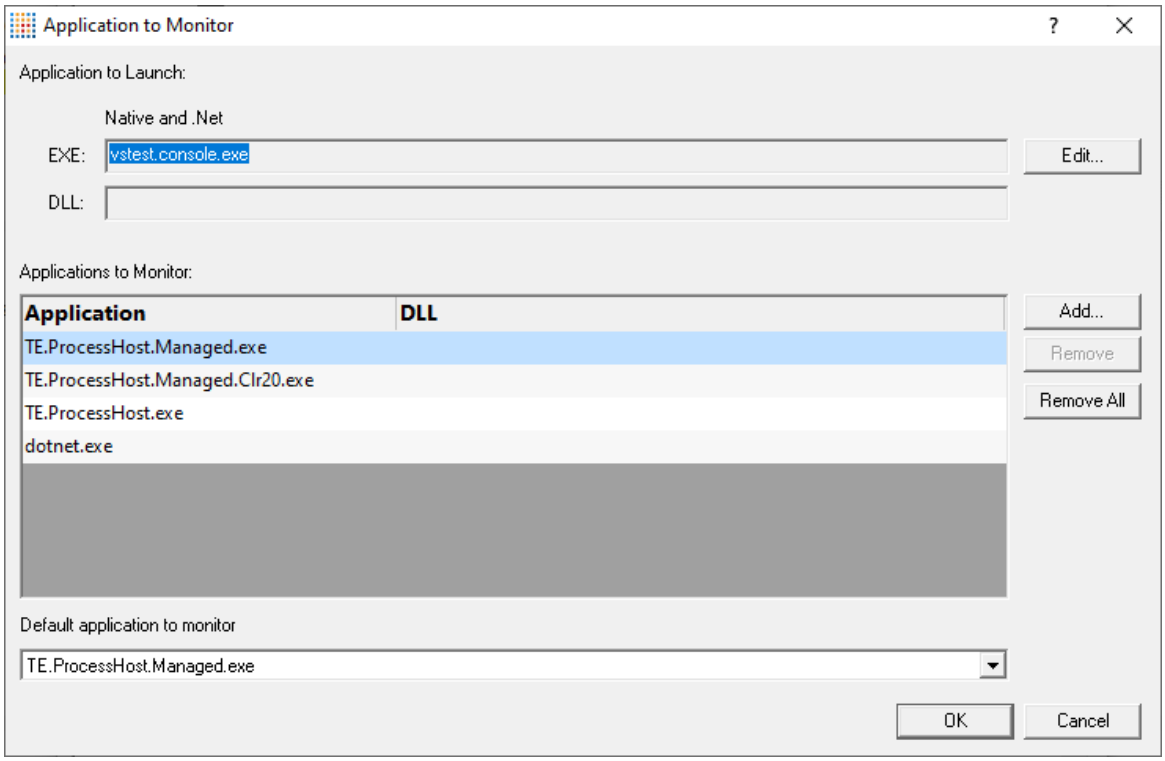
- **Add** › add a new module definition using the **Application to Monitor** dialog below
- **Edit** › modify a selected definition in the list, using the **Application to Monitor** dialog again
- **Remove** › removes any selected definitions in the list
- **Remove All** › clears the list
- **Set Defaults** › reset the list of known applications to those as configured with a new install of Memory Validator

  The defaults are currently setup for Microsoft's Visual Test software `vstest.console.exe`.

The Application to Monitor dialog

The **Application to Monitor** dialog lets you define or edit a launch program and it's child applications.

The values you specify here are the ones used on the launch dialog and launch wizard to customize which application actually gets monitored.



- **Application to Launch** › **Edit...** to select the initial starting application that will be *launching* the applications you want to monitor

  Any executable names found in the selected program will automatically be displayed in the list of **Applications to Monitor**.

  If you don't wish to use these automatic names you can **Remove** them.

- **Add** › add an additional application that you know will be started by the launch program

  Child applications that you add are used *without* the path.

  Excluding the path gives more scope for matching launched application names if they are launched with a different path.

- **Remove** › removes any selected applications in the list

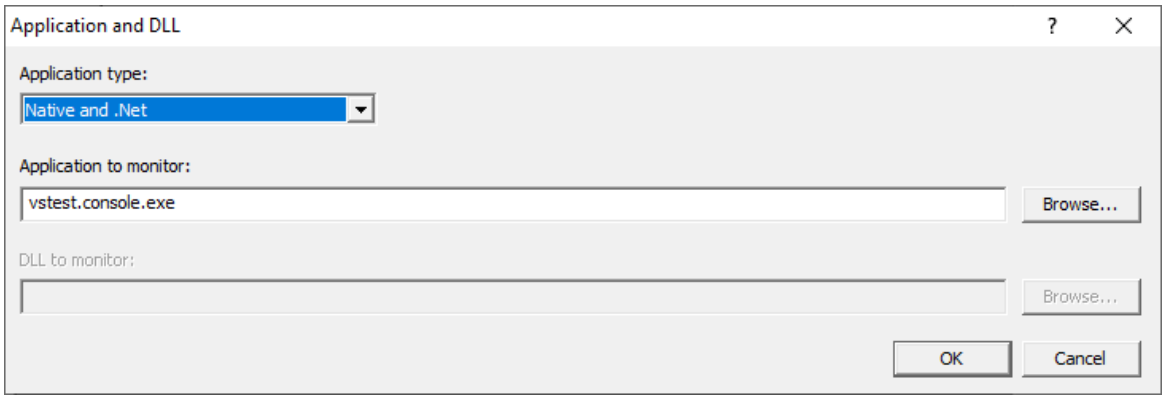
- **Remove All** › clears the list

- **Default application to monitor** › choose the appropriate item to be the default item

  The default application will be selected on the launch dialog (or wizard) whenever the *start* program is specified as the one at the top of this dialog.

The Application and DLL dialog

The **Application and DLL** dialog lets you define or edit a launch program and a launch DLL.



- **Application type** › choose the type of application
  - Native and .Net
  - .Net Core (Framework Dependent)
  - .Net Core (Self Contained)
- **Application to monitor** › edit or **Browse...** the application EXE to monitor.

  This can be an executable name or the full path to the executable. For example **test.exe** or **c:\unitTests\test.exe**.

  For native applications this is the application executable.

  For .Net Framework applications this is the application executable.

  For .Net Core Framework-dependent applications this is most likely going to be **c:\program files\dotnet\dotnet.exe**.

  For .Net Core Self-contained applications this is the application executable.
- **DLL to monitor** › edit or **Browse...** the application DLL to monitor. This field is only needed for .Net Core applications.

  This can be an executable name or the full path to the executable. For example **test.dll** or **c:\unitTests\test.dll**.

  For native applications this is not used.

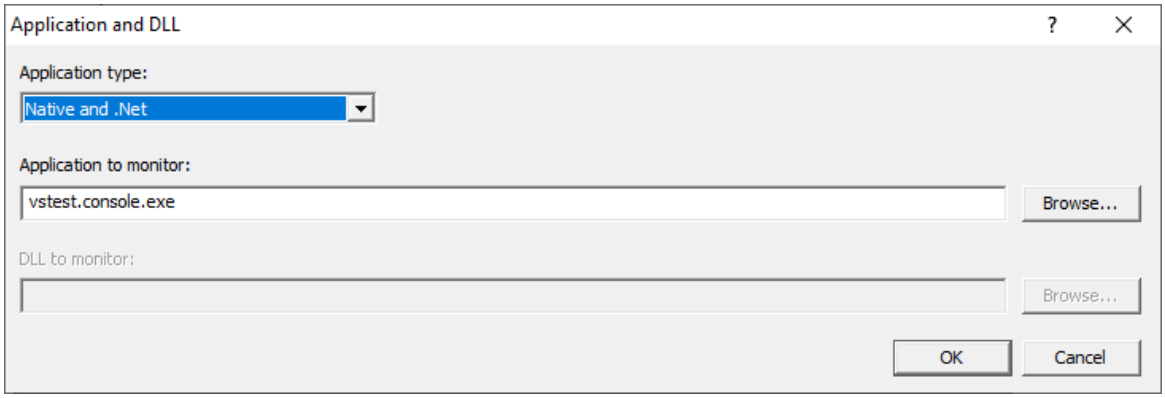
  For .Net Framework applications this is not used.

  For .Net Core Framework-dependent applications this is the application dll. (the name of the dll that you would pass to dotnet.exe on the command line).

  For .Net Core Self-contained applications this is the dll that has the same name as the application executable. (for theApp.exe, the dll name is theApp.dll).

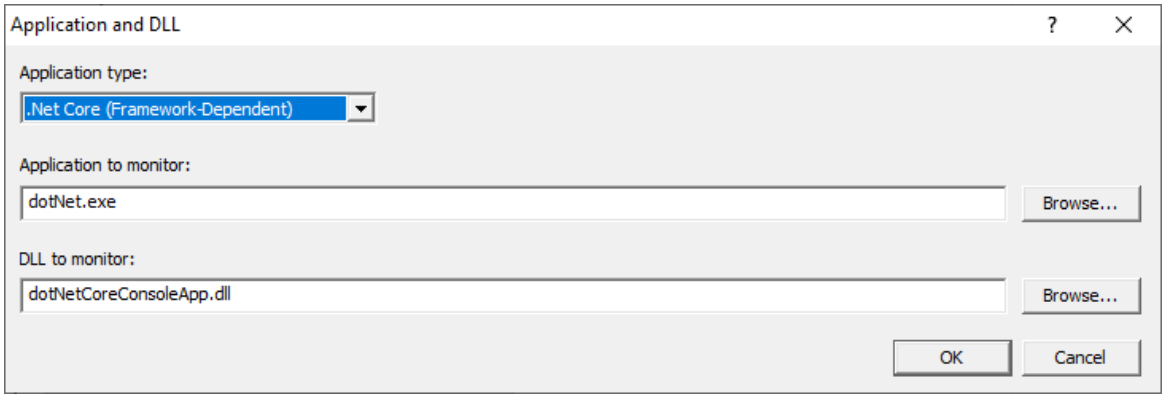
Example Dialogs

Native

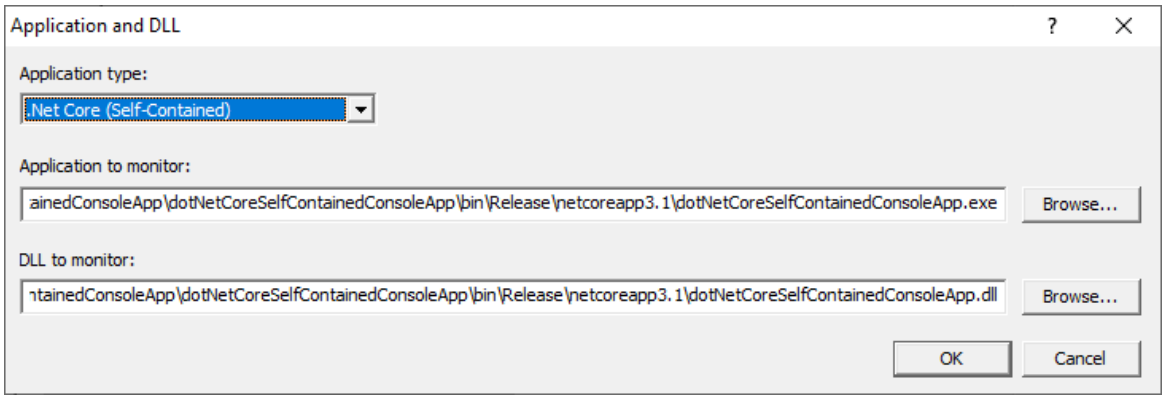


.Net

.Net Core (Framework-dependent)



.Net Core (Self-contained)



Reset All - Resets **all** global settings, not just those on the current page.

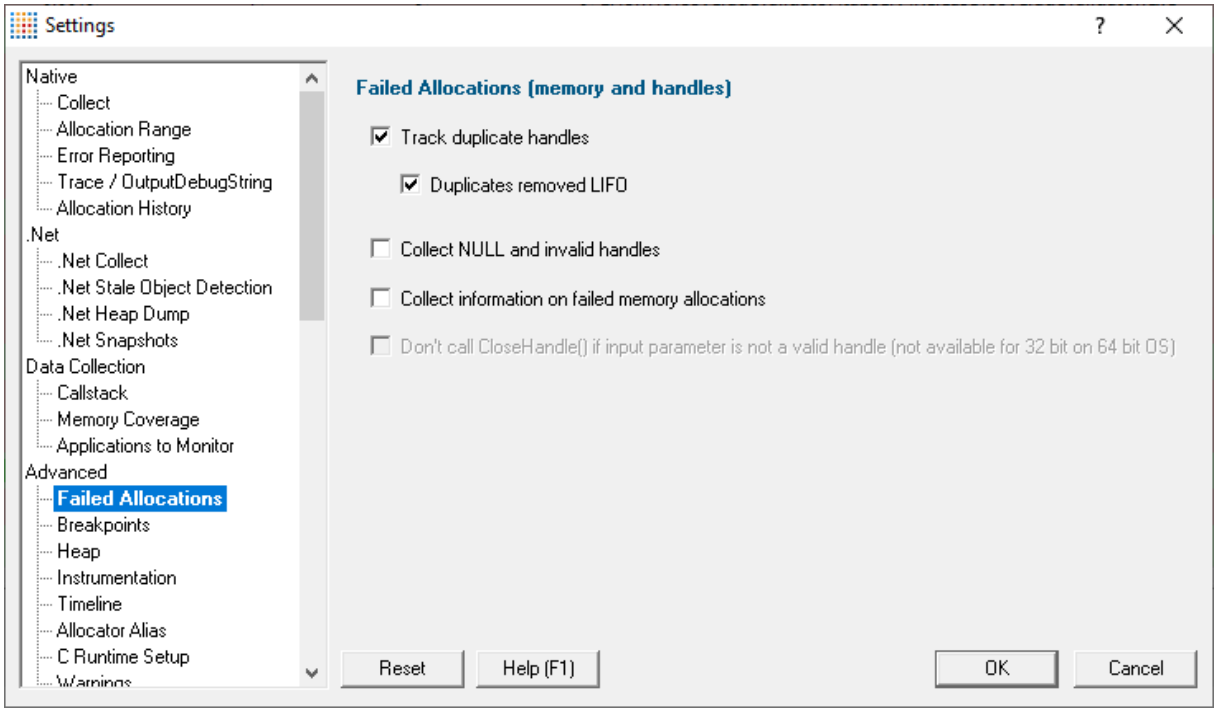
Reset - Resets the settings on the current page.

3.12.1.4 Advanced

3.12.1.4.1 Failed Allocations

The **Failed Allocations** tab allows you to control how Memory Validator responds to allocations that fail to allocate memory or fail to allocate handles.

The picture below shows the default settings:



Duplicate handles

Some handle allocation functions return the same value as a previous handle, even though the previous one hasn't been deallocated.

Memory Validator can accommodate this, but cannot guarantee to match up the correct allocation and deallocation location for these handles (since they are duplicates).

You can choose if the values are resolved first in first out (FIFO), or last in first out (LIFO). This will affect how duplicate handle *deallocation* and *allocation* stack traces are related.

- **Track duplicate handles** › enable the tracking of duplicate handles (default)

  Duplicates removed LIFO › resolves duplicate deallocation handle values in a last in first out manner (default)

  Only enabled if tracking duplicate handles.

NULL/invalid handles

Memory Validator can collect or discard `NULL` and `INVALID_HANDLE_VALUE` handles. If you are having problems with bad handle values, you may want this option enabled.

- **Collect NULL and invalid handles** › enable collection of invalid handle values

Preventing CloseHandle() from being called with bad values

When handles are closed using **CloseHandle(),** some non-valid handles can cause your application to crash if they are passed in. Memory Validator detects if the handle is not a valid handle and reports errors.

- **Don't call CloseHandle...** › prevent non-valid handles from being passed to **CloseHandle()**

  The recommended setting is off, meaning that `CloseHandle()` will always be called

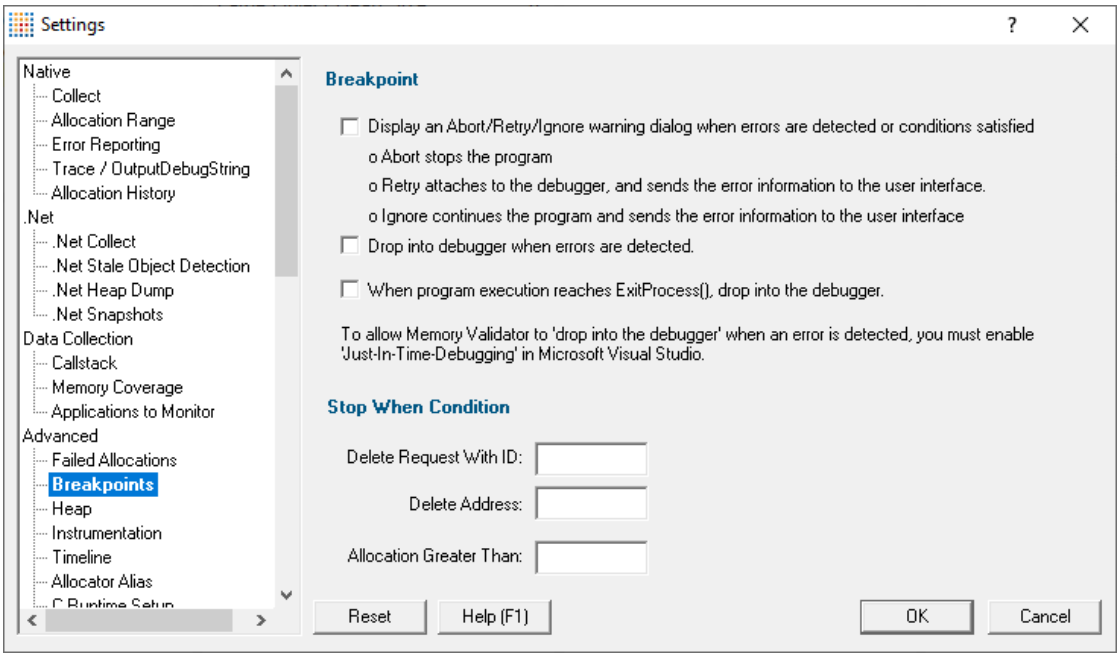
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.2 Breakpoints

The **Breakpoints** tab allows you to control how Memory Validator responds to error conditions it detects in your application, such as prompting for dropping into a debugger.

The picture below shows the default settings:



Breakpoints

Memory Validator can cause the target program to display warning dialogs or stop at a breakpoint when various conditions are met during the allocation or deallocation of an object.

- **Display an Abort/Retry/Ignore warning dialog when errors are detected** › shows the warning dialog below (disabled by default)

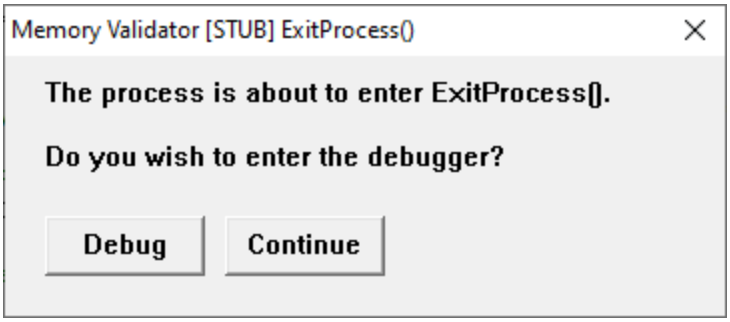
The warning dialog message varies according to the error encountered
Abort › lets the program to terminate at this point
Retry › enter the debugger
Ignore › lets the program to continue as normal
Don't show this again › prevents seeing warning dialogs (whatever the message) for the duration of this session

- **Drop into debugger when errors are detected** › forces a breakpoint instruction (int 3) to be executed causing the program to stop in the debugger, if attached (disabled by default)

  If the program is not attached to a debugger, the standard Microsoft® attach to debugger dialog will be displayed.

If both the above options are selected, the first one will be used as it allows the user to action the breakpoint by choosing **Retry**.

- **When program execution reaches ExitProcess(), drop into debugger** › forces a breakpoint instruction to be executed on exit, showing a dialog as below



Debug › enter the debugger and determine what caused the process to exit
Continue › allow the program to exit as normal

Exit without warning!

Some bugs manifest themselves by the process suddenly exiting without any particular reason.

There are two common causes of this:

- The program encountered a serious error, did not warn the user and called `exit()` or `ExitProcess()`

  If the program calls `exit()`, the code will still be routed via `ExitProcess()`.

- The program exhausted its stack space and could not continue

  It's possible there's not even enough stack space to warn the user that the stack space has been exhausted!

  When the program runs out of stack space, Memory Validator can't help.

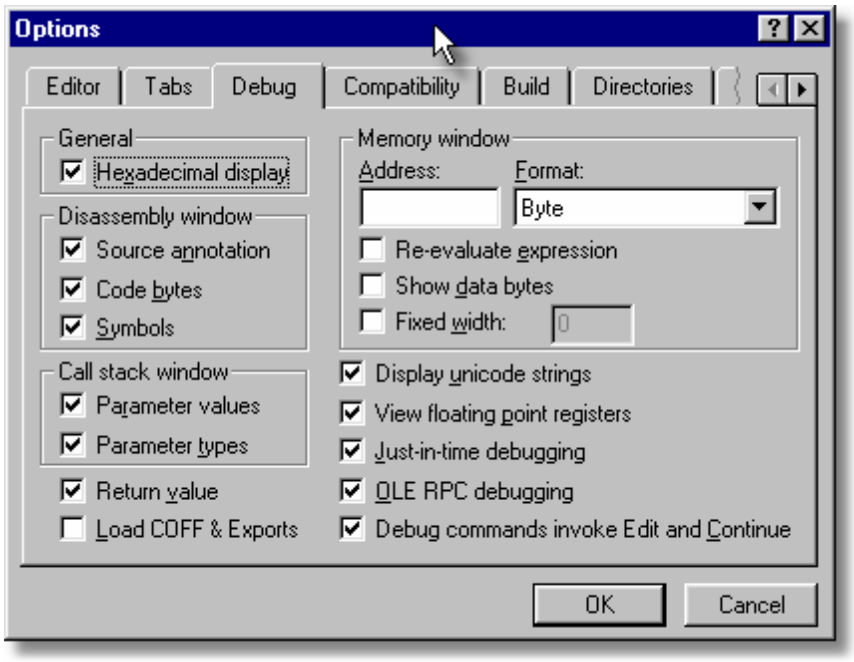
Just-In-Time debugging and exception handling

If your program has an exception handler that catches exceptions and closes the application, enabling these options may cause your program to terminate instead of causing the debugger to attach to your program. Otherwise, the options described here will perform correctly.

For breakpoints triggered by Memory Validator to work, your debugger (Microsoft® Visual Studio® or other) must have **Just-In-Time debugging** enabled.

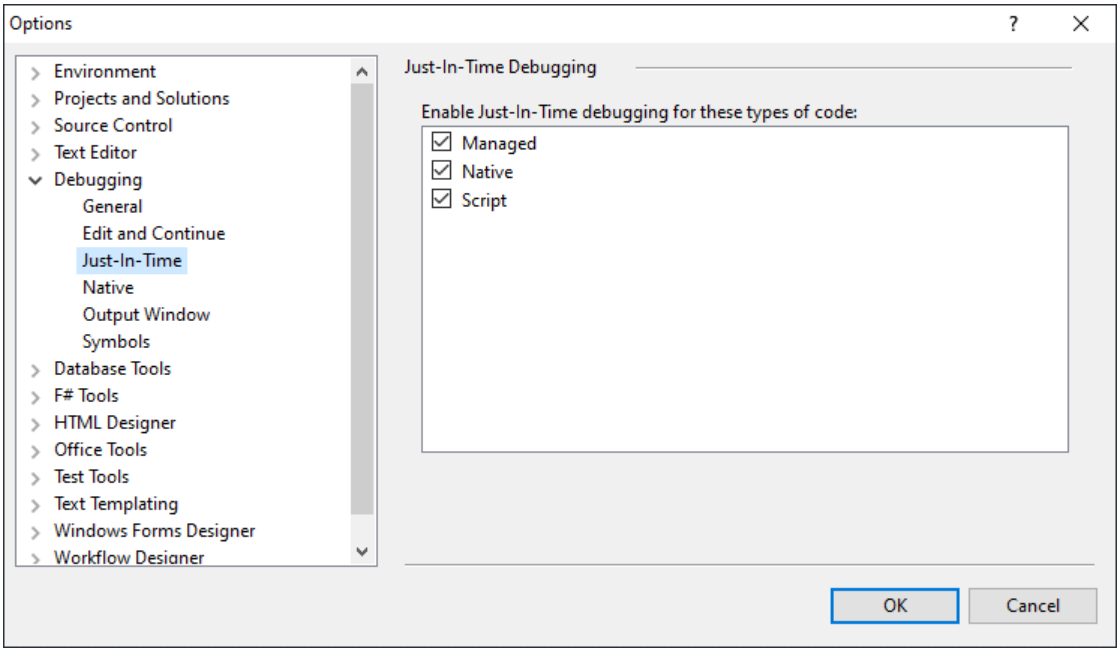
In older versions of Microsoft® Visual Studio®:

Tools menu › **Options...** › **Debug** tab › select the **Just-in-time debugging**



In newer versions of Visual Studio®:

Tools menu › **Options and Settings...** › select **Debugging** in the list › **Just-in-time** › tick the relevant checkbox(es)



'Stop When' conditions

An ID, an address or a size can be checked at various times to cause either a warning or breakpoint, as designated in the breakpoints settings:

- **Stop When Delete Request With ID** › enter an ID to be used for the delete request to trigger the warning dialog

  This is similar to using the Microsoft® `_crtBreakAlloc` variable in the debugger. See also `_CrtSetBreakAlloc()`.

- **Stop When Delete Address** › enter a specific address for deletion to trigger the warning dialog

- **Stop On Allocation Greater Than** › enter a size theshold above which allocation will trigger the warning dialog

All three conditions can be active at the same time, and leaving any of the above fields blank resets that condition.

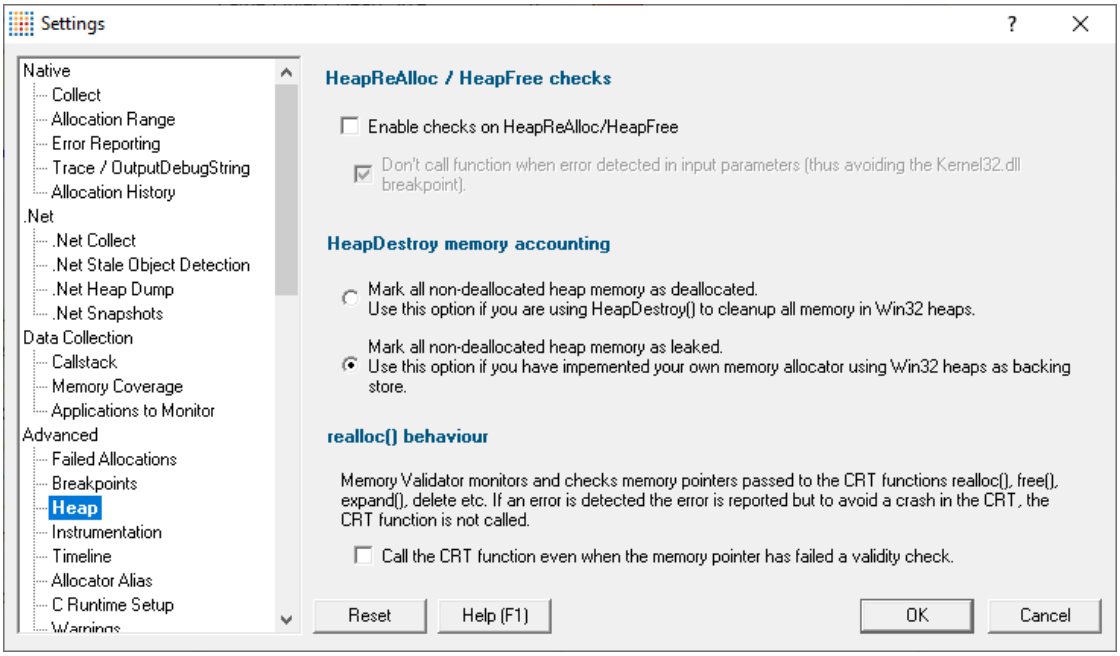
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.3 Heap

The **Heap** tab allows you to choose how Win32 Heaps are managed.

The picture below shows the default settings:



Checking for damaged memory before HeapReAlloc / HeapFree

Memory Validator can run background checks on the input pointers to `HeapReAlloc()` and `HeapFree()` for validity and report any errors.

- **Enable checks on HeapReAlloc and HeapFree** › enables detection (off by default)

When this is enabled you may wish to prevent those functions from being called when an invalid pointer is detected.

- **Don't call function when error detected** › prevent Win32 heap checking breakpoints from being triggered when an error is detected

  The reason you might not want to call the functions is that the Win32 heap checking code will force a debugger breakpoint instruction to be executed, which will either terminate your program or start a debugger to attach to the program. Avoiding these heap checks when Memory Validator already knows about the error will prevent the program run from being interrupted by a breakpoint.

How to handle non-deallocated memory when you destroy a Win32 heap using HeapDestroy

Memory Validator can interpret the call to HeapDestroy() in two ways.

- **Mark all non-deallocated memory as deallocted.** › use this if you are ignoring calls to HeapFree() because you will be calling HeapDestroy() at the end of the function. An example use case would be implementing a linear heap.

- **Mark all non-deallocated memory as leaked.** › use this if you are using Win32 heaps as the backing store for your own memory allocator.

The default option is to mark non-deallocated memory as leaked as this is the most common usage pattern.

Ignoring CRT calls after detecting damaged memory

Memory Validator inspects the memory pointers passed to `realloc()`, `_expand()`, `free()`, `delete` and `delete []`.

Any damage to the part of the heap that contains the memory pointer is reported, and normally, the intended CRT function call is ignored to prevent the application from the typically resulting crash.

- **Call the CRT function...** › allows the CRT function call to proceed anyway even if damaged memory has been detected (switched off by default)

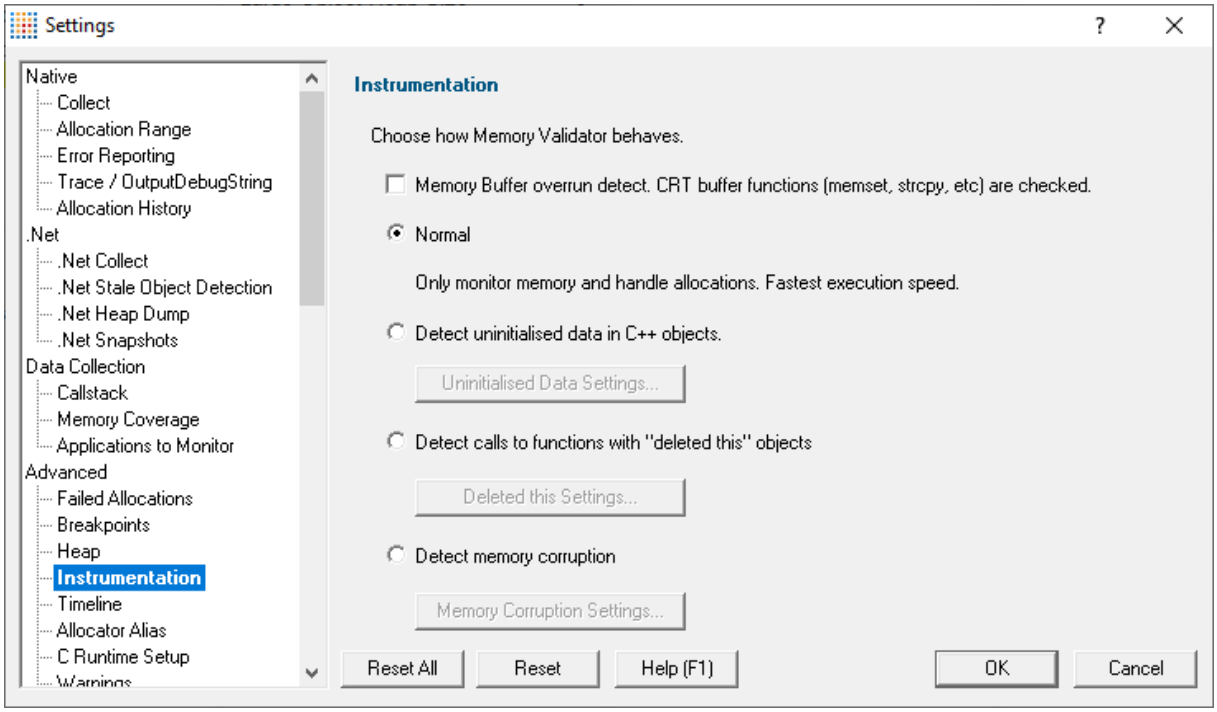
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.4 Instrumentation

The **Instrumentation** tab allows you to choose how Memory Validator monitors memory.

The picture below shows the default settings:



- **Memory Buffer overrun detect** › The standard CRT memory management functions (memset, strcpy, etc) are monitored for overruns and underruns.

There are four options to choose from. Each option is slower than the previous option.

- **Normal** › Memory allocations and handle allocations are monitored. No other checks are performed.
- **Detect uninitialised data in C++ objects** › When objects are created they are checked to ensure all data members are initialised. ***Debug mode applications only.***
- **Detect "deleted" this objects** › When methods are called the "this" pointer is checked to ensure it is valid.
- **Detect memory corruption** › The CRT heap is scanned every N function/method calls to check it's integrity. ***Debug mode applications only.***

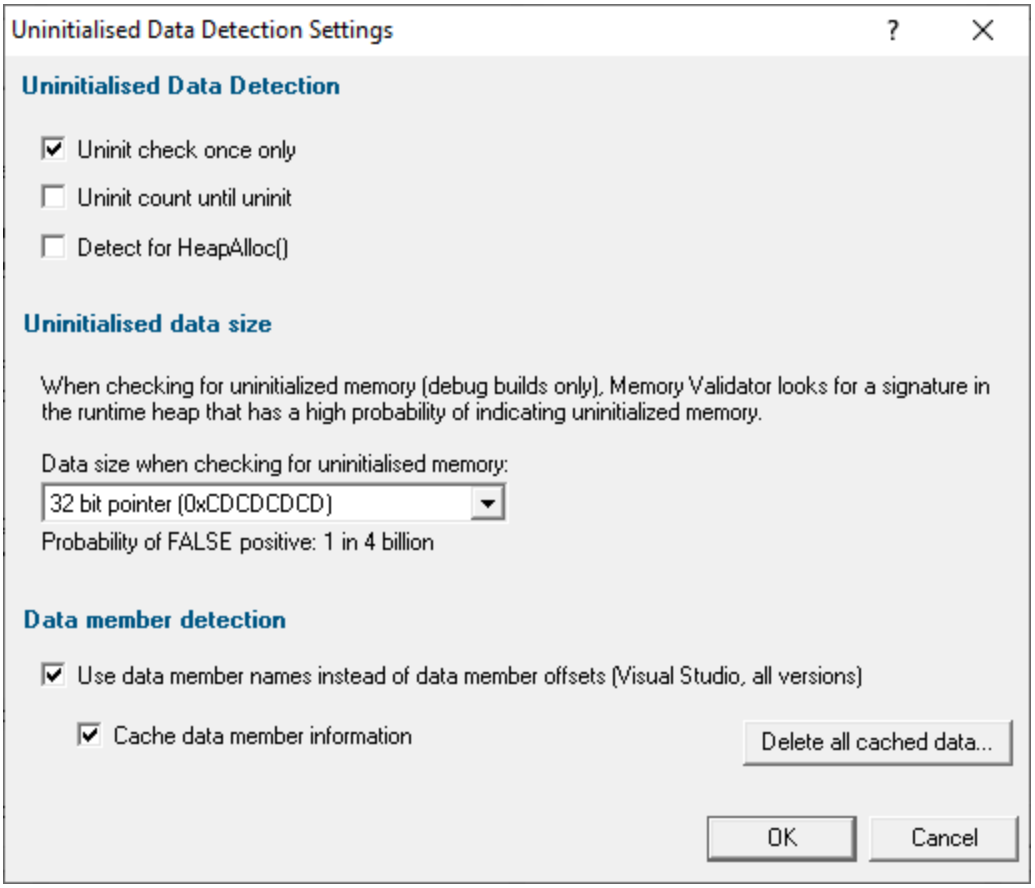
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.4.1 Uninitialised Data

The **Uninitialized Data** tab provides controls for determining how uninitialized data detection hooks are installed.

The default settings are shown below:



Uninitialized data detection

By default, uninitialized data checking is switched off, but you can turn it on and control how it works

- **Enable detection of uninitialized data** › detects uninitialized data in C++ objects in your application

Once enabled the additional options are

- **Uninit check once only** › check each object type once (the default), rather than every time their constructors are called

- **Uninit count until uninit** › check an object type for uninitialized data only until uninitialized data is detected

  The recommended option is to disable this and always check for uninitialized data irrespective of when an instance is found.

- **Detect for uninitialized HeapAlloc()** › enables detection of uninitialized data in C++ objects which have overridden the `new` operator and instead allocate from a heap using HeapAlloc()

  Your overridden `new` operator will need to initialise any allocated memory with the DWORD value `0xbaadf00d` for uninitialized data to be detected.

  ➡ See the source code for the example program that ships with Memory Validator for an implementation example in the class `heapNewBaseClass`.

Uninitialized data size

When Memory Validator is looking for uninitialized memory, it searches for bit patterns in memory that match the default value assigned by the debug Microsoft® C runtime heap:

- `0xCD` for BYTEs
- `0xCDCD` for WORDs
- `0xCDCDCDCD` for DWORDs and 32 bit pointers
- `0xCDCDCDCDCDCDCDCD` for QWORDs and 64 bit pointers

It may not be that uncommon to find a byte that has actually been *initialised* to `0xCD` - a 1 in 256 chance for random data for example.

It's less likely to find words similarly initialised to `0xCDCD` and really very unlikely to find such DWORDS (1 in 4 billion for random data) and QWORDS (1 in 16 quintillion).

If you are finding false positives however, where an initialised value is the same as the uninitialized default, you can choose which data size Memory Validator uses to detect uninitialized memory:

- **Data size when checking for unitialised memory** › choose a data size to match against from the list - BYTE, WORD, 32 bit pointer or 64 bit pointer

  The default is 32 bit pointer (same size as a DWORD) which normally minimizes the chance of random false positives.

  If you want to detect uninitialised pointers on 64 bit machines choose the 64 bit pointer option. *Option only available on 64 bit machines.*

Data member detection

Choose to use data member names or member offsets.

By default, the reporting of uninitialised data uses data member *names* which are cached between sessions, being recalculating only when the module containing the class is rebuilt.

However, it's not always possible to determine data member names, so on these occasions data member *offsets* are displayed.

You can choose to always display the offsets rather than the names:

- **Use data member names...** › unchecked: displays offsets all the time and doesn't use member names at all

  When use of data member names is enabled, you can control the caching:

  Cache data member information › unchecked: recalculates data member names every session, rather than the default of caching data on disk

  Delete all cached data › removes all cached data member information from disk

Reset All - Resets **all** global settings, not just those on the current page.

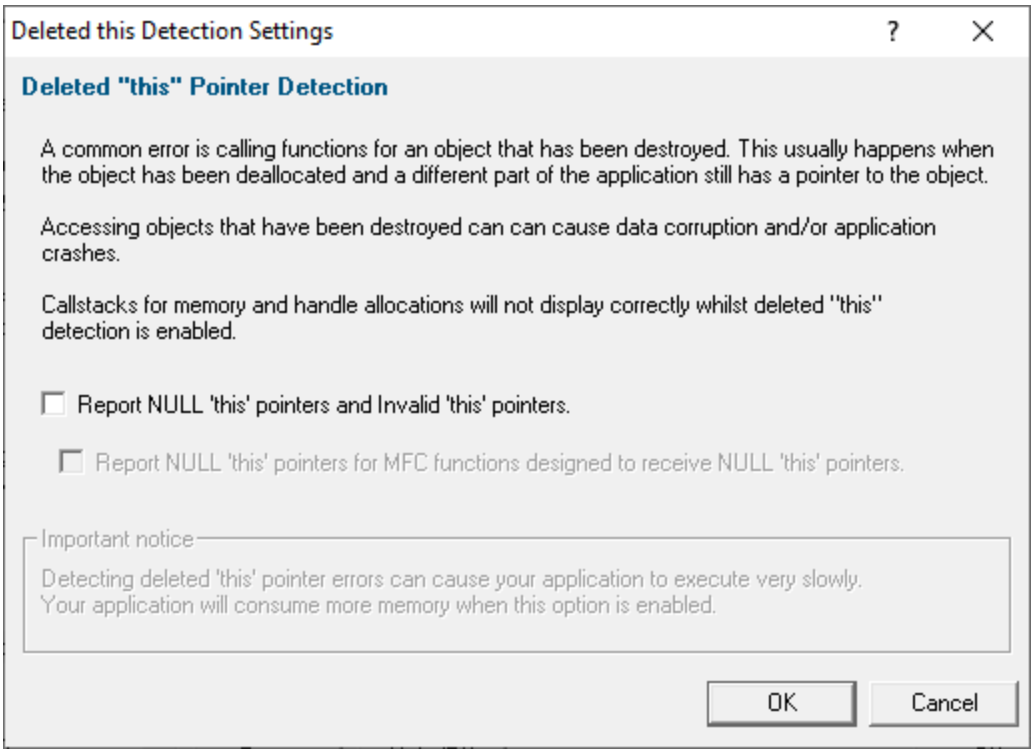
Reset - Resets the settings on the current page.

3.12.1.4.4.2 Deleted "this" Pointer

The **Deleted "this" Pointer** tab provides controls for determining how some deleted object hooks are installed relating to track functions called for deleted, NULL, or invalid 'this' pointers.

➡ The default setting is **not** to detect such errors.

The default settings are shown below:



Tracking the 'this' pointer

Calling functions that belong to a deleted object is a common error, especially if different parts of the application are holding the object pointer.

Memory Validator can check the `this` pointer of every C++ class member function to ensure that it is:

- not a pointer to a deallocated object
- not NULL
- does not point to a memory address that is not accessible

These checks are not enabled by default because it can significantly affect performance speed and memory usage.

Note that uninitialised data detection is disabled when this feature is enabled

- **Detect calling of functions for deleted C++ objects on the CRT heap** › enables detection of functions being called for deleted C++ objects

  When enabled you can additionally check for the following:

  - **Report NULL this pointers and Invalid this pointers** › enables detection of functions being called for NULL and invalid `this` pointers

    An Invalid `this` pointer is defined as one that does not point at the stack, memory in any DLL, or any other allocated memory.

You may want to see NULL `this` pointers for the specific MFC functions *designed* to receive NULL pointers (see list below):

- **Report NULL this pointers for MFC functions designed to receive NULL this pointers** enables the relevant reporting (disabled by default)

  This option is not generally recommended as you will typically see many such reports. However, there may be uses for this in specific cases, and we like to give you the option!

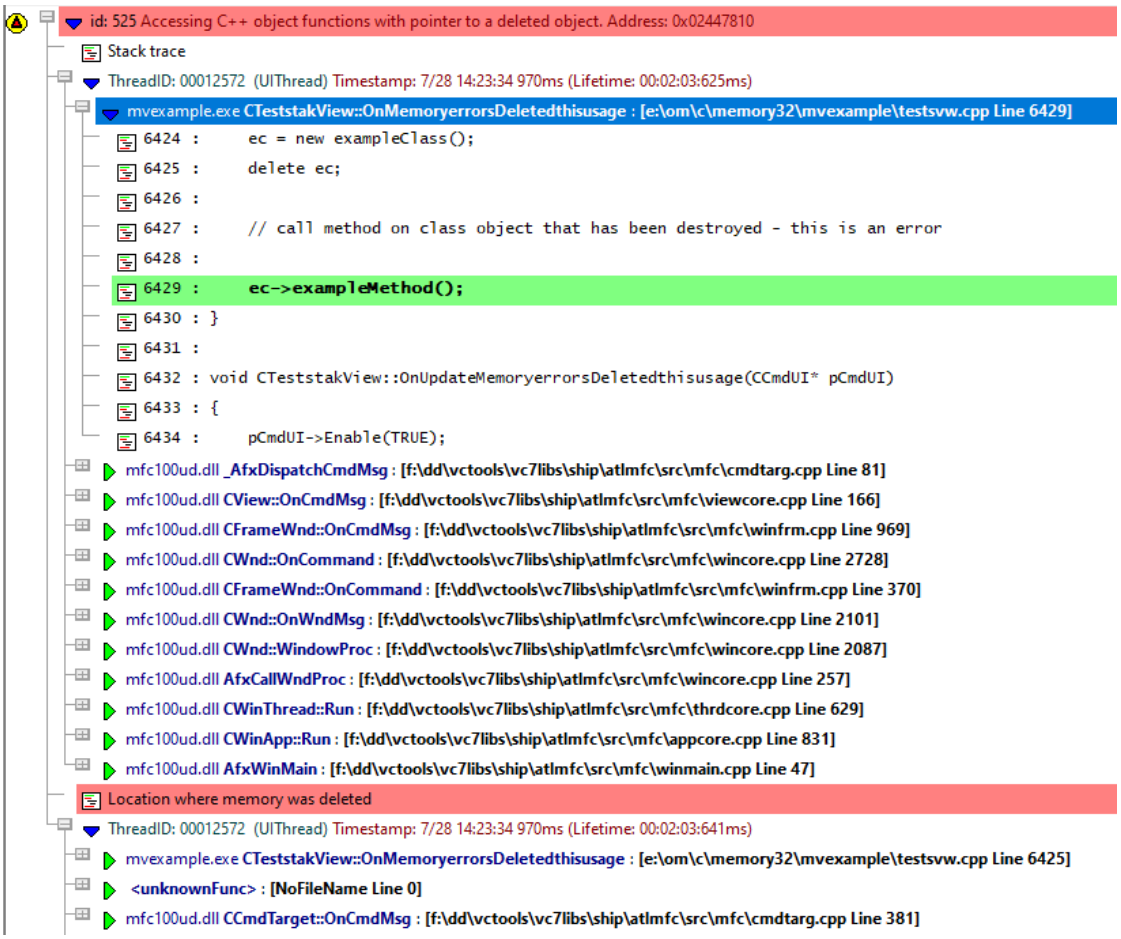
➜ Some example error scenarios can be found at the end of this section.

Displaying reported errors

When a deleted object has a function called and has been detected, the display in the Memory tab shows an entry to indicate where the error occurred.

The trace can be expanded to show the callstack and the source code.

In the example shown here, the `ex->exampleMethod()` has just been called, which has triggered the error report.



Some examples follow, and a list of MFC functions that are designed to be called with NULL `this` pointers.

Examples

Two examples are illustrated below which demonstrate common scenarios in which these errors occur.

Example 1 - calling a function for a deleted object:

Consider the following much simplified example code:

```
myObj  *obj;

// create an object, do some work and then destroy the object

obj = new myObj();
obj->doWork();
delete obj;

// at this point 'obj' is no longer valid,

... application does some more work

// another part of the application incorrectly uses the object that was deleted
earlier on (because
// the pointer wasn't reset to NULL, or more than one copy of the pointer
existed).

obj->doSomeMoreWork();
```

There is no guarantee that the call to `doSomeMoreWork()` will actually crash, which would otherwise alert you to the error.

If the memory pointed to by `obj` is still a valid memory address, then:

- reads of data will read undefined values
- writes may corrupt existing data, or
- writes may write to memory locations that will never be accessed by the application again, or that may be overwritten by any following memory allocation.

Enabling the features outlined above will help detect these errors.

Example 2 - MFC functions designed for NULL this pointers:

Consider the following snippet of example code that demonstrates an error:

```
CWnd *wnd;

wnd = getMyWindow(); // function that should return a valid window, but sometimes returns N
wnd->GetSafeHwnd();
```

The above code *looks* unsafe, as there is the potential to call GetSafeHwnd() with a NULL `this` pointer.

However, if you examine the MFC source or documentation for `CWnd::GetSafeHwnd()` you will see that this function is designed to *allow* it to be called with a NULL this pointer. See just below for a list of such functions.

As described above, Memory Validator allows you to detect functions being called with NULL `this` pointers, whilst also allowing you to ignore the selected MFC functions that are designed to be called with NULL `this` pointers.

MFC functions designed to be called with a NULL 'this' pointer

The list below details those MFC functions are designed to be called with NULL this pointers.

- `CHandleMap::DeleteTemp`
- `CImageList::GetSafeHandle`
- `CGdiObject::operator HGDIOBJ`
- `CGdiObject::GetSafeHandle`
- `CPen::operator HPEN`
- `CBrush::operator HBRUSH`
- `CFont::operator HFONT`
- `CBitmap::operator HBITMAP`
- `CPalette::operator HPALETTE`
- `CRgn::operator HRGN`
- `CDC::operator HDC`
- `CDC::GetSafeHdc`
- `CMenu::operator HMENU`
- `CMenu::GetSafeHmenu`
- `CWnd::operator HWND`
- `CWnd::GetSafeHwnd`
- `CWinThread::operator HANDLE`
- `CWinApp::DoMessageBox`
- `CDC::DPtoHIMETRIC`
- `CDC::HIMETRICtoDP`
- `CWnd::AttachControlSite`
- `CWnd::GetParentFrame`
- `CWnd::GetTopLevelParent`
- `CWnd::GetTopLevelOwner`
- `CWnd::GetParentOwner`
- `CWnd::GetTopLevelFrame`
- `CPlex::FreeDataChain`

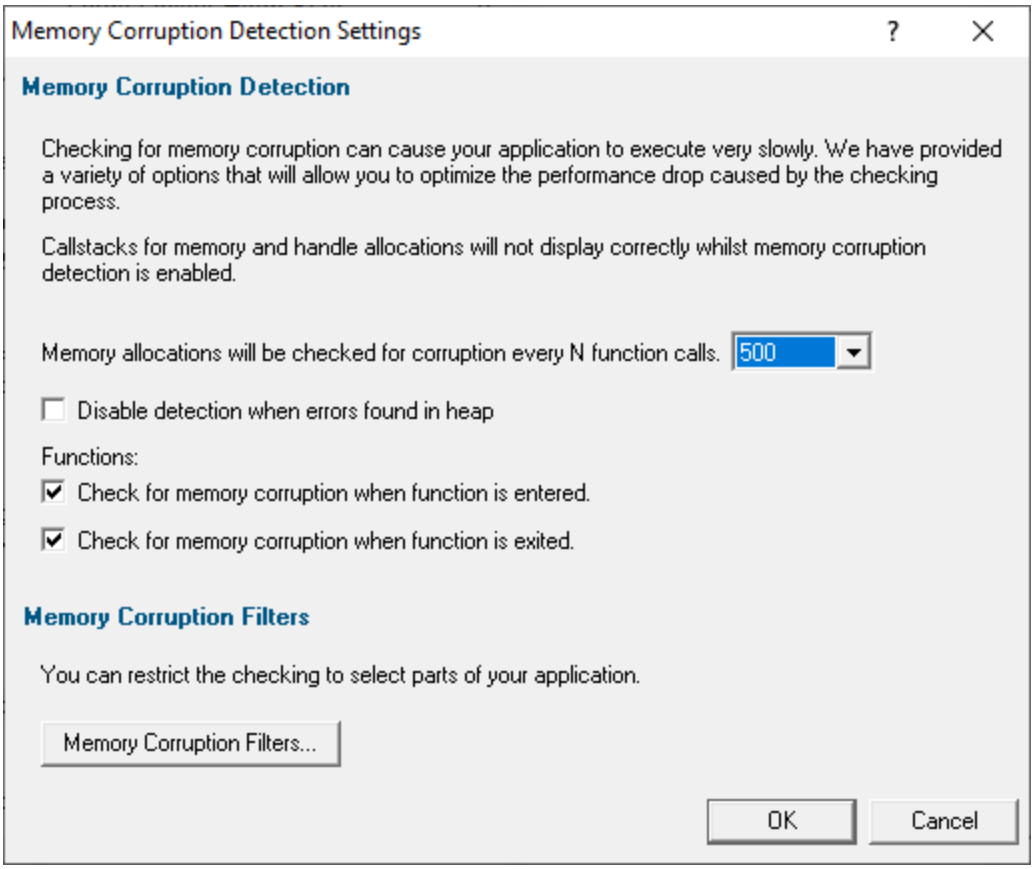
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.4.3 Memory Corruption Detection

The **Memory Corruption Detection** tab provides controls for detect memory corruption effectively while reducing the performance hit involved

The picture below shows this tab with the setting enabled, but the **default** setting is **not** to enable detection of memory corruption



How is memory corruption tested for?

Memory corruption is detected by visiting every memory allocation in the CRT heap and checking the entries in each location to see if the guard blocks on either side of the allocation are still valid.

In addition, any free blocks the CRT heap is still holding are also checked, to ensure that they have not been written to.

Note that this only works for DEBUG builds as information in the DEBUG heap is required to enable this detection.

Detecting memory corruption

Memory Validator can detect memory corruption at the entry and exit points of functions in your application.

- **Detect Memory Corruption** › enables detection

  Note that uninitialised data detection is disabled when this feature is enabled, and also callstacks for memory and handle allocations will not display correctly

  The default setting is **not** to detect memory corruption as it can significantly affect performance speed.
  Why does detecting memory curruption affect performance so much?Enabling detecting memory corruption will cause a serious performance degradation for your application. This is because, in general, the longer your application executes, the more memory the application has used. This means the CRT heap has more allocations in it, and thus the task of checking the CRT heap for corruption takes longer. For small applications, or applications that allocate small amounts of memory the performance hit will not be too great. For other applications the performance hit will be quite high.

Once detection is enabled, the other settings help lower the impact on performance by reducing frequency of checks:

- **Check for memory corruption every N functions calls** › reduces the performance hit by less frequent corruption checks

  The default is every 500 calls, but you can change it in the range 1 to 100,000.

  ➜ See below for how to use this setting effectively with the memory corruption filters which are also described below.

- **Disable detection when errors found in heap** › stops detecting more memory corruption errors once an error has been found.

  The default is continue detection even after errors are found (i.e. the checkbox is off by default)

- **Check for memory corruption when function is...**

  ...entered › perform checks *before* executing a function
  ...exited › perform checks *after* execution

Understanding the memory corruption results

When a memory corruption is detected, the main Memory tab shows the following items:

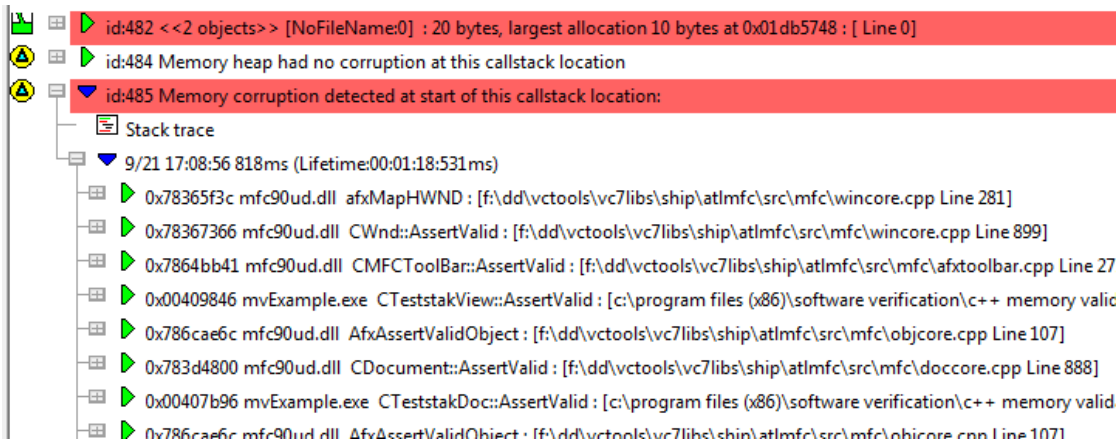
- some data to indicate the memory allocations that have been corrupted (one line may relate to one or more corruptions)
- a last known callstack *prior* to this corruption being detected
- a callstack of the location when the corruption was detected

The two callstacks help to identify the area of the program between them as the potential cause of the memory corruption.

In the example below line id:482 shows that two objects of 10 bytes were corrupted, followed by a last known good and bad callstack for each corruption detected.

The known *good* callstack at line id:484 will show the last location in the program at which the CRT heap was known *not* to have been corrupted (or if already corrupted, then at least not corrupted any more!).

Finally, the known *bad* callstack at line id:485 is shown expanded and indicates the location at which the memory corruption was *detected.* Note that this is not necessarily the point at which memory was *actually* corrupted, as this depends on the value of **N** described above.



Because line id:482 related to two corruption detections, another good and bad callstack would be displayed below those shown here.

Memory corruption filters

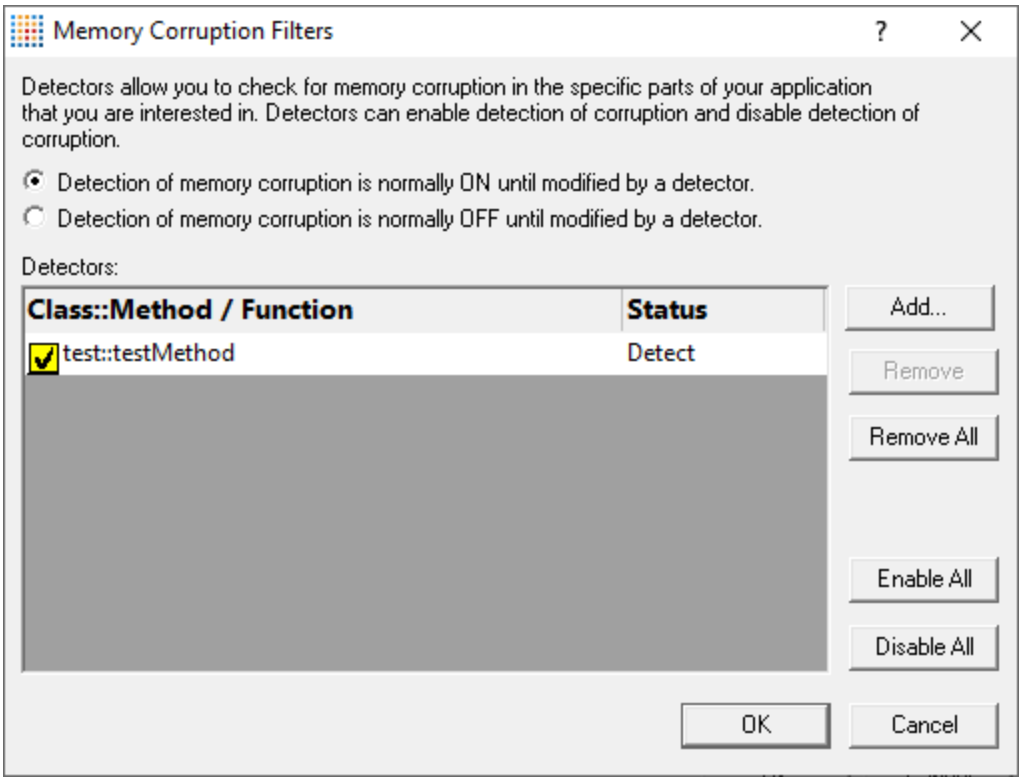
Because of the performance hit of detecting memory corruption, it can be advisable to constrain detection to only part of an application.

To do this, you specify 'detectors' that switch the checking on and off.

A detector specifies a function name to which it applies and a status that applies when the program reaches the detector. When the program *leaves* the detector function, the previous status is applied.

Using the yellow check box, detectors can be disabled so as not to affect the detection status.

- **Memory Corruption Filters...** › display the Memory Corruption Filters dialog (see below)



Using memory corruption filters

- **Detection of memory corruption is normally ON / OFF until modified by a detector** › sets the default detection state

- **Add...** › enter **function**, or **class::method** in column 1 › Click column 2 to add the state (default is **Detect**), double click to change it to **Don't Detect**.

  Detectors are initially enabled. To disable a detector, use the yellow check box.

  The function name can take the forms below - no need to specify return types or argument types

  - **className::methodName** Matches class name *and* method name exactly
  - **className::** Any method in class **className**
  - **::methodName** Matches method **methodName** in any class
  - **functionName** Matches function name only

- **Remove** › removes selected detectors from the list
- **Remove All** › removes all detectors from the list

- **Enable All** › enables all detectors
- **Disable All** › disables all detectors

  These don't affect the Detect status in column 2

Note that data collection is still subject to the global data collection flags, so turn global data collection off also turns off memory corruption detection

Example of using memory corruption filters

Consider an application where only the `myApp::sortThisData()` function, and all functions called from it are of interest:

> Choose **Detection of memory corruption is normally OFF until modified by a detector** › sets default checking state to OFF
>
> Then **Add...** › enter `myApp::sortThisData()` in first column of the table › click in the second column to choose **Detect**

When the application reaches `myApp::sortThisData()` the detection of memory corruption is turned ON because of the **Detect** status.

The application runs as normal, and memory corruption is detected, until `myApp::sortThisData()` finishes executing.

At this point the previous detection status (OFF) is restored.

Using the filters and the '*check every N functions*' settings together

Setting N to 1 checks the CRT heap at every function entry and exit and corruption will be detected very close to the actual line of source code that caused the problem. However, the performance hit will be high - huge in fact, for larger applications.

Setting N to, say 100, means the performance hit will be lower, but up to 100 functions may have been called between the memory corruption and its detection.

One strategy might be to start with a high value for N to get a feel for which part of the application is causing the corruption.

Then hone in on the corruption by using the memory corruption filters to focus on that part of the application, and decrease N to 1 so that every function entry and exit is monitored inside the area covered by the filters.

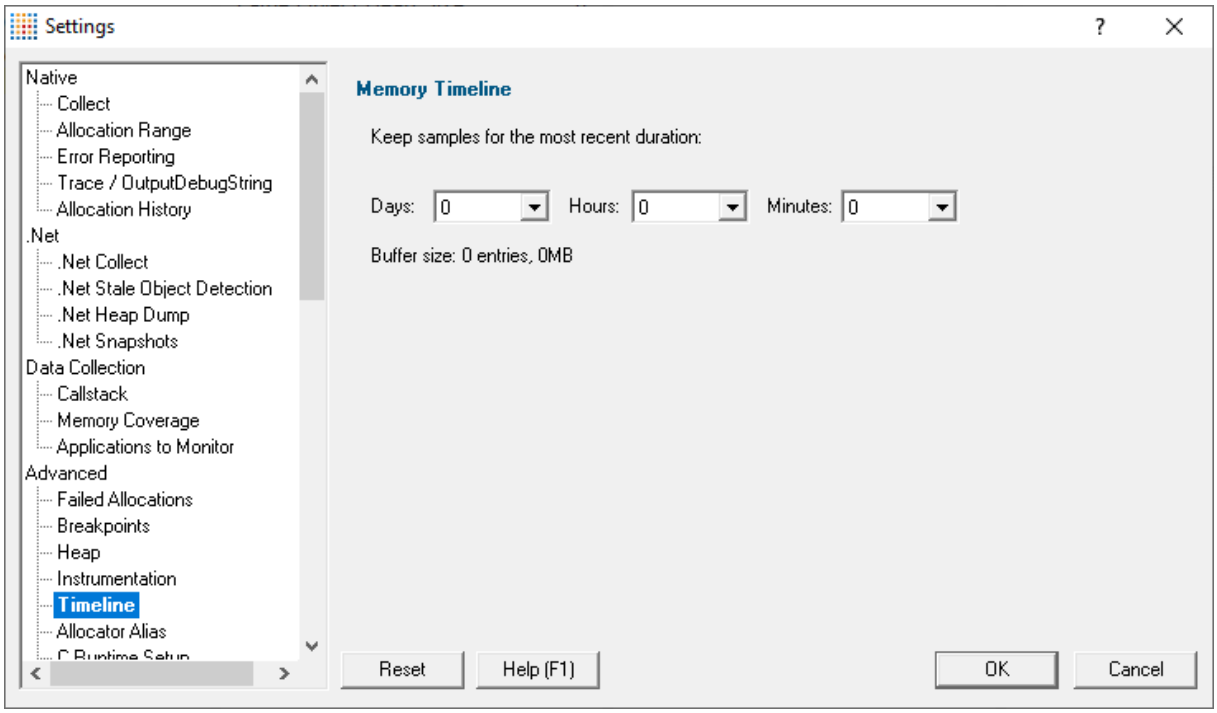
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.5 Timeline

The **Timeline** page allows you to control how much data Memory Validator will store for displaying on the Timeline.

The default settings are shown below:



Memory timeline duration

The memory and handle timeline samples data every second while your application is running. For long running programs, that can mean storing a large amount of data.

To prevent too much storage being used, you can specify the maximum as a time duration in **Days**, **Hours**, and **Minutes** for the most recent group of samples.

Once more samples have been collected than fit in that time duration the oldest samples will be discarded. The default is 1 hour.

Reset All - Resets **all** global settings, not just those on the current page.

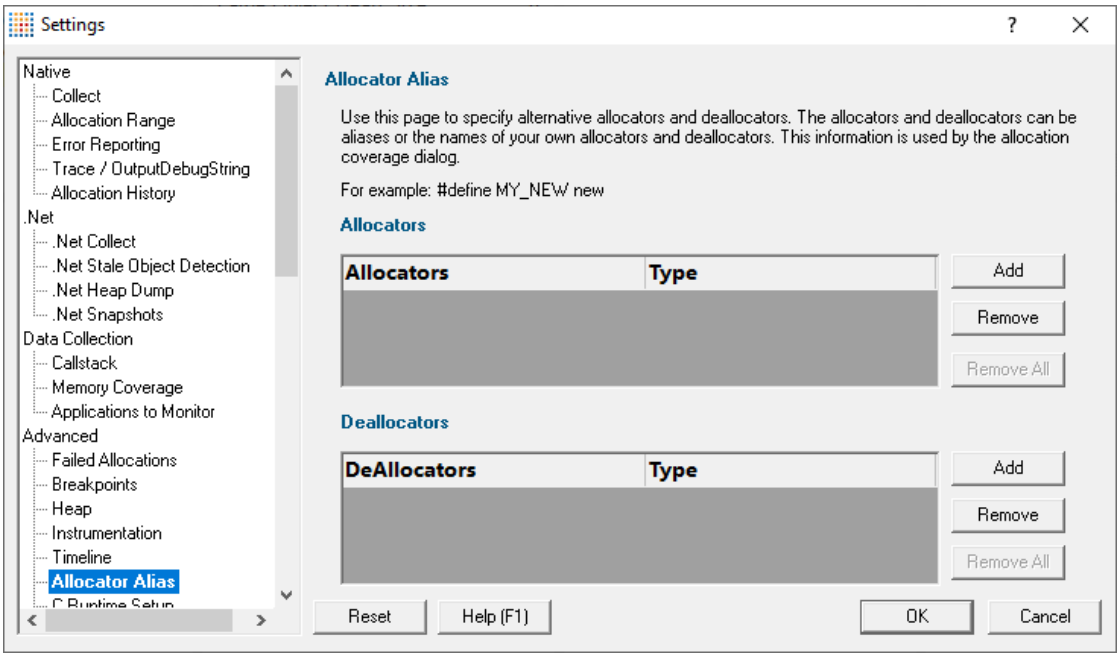
Reset - Resets the settings on the current page.

3.12.1.4.6 Allocator Alias

The **Allocator Alias** tab allows you to specify any custom or in-house MACROs used to wrap C/C++ keywords and functions such as `new`, `delete`, `malloc`, `realloc`, `free`, etc.

This feature was added to improve source code parsing for the case when users have wrapped the keywords.

The default behaviour is to have no aliases set, and if you don't wrap the C/C++ keywords in this way, you won't need to define any allocator aliases here.



The image above shows an example where `MY_NEW`, `MY_NEW_ARRAY`, `MY_DELETE` and `MY_DELETE_ARRAY` have been used to wrap the C/C++ keywords `new`, `new []`, `delete` and `delete []`.

Note: when using allocator and deallocator alias macros, ensure that the macro and all arguments to the macro are on **the same source code line**. Macros that span lines will not be parsed correctly. This is because the current parser works a line at a time.

Allocators

To add a custom allocator, use the top table:

- **Add** › enter your custom macro name in the **Allocators** column › double click in the **Type** column › Choose the matching **Allocator Type** from the list

  The allocator type can be one of the following:

  - **NEW** - `new()`
  - **NEW [ ] ARRAY** - `new []`
  - **MALLOC** - `malloc()`
  - **CALLOC** - `calloc()`
  - **REALLOC** - `realloc()`
  - **EXPAND** - `_expand()`
  - **HEAP_ALLOC** - `HeapAlloc()`
  - **HEAP_REALLOC** - `HeapReAlloc()`
  - **GLOBAL_ALLOC** - `GlobalAlloc()`
  - **GLOBAL_REALLOC** - `GlobalReAlloc()`
  - **LOCAL_ALLOC** - `LocalAlloc()`
  - **VIRTUAL_ALLOC** - `VirtualAlloc()`
  - **VIRTUAL_REALLOC** - `VirtualReAlloc()`
  - **COTASKMEM_ALLOC** - `CoTaskMemAlloc()`
  - **COTASKMEM_REALLOC** - `CoTaskMemReAlloc()`
  - **SYS_ALLOC_STRING** - `SysAllocString()`
  - **SYS_ALLOC_STRING_BYTE_LEN** - `SysAllocStringByteLen()`
  - **SYS_ALLOC_STRING_LEN** - `SysAllocStringLen()`
  - **SYS_REALLOC_STRING** - `SysReallocString()`
  - **SYS_REALLOC_STRING_LEN** - `SysReallocString()`
  - **USER_ALLOC** - see below
  - **USER_REALLOC** - see below
  - **USER_ALLOC_ARG1...10** - see below

- **LOCAL_REALLOC** - `LocalReAlloc()`

USER_ALLOC - is a substitute for a user specified alloc where your macro simply replaces a keyword (such as `new`, `malloc`, etc)
USER_REALLOC - is for user specified realloc
USER_ALLOC_ARG1...10 - is for a user specified alloc where your macro takes arguments and one of the arguments 1...10 is the allocation. See the end of this topic for example code.

Deallocators

Managing deallocator alias works in the same was as allocators above, but via the lower table, with deallocator types being one of the following:

- **DELETE** - `delete`
- **DELETE [] ARRAY** - `delete []`
- **FREE** - `free()`
- **HEAP_FREE** - `HeapFree()`
- **GLOBAL_FREE** - `GlobalFree()`
- **LOCAL_FREE** - `LocalFree()`
- **VIRTUAL_FREE** - `VirtualFree()`
- **COTASKMEM_FREE** - `CoTaskMemFree()`
- **SYS_FREE_STRING** - `SysFreeString()`
- **VARIANT_CLEAR** - `VarianctClear()`
- **USER_FREE** - user specified free

Removing aliases

You can remove either selected alias or all aliases from each table

- **Select** one or more items in the list › **Remove** › removes all selected allocator aliases
- **Remove All** › removes all the allocators in the table

Reset All - Resets **all** global settings but **does not remove** the allocator and deallocator aliases set on this page of the settings dialog.

Reset - Resets the settings on the current page.

Example use of the USER_ALLOC_ARG1...10 allocator alias

For the USER_ALLOC_ARG1...10 allocator aliases, you would use these in the situation where you have defined a macro that takes arguments, one of which is the allocation.

For example, consider the following code:

```
#define PROTECTED_ALLOC1(_allocExpression, _p, dummyArg) \
        try                                              \
        {                                                \
                (_p) = (_allocExpression);               \
        }                                                \
```

```
        catch(...)                                          \
        {                                                   \
                (_p) = NULL;                                \
        }

#define PROTECTED_ALLOC2( _p, _allocExpression, dummyArg) \
        try                                                 \
        {                                                   \
                (_p) = (_allocExpression);                  \
        }                                                   \
        catch(...)                                          \
        {                                                   \
                (_p) = NULL;                                \
        }

        allocateInSideThis   *ec1;
        allocateInSideThis   *ec2;
        char                        *someData;

        PROTECTED_ALLOC1(new allocateInSideThis, ec1, false);
        PROTECTED_ALLOC1(new allocateInSideThis[4], ec2, false);
        PROTECTED_ALLOC1((char *)malloc(31), someData, true);

        // do some work...

        MY_DELETE ec1;       // this one was a single value
        MY_DELETE [] ec2;    // this one was an array
        MY_FREE(someData);   // this one was malloc

        PROTECTED_ALLOC2(ec1, new allocateInSideThis, false);
        PROTECTED_ALLOC2(ec2, new allocateInSideThis[4], false);
        PROTECTED_ALLOC2(someData, (char *)malloc(31), true);

        // do some work...

        MY_DELETE ec1;       // this one was a single value
        MY_DELETE [] ec2;    // this one was an array
        MY_FREE(someData);   // this one was malloc
```

To use the above macros, you would:

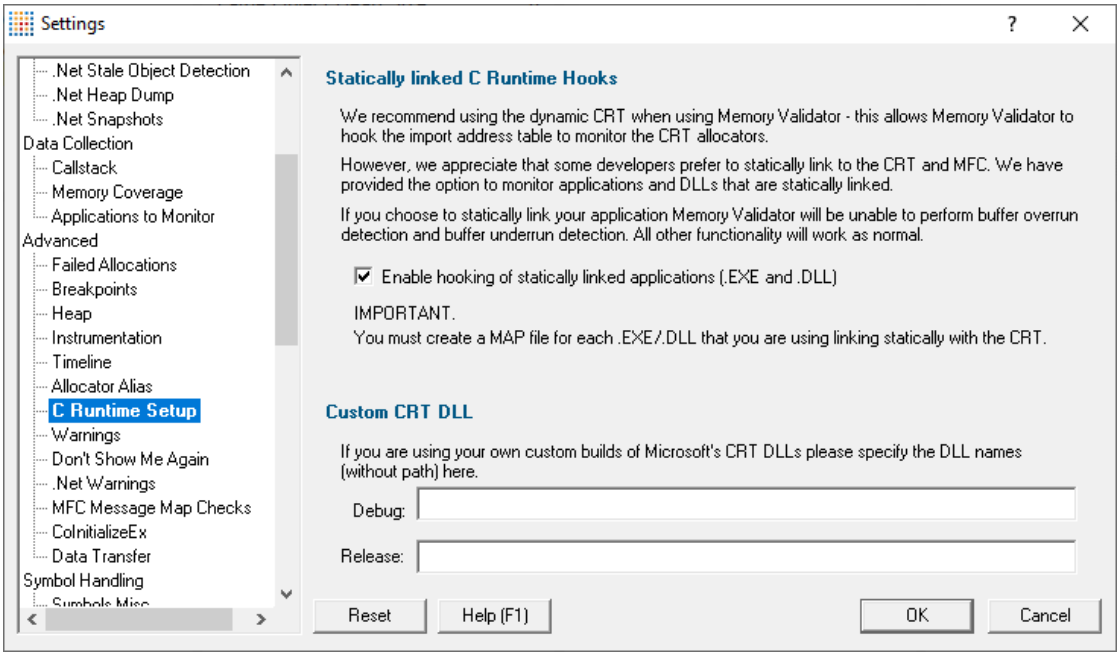
- define `PROTECTED_ALLOC1` as a **USER_ALLOC_ARG1** since it uses the *first* argument for the allocation
- define `PROTECTED_ALLOC2` as a **USER_ALLOC_ARG2,** since it uses the *second* argument

3.12.1.4.7 C Runtime Setup

The **C Runtime Setup** tab allows you to control whether hooking of statically linked applications is enabled, or even specify a custom CRT build

➡ Read more about dynamic and statically linked CRT libraries, map fles, and related compiler options in the before you start section.

Note that where possible, we recommend that you link your program with the *dynamic* C runtime library for use with Memory Validator.



Statically linked C Runtime Hooks

Just to reiterate from above: when monitoring programs written with Microsoft Visual Studio, the best performance is achieved when those programs are *dynamically* linked to the C runtime libraries.

For applications built using the *statically* linked C runtime libraries, support is provided by using the debugging symbol table to locate the functions.

- **Enable hooking of statically linked applications (.EXE and .DLL)** › enables use of the static C runtime library support (the default)

Caveats with using statically linked C runtime libraries

When running Memory Validator with applications using statically linked C runtime libraries:

- Buffer overrun detection is not available
- Buffer underrun detection is not available
- For release builds using the statically linked C runtime library, Microsoft do not emit symbols for all memory allocators/deallocators - making it impossible to monitor them in such builds
- Performance of applications will be slower, since the hooking mechanism is more complex

Requirements for using statically linked C runtime libraries

Using Microsoft?

For Microsoft runtimes, we always use the PDB file where possible, only using a map file if no PDB file containing symbols is available.

In the event PDB files are not available, you'll need to generate a map file for each statically linked DLL/EXE that you wish to monitor. The map file must have the same name as the DLL/EXE with the extension ".map". For example, if you are building the DLL "example.dll", the map file name should be "example.map".

Using Borland Dephi, Borland C++?

This topic does not apply. Memory Validator can handle statically linked applications correctly. Just ensure you create appropriate TDS symbols and detailed map files.

Using Metrowerks C/C++, Visual Basic, or Fortran95?

This topic does not apply.

Custom CRT DLL

If you have rebuilt your Microsoft C Runtime DLLs, you can specify the names (just names, no path) of the Debug and Release DLLs in the two fields provided.

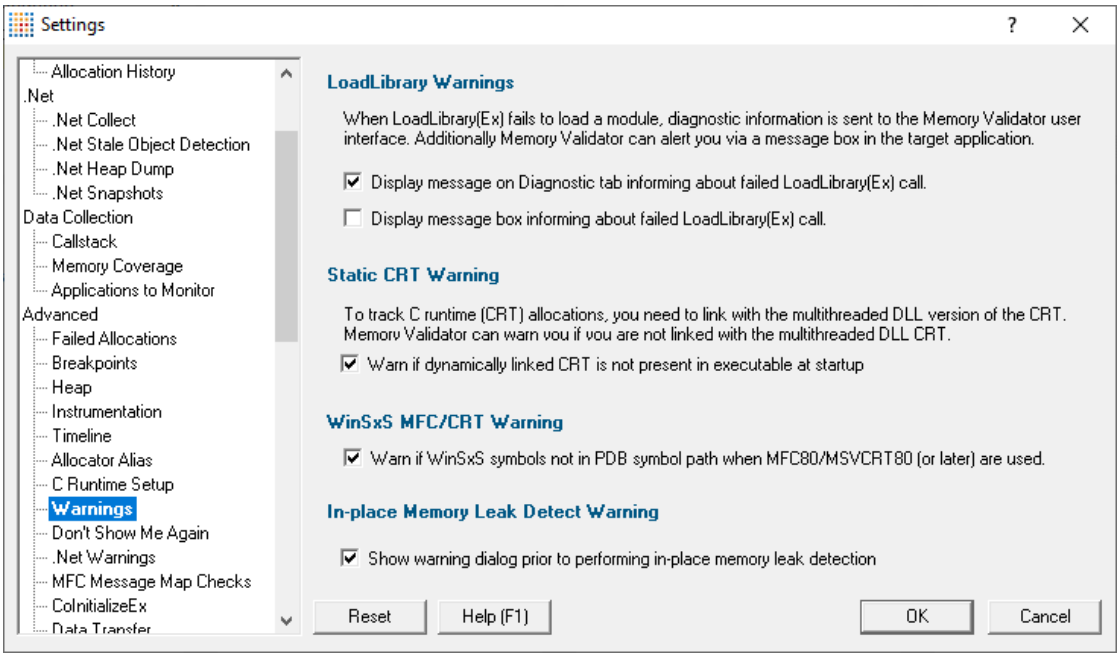
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.8 Warning

The **Warning** page allows you to control warnings that Memory Validator can display.

The default settings are shown below:



LoadLibrary Warnings

When you want to know if a module is failing to load and why, you can be informed on the diagnostics tab or via a message box:

- **Display message on Diagnostic tab informing about failed LoadLibrary(Ex) call** › shows diagnostic message (on by default)
- **Display message box informing ...** › shows warning message dialog (off by default)

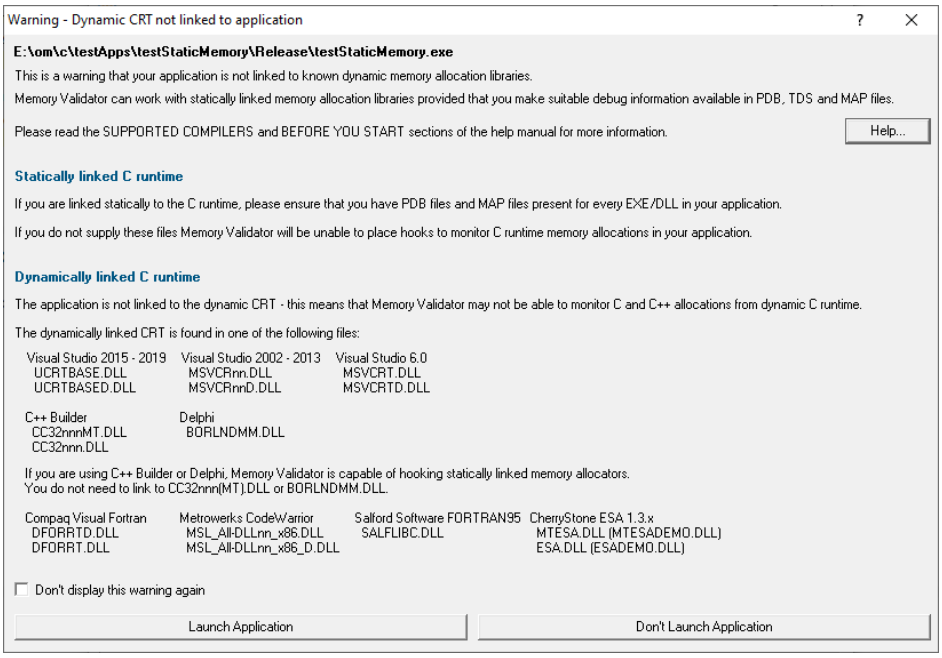
  This message box is launched in the target application as a convenience for attaching your debugger before clicking the OK button.

Static CRT Warning

Memory Validator instruments the C runtime heap by monitoring calls to the dynamically linked CRT.

- **Warn if dynamically linked CRT is not present ...** › shows a warning dialog (example below) if the application starting up is using static rather than dynamically linked CRT

  ➡ See the before you start section for more information about the dynamic and static CRTs.



WinSxS MFC80/MSVCRT80 Warning

This warning applies to all MFC/VC++ from MFC80 (Visual Studio 2005) onwards through to Visual Studio 2014 (MFC140(u)(d).dll/APPCRT140(U)(D).DLL)

When working with these versions, dlls are installed as side by side DLLs in the c:\windows\WinSxS folder (or equivalent).

The symbols for these DLLs are not kept in the same folder and will not be found automatically by Memory Validator.

A warning is optionally shown indicating the PDB symbol search path may be incomplete if the c:\windows\symbols\dll folder (or equivalent) is not part of the PDB symbol path when VC or MFC DLLs are used.

- **Warn if WinSxS symbols not in PDB symbol path ...** › shows a warning that the PDB symbol search path may be incomplete (on by default)

In-place Memory Leak Detect Warning

This warning is displayed when an in-place memory leak detection is started. The warning informs you how many pointers and/or handles will be checked (the more there are, the longer it takes).

- **Show warning dialog prior to performing ...** › shows a warning that an in-place memory leak detect is about to take place.

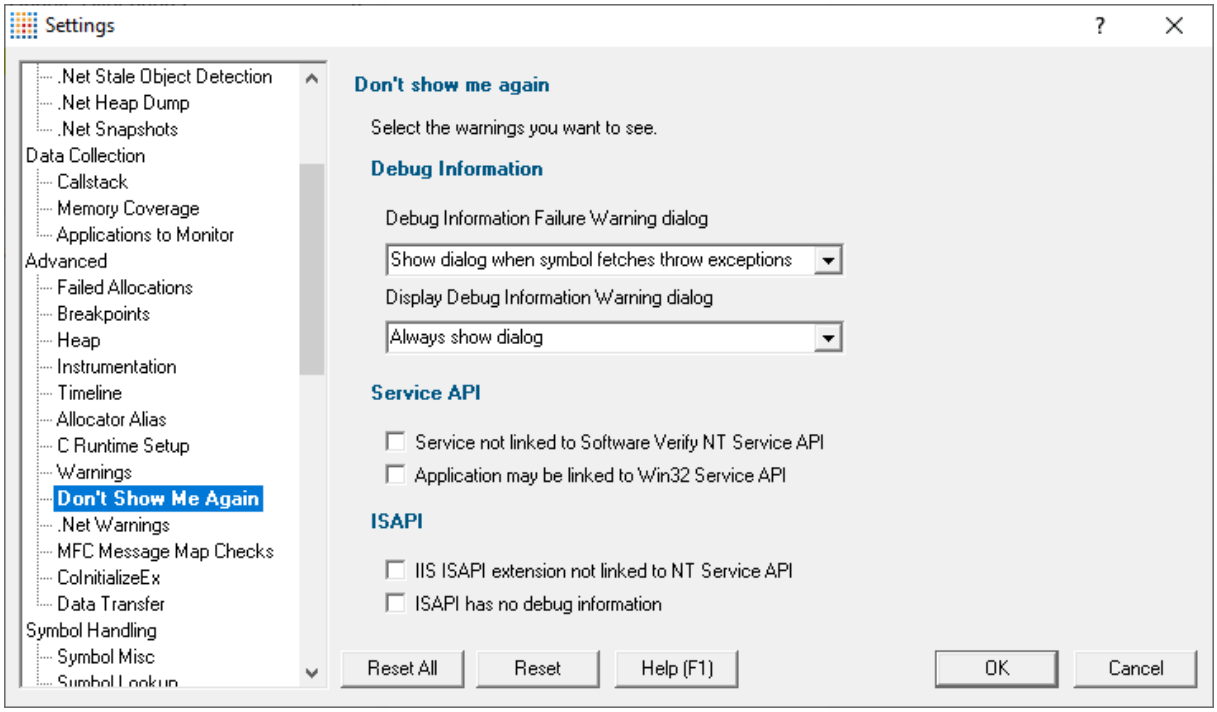
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.9 Don't Show Me Again

The **Don't Show Me Again** page allows you to control warnings that Memory Validator displays.

The default settings are shown below:



Debug Information

Debug Information Failure Warning

When there is a failure collecting debug symbols a warning can be displayed. The options are:
- Always show dialog
- Never show dialog
- Show dialog when symbol fetches throw exceptions

Display Debug Information Warning

When no debug information is available for at least one module a warning can be displayed. The options are:
- Always show dialog
- Never show dialog
- Show dialog when debug information is missing

Services API

- **Service not linked to Software Verify NT Service API** › warning will be shown if you try to monitor a service not linked to the Software Verify NT Service API. (on by default)

  When trying to monitor a service Memory Validator can detect if the service is not linked to the NT Service API and display a warning.

  It is possible to use the service API without linking to it (use GetProcAddress() to lookup the functions and call them) - in this case you would want to turn this warning off.

- **Application may be linked to Win32 Service API** › warning will be shown if you try to start an application that appears to be a service - it uses Win32 Service APIs. (on by default)

ISAPI

- **IIS ISAPI extension not linked to NT Service API** › warn when using an ISAPI not linked to the NT Service API

When trying to monitor ISAPI extensions Memory Validator can detect if the ISAPI is not linked to the NT Service API and display a warning.

It is possible to use the NT Service API without linking to it (use GetProcAddress() to lookup the functions and call them) - in this case you would want to turn this warning off.

- **ISAPI has no debug information** › warn when using an ISAPI that has no debug information

Memory Validator can warn if the ISAPI has no debug information. There may be cases where you don't want to see this warning.

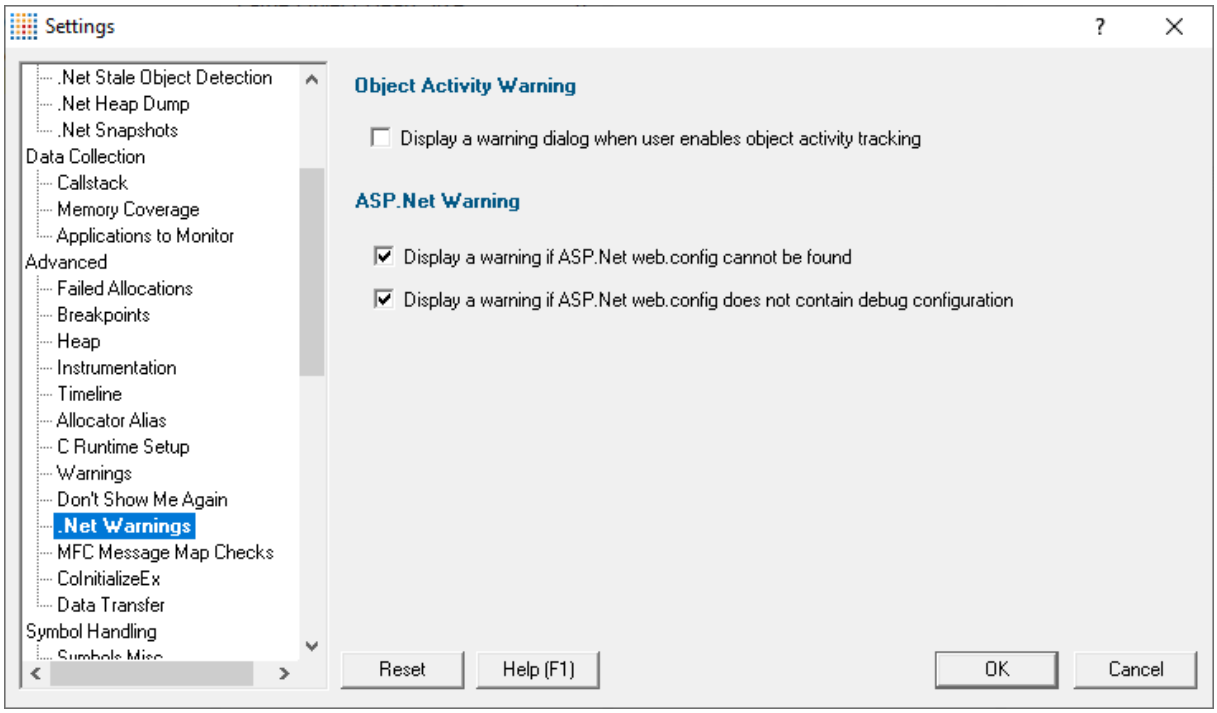
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.10 .Net Warnings

The **.Net Warnings** page allows you to control .Net related warnings that Memory Validator displays.

The default settings are shown below:

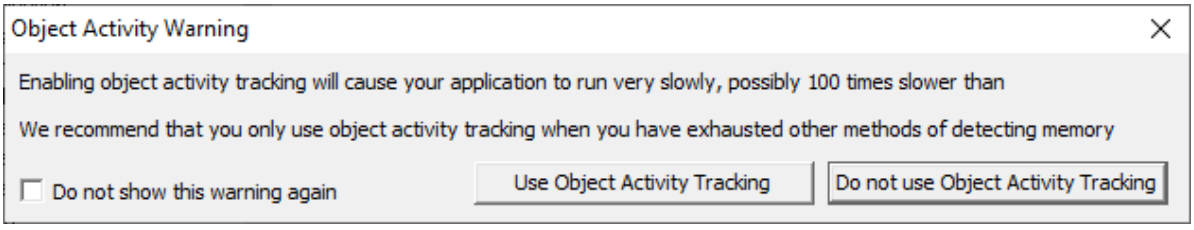


Object Activity Warning

Collecting object activity statistics to help determine which .Net objects have not been used and therefore might be leaked is an expensive activity. It will slow your program performance. Because of this Memory Validator can warn you when you enable this option so that you don't enable it and then wonder what happened to your application's performance.

- **Display a warning dialog when user enables object activity tracking** › does what it says on the tin!

When this warning is enabled and the user tries to enable object activity tracking they are shown this warning as a reminder of the performance penalty.



ASP.Net Warning

There are a couple of useful ASP.Net configuration warnings which Memory Validator can display.

- **Display a warning if ASP.Net web.config cannot be found** › The web.config affects how your ASP.Net website behaves. If this isn't present the website may not display correctly.
- **Display a warning if ASP.Net web.config does not contain debug configuration** › Without the debug configuration debug information won't be available.

Reset All - Resets **all** global settings, not just those on the current page.

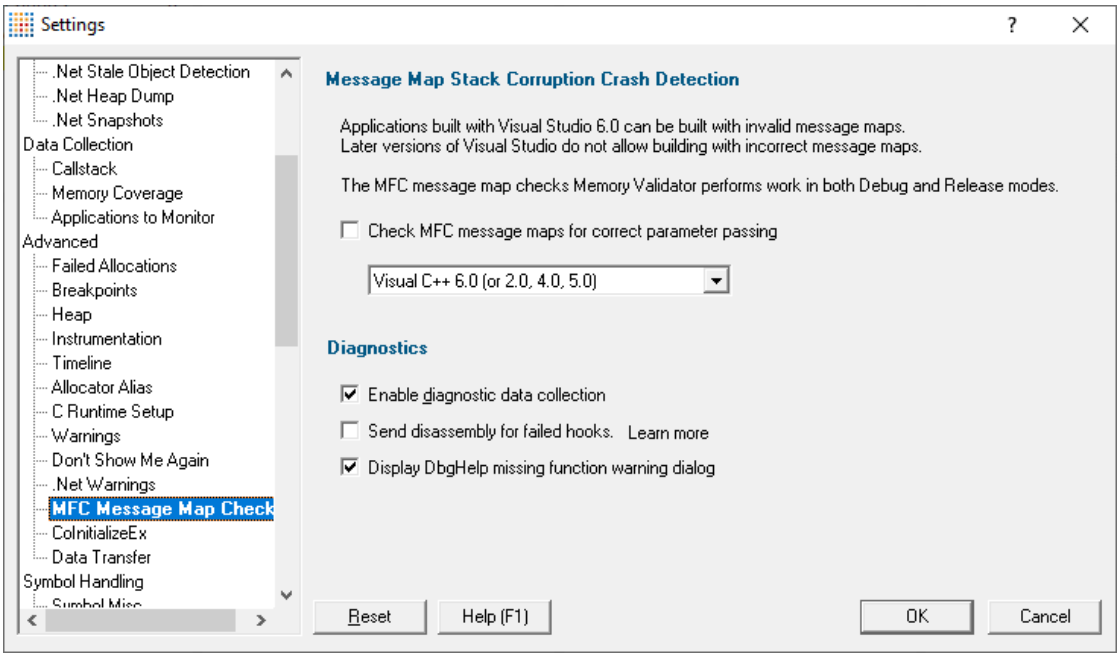
Reset - Resets the settings on the current page.

3.12.1.4.1 MFC Message Map Checks

The **MFC Message Map Checks** tab support validation of function signatures for functions called by MFC message map processing.

This option is not relevant if using Visual Studio .Net or VC++ 7.0 or above.

The default settings are shown below:



Checking MFC message map parameter counts

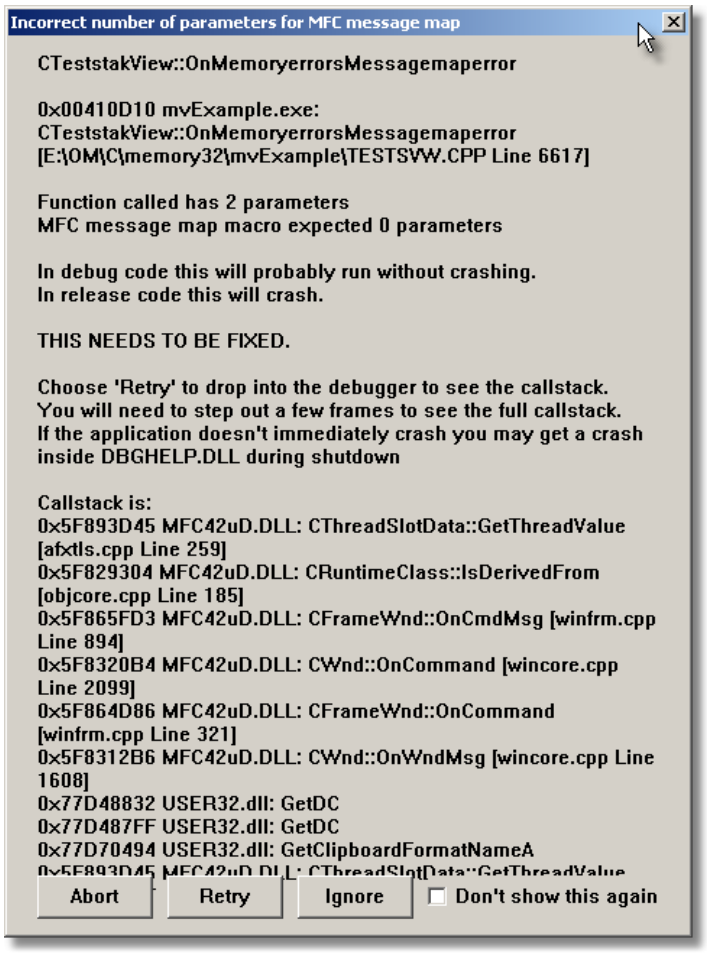
Enabling message map stack corruption detection will check message map function parameter counts called in release mode and debug mode.

- **Check MFC message maps for correct parameter passing** › enable the message map parameter checks (off by default)

If enabled, ensure you also choose the correct Visual C++ version from the list to ensure errors are reported correctly

Message map error dialog

When an error is detected, a dialog is displayed. An example is shown below. The example shows that a handler called `OnMemoryerrorsMessagemaperror` with 2 parameters was called when 0 were expected. If a file and line number for the function can be identified, they will be displayed.



If you choose **Retry**, an int 3 breakpoint instruction will be executed, causing the program to either stop in the debugger, or start the debugger if Just-In-Time debugging is enabled.

You may not be able to see the full callstack, and will need to step out 3 or 4 levels (out of SvlMemoryValidatorStub.dll) before the full callstack is displayed.

This option is not relevant if using Visual Studio .Net or VC++ 7.0 or above. In the test example application, the *Message Map Error* button in the *Memory Errors* menu will be disabled for builds with newer compilers

How MFC message map can be a problem

If you specify the wrong function signature (by specifying too many or too few parameters), the code will execute without error in Debug mode, but will crash with a corrupt stack in Release mode.

For example, a button handler would be specified in the message map as:-

```
BEGIN_MESSAGE_MAP(CtheDlg, CDialog)
    //{{AFX_MSG_MAP(CtheDlg)
    ON_BN_CLICKED(IDC_BUTTON_BROWSE, OnButtonBrowse)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The function prototype would be:

```
void OnButtonBrowse();
```

and the implementation would be

```
void CtheDlg::OnButtonBrowse()
{
    // do stuff here
}
```

Consider if however, this was specified as:

```
void OnButtonBrowse(void       *anArg);

void CtheDlg::OnButtonBrowse(void     *anArg)
{
    // do stuff here
}
```

A crash would happen with a corrupted stack after the function `OnButtonBrowse` executes. This is because the `anArg` argument is popped off the stack by the `OnButtonBrowse()` function, although the calling MFC message map dispatcher did not push the parameter onto the stack.

This causes very hard-to-identify "release only" bugs.

In debug mode, the error is still present, but the way MFC processes message maps means that in debug mode, this error does not result in a crash.

Diagnostics

Collection: A lot of diagnostic information is collected and displayed on the diagnostic tab when attaching to a target program.

Some of this information is *always* sent to Memory Validator, but if you may not want to see it all.

- **Enable diagnostic data collection** › displays all diagnostic information in the diagnostic tab (on by default)

Disassembly: When hooking functions, some functions cannot be hooked due to the object code that corresponds to the source code location.

- **Send disassembly for failed hooks** › shows the disassembly for function lines that cannot be hooked (off by default)

  Use caution when enabling this as it can increase startup time and memory usage.

Missing functions: If the wrong version of DbgHelp.dll is loaded certain functions will not be available.

- **Display DbgHelp missing function warning dialog** › show a warning when some functions are not available (on by default)

  Use caution when enabling this as it can increase startup time and memory usage.

 To see the order in which the *DbgHelp.dll* process checks directories to find symbols, see the diagnostic tab with the filter set to *DbgHelp debug*.

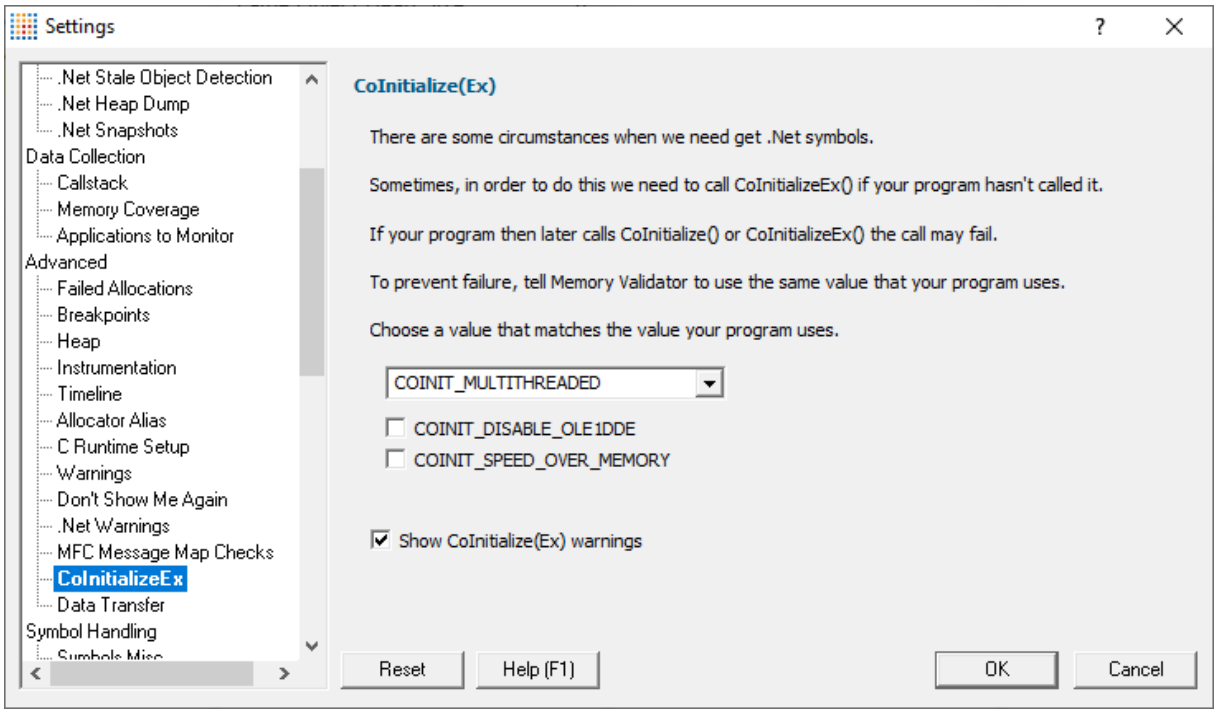
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.12 CoInitializeEx

The **CoInitializeEx** tab allows you to set the default behaviour used to initialize COM if Memory Validator needs to initialize COM to acquire symbols for .Net modules.

The default settings are shown below:



CoInitializeEx

In some situations the Validator needs to get .Net symbols and to do that COM needs to be initialized. This normally isn't a problem, but if your program also performs COM initialization and the sequence of events results in your COM initialization coming after the Validator's COM initialisation rather than getting the expected `ERROR_SUCCESS` return code you'll get either `ERROR_INVALID_FUNCTION` or `RPC_E_CHANGED_MODE`.

If you get `ERROR_INVALID_FUNCTION` this is OK, this just means you've called CoInitialize() or CoInitializeEx() multiple times with the same flags. Your code needs to handle `ERROR_INVALID_FUNCTION` as **not an error**.

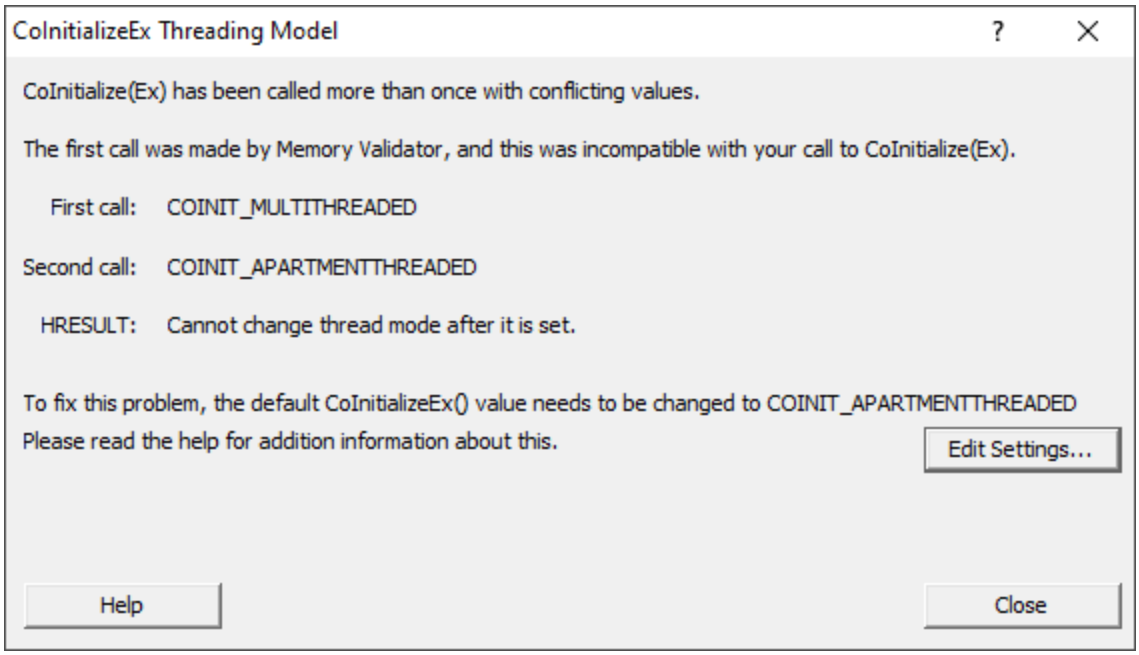
If you get `RPC_E_CHANGED_MODE` this means you need to change the Validator's default value to the same value your program is using. That's what this dialog allows you to do.

If you also wish to disable OLE DDE or favour speed rather than memory use we've provided appropriate options for you to select to add those flags to the threading mode.

See the Microsoft documentation for additional information on the behaviour of CoInitialize() and CoInitializeEx().

Runtime detection of CoInitializeEx conflict

When the above scenario happens, that the Validator has initialized COM before your code initializes COM and your call returns RPC_E_CHANGED_MODE, we display a dialog to warn you about this failure and provide you with the option of editing the default value for subsequent runs of your application.



- **Edit Settings...** > opens the CoInitializeEx dialog shown above

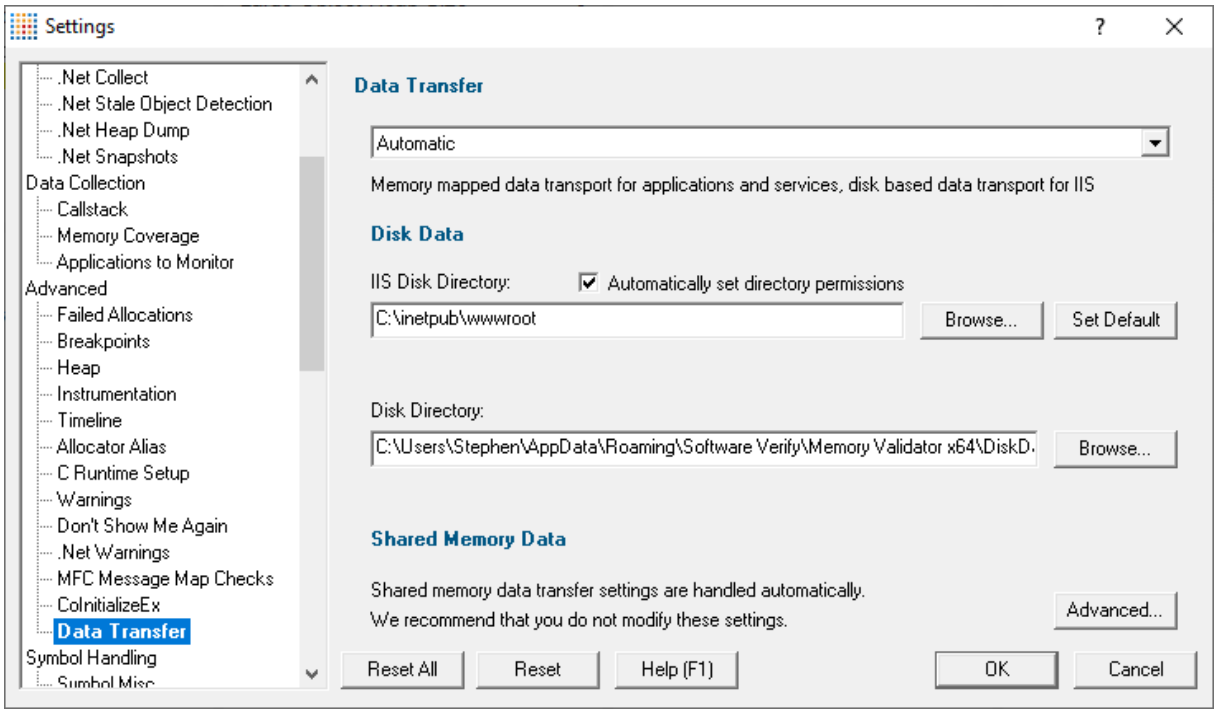
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.4.13 Data Transfer

The **Data Transfer** tab allows you to specify the overall behaviour of data transfer between your application and Memory Validator.

The default settings are shown below:



Data Transport

Choose the type of data transport you wish to use.

- **Automatic**. Applications and services use shared memory to transfer data. IIS uses disk based data transfer.
- **Disk**. Applications, services and IIS use disk based data transfer.
- **High Volume**. Data transfer has no data throttling applied to it. This mode is for use with applications that generate very high volumes of data rapidly. They typically exceed the buffering capabilities of Memory Validator when working with shared memory. The High Volume setting uses a data transport that doesn't have a data-throttling requirement allowing the high volume application to continue without waiting.

Automatic

Under most circumstances data transfer between Memory Validator and the target program (desktop, service, etc) is via shared memory. This is handled automatically.

Disk Data

Some applications and services don't allow shared memory access. For these occasions we use a file based data transfer, where the files are stored in a directory of your choice.

We provide two options for this, one for most applications and services (**Disk Directory**), and one for Internet Information Server (**IIS Disk Directory**), as this operates in a very restricted environment.

Both options are configured automatically, but you can override either by typing the path to a suitable directory or using the Microsoft directory browser.

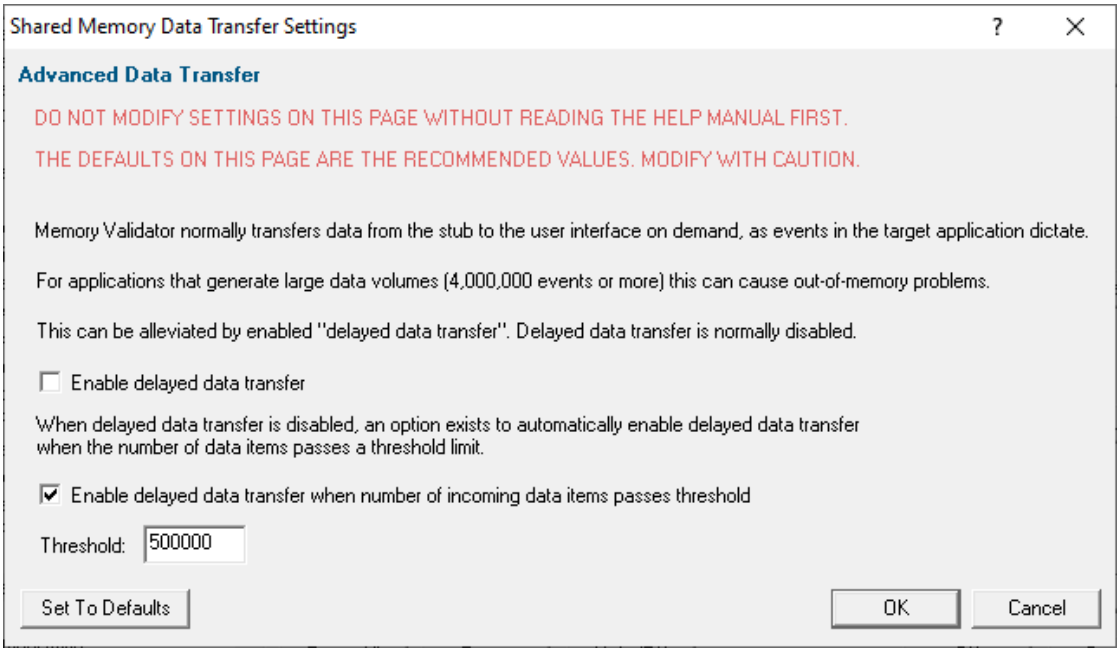
The ISS path you enter will be determined by the settings you have configured for IIS using the Internet Information Services Manager tool. We won't discuss that here because if you're using IIS we assume you already know how to configure IIS correctly.

Advanced

Shared memory data transfer can also be configured **but we strongly recommend that you leave these settings alone**.

The **Data Transfer Helper** is a separate application supplied in the installation directory.

- **Advanced...** › opens the data transfer settings dialog.



Here be dragons!

Caution: Modifying the settings on this page and using the data transfer helper application can prevent Memory Validator from working correctly.

- **Set To Defaults** › if you have modified the settings, this resets them

➜ See also the Reset to default buttons on the data transfer helper application below

If in doubt, don't modify these settings. If you promise to be careful, read on!

Delayed data transfer

Delayed data transfer is the process of throttling data rates in the stub so that the slower user interface can keep up with processing the data received.

In the stub, as an event occurs, data is queued and then sent to the user interface.

In the user interface, data from the stub is received and queued again for processing.

Any delay is usually in the slower user interface, but still not a problem for most applications.

However, some data intensive applications can generate so much data that the user interface gets swamped and can't process it all before running out of memory.

Temporarily limiting the data rate in the *stub* allows the user interface to stabilize the data processing.

Managing data rates

We recommend the default settings as shown above:

- disable delay data transfer for most applications
- enable *automatic* delay data transfer at a threshold of between 100,000 and 1,000,000 data items

If delayed data transfer is enabled all the time, the automatic options don't apply.

If you have more than 1GB RAM, you can raise these thresholds.

Data transfer helper application

A separate data transfer helper application is supplied in the installation directory.

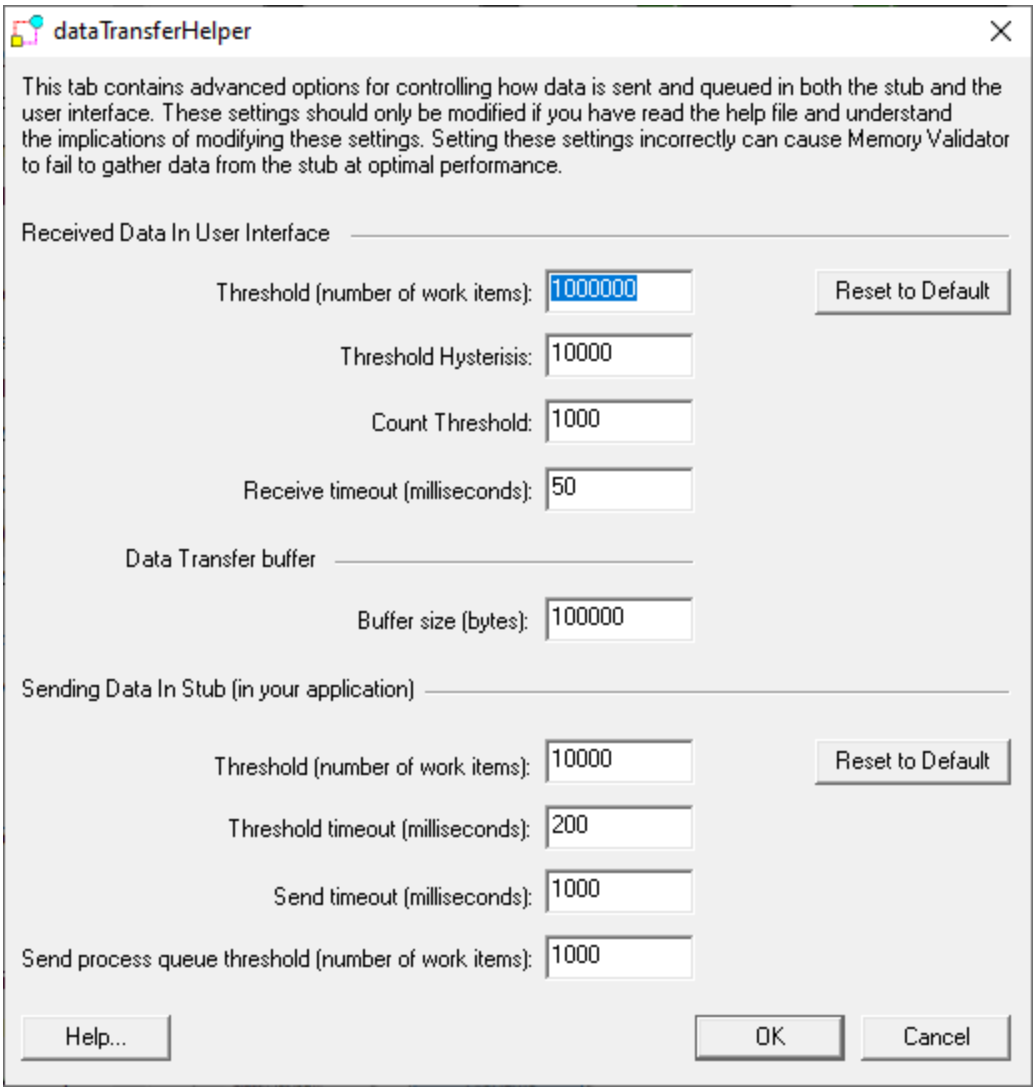
The helper application can be used to modify low level settings that apply when delay data transfer is activated as above.

 The helper should be used with care. We already warned of dragons above, but here we are, warning you again!

An HTML help page for this application is available by clicking the **Help** button on the helper application.

You can also find the help page directly as `dataTransferHelp.html`.

Please do take a moment to read the help before use.



Reset All - Resets **all** global settings, not just those on the current page.

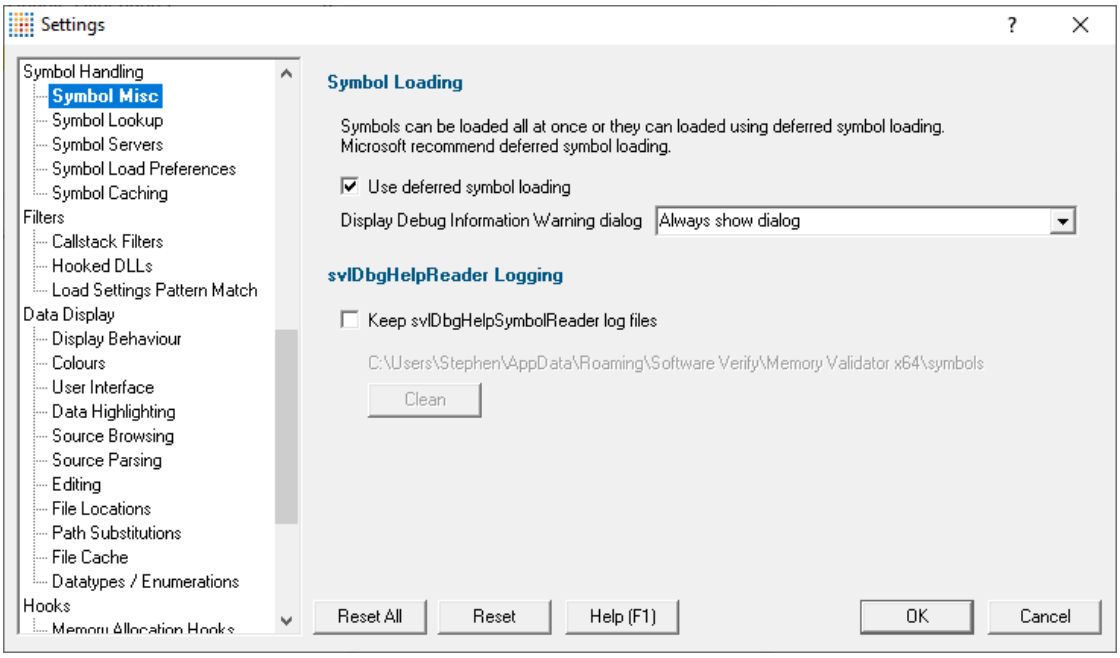
Reset - Resets the settings on the current page.

3.12.1.5 Symbol Handling

3.12.1.5.1 Symbols Misc

The **Symbols Misc** tab allows you to set miscellaneous symbols loading and resolving settings and some diagnostic options.

The default settings are shown below:



Immediate or deferred symbol loading

When converting program addresses to symbol names, you can choose immediate or defer loading until each symbol is needed.

- **Use deferred symbol loading** › uses deferred symbol loading rather than 'all at once' (on by default)

  Microsoft® recommend deferred symbol loading, claiming it is the fastest option. We give you the choice.

Symbol Reader Logging

Symbols are fetched from symbol servers using a helper process svlDbgHelpSymbolReader.exe. We log the command line and behaviour of this helper tool. This is displayed on the diagnostic tab.

If you wish the log files can be kept for later analysis. By default this option is turned off.

- **Keep svlDbgHelpSymbolReader log files** › keep the log files after Memory Validator has finished processing them

The path to the directory containing the log files is shown.

- **Clean** › delete all svlDbgHelpSymbolReader log files

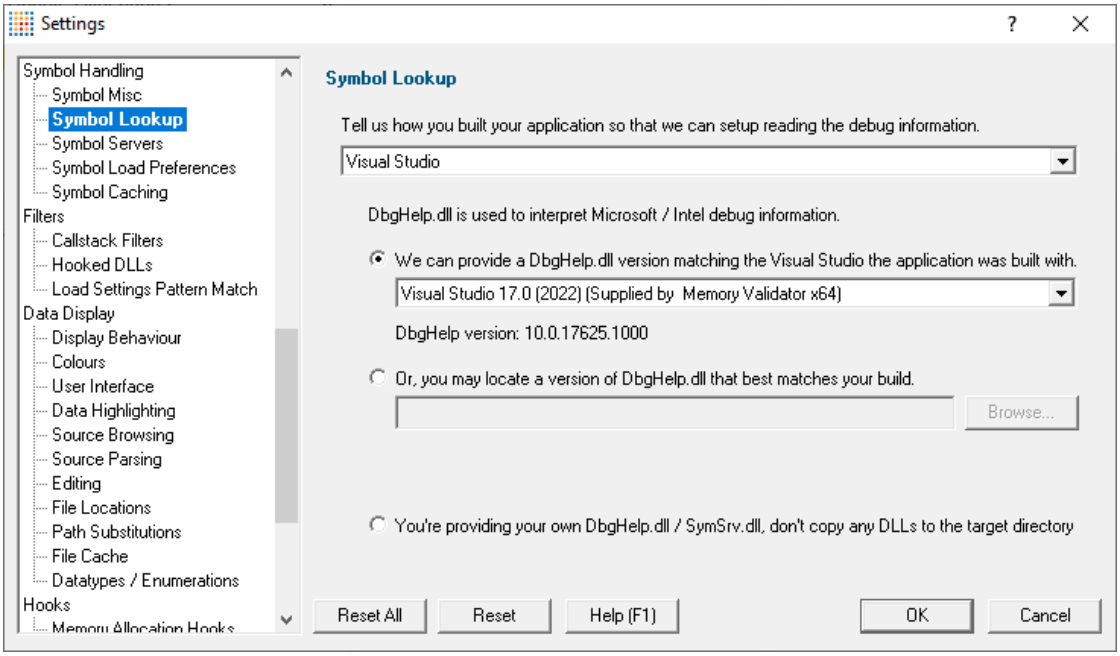
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.5.2 Symbol Lookup

The **Symbol Lookup** tab allows you to specify how and where symbolic information is retrieved for your application or service.

The default settings are shown below, although the Visual Studio version may vary.



Compiler / IDE Choice

Use the first combo box to choose which compiler / IDE you used to build your software.

Memory Validator will use the appropriate methods to read your symbols.

The choices are:

- Visual Studio
- Visual Basic 6
- Delphi or C++ Builder
- MingW
- Rust
- Dev C++
- Metrowerks CodeWarrior
- Salford Fortran 95
- Other

Symbol lookup for Microsoft / Intel compilers

- **We can provide a Dbghelp.dll** › choose one of Memory Validator's known good DbgHelp.dll's based on the version of Visual Studio you are using

  Memory Validator fetches symbols for your application using an appropriate symbol handler for the type of debugging information you have.

  For Microsoft Visual Studio users each version of Visual Studio provides different debugging formats which are readable by the appropriate DbgHelp.dll supplied by Visual Studio. A given version of DbgHelp.dll is usually able to read earlier formats of Microsoft debugging information but is not able to read a future format. For example Visual Studio 2005 (version 8) can read Visual Studio 6 debug information but cannot read Visual Studio 2008 debug information.

  Visual Studio 6.0 doesn't supply a DbgHelp.dll so we have provided one for use with Visual Studio 6.0.

  Visual Studio 10 is unusual in that the DbgHelp.dll (6.12) supplied by Visual Studio cannot read the debug information created by Visual Studio. To solve this problem we have supplied DbgHelp.dll (6.11) as an alternative.

  Memory Validator will choose the appropriate (most recent) version of Visual Studio automatically. You can override Memory Validator's choice by choosing the Visual Studio version from the **Visual Studio** combo box.

Specify your own DbgHelp.dll

- **Or, you may locate a version of DbgHelp.dll** › specify your own DbgHelp.dll to use with Memory Validator

  If you wish to explicitly specify which DbgHelp.dll to use choose the **Or, you may locate a version of DbgHelp.dll** option enter the path in the **DbgHelp.dll** edit field or use the **Browse...** button to select the dbgHelp.dll.

  Note that the directory that contains DbgHelp.dll ***should also contain symsrv.dll*** if you wish to use symbol servers with Memory Validator.

Don't update DbgHelp.dll

- **You're providing your own DbgHelp.dll** › use the DbgHelp.dll that ships with your application

  If your application needs to use a specific version of DbgHelp.dll that you're already providing with your application you should choose the **You're providing your own DbgHelp.dll** option to prevent Memory Validator from overwriting your DbgHelp.dll.

  Note that the directory that contains DbgHelp.dll ***should also contain symsrv.dll*** if you wish to use symbol servers with Memory Validator.

Visual Studio DbgHelp.dll version compatibility

For Microsoft Visual Studio users, each VS version provides different debugging formats which are readable by the appropriate DbgHelp.dll supplied with Visual Studio.

These handlers are usually backwards compatible, but not forwards compatible. For example Visual Studio 2005 (version 8) can read Visual Studio 6 debug information but cannot read Visual Studio 2008 debug information.

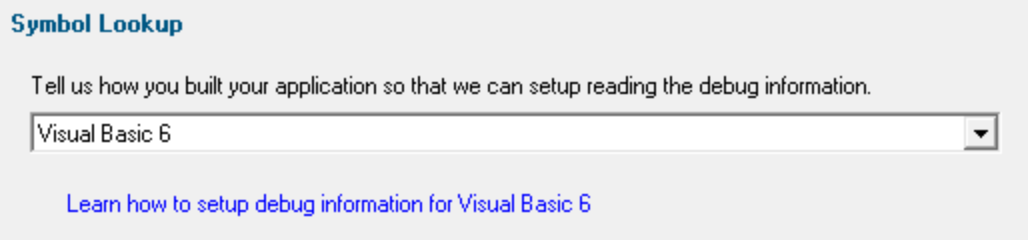
Visual Studio 6.0 doesn't supply a DbgHelp.dll so we have provided one for use with Visual Studio 6.0.

Visual Studio 10 is unusual in that the DbgHelp.dll (6.12) supplied by Visual Studio cannot read the debug information created by Visual Studio! To solve this problem we supply version 6.11 as an alternative.

To see the order in which the *DbgHelp.dll* process checks directories to find symbols, see the diagnostic tab with the filter set to *DbgHelp debug*.

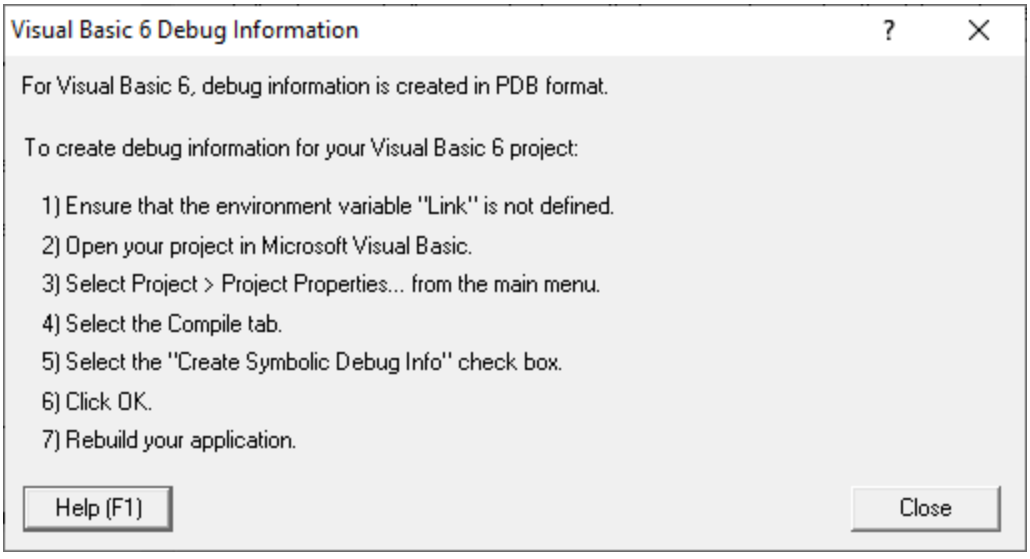
Symbol lookup for other compilers

If you are using another compiler click the link to see information about configuring debug information for that compiler.



After selecting the compiler, clicking the link will show a dialog box containing information relevant to the selected compiler.

For example:



Reset All - Resets **all** global settings, not just those on the current page.

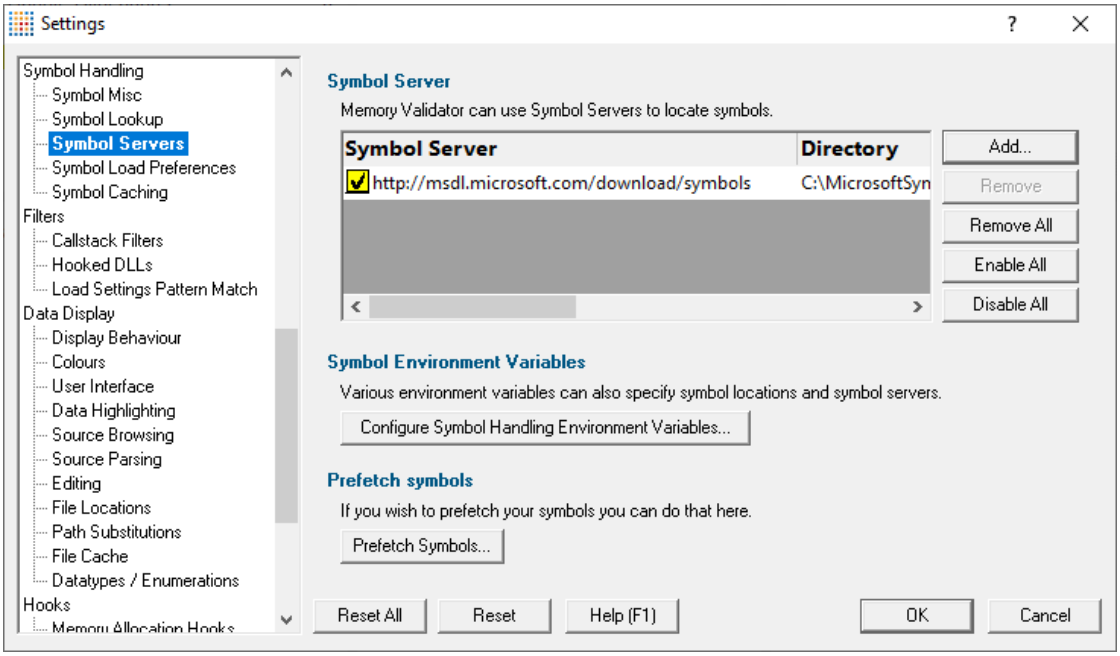
Reset - Resets the settings on the current page.

3.12.1.5.3 Symbol Servers

The **Symbol Servers** tab allows you to specify Symbol Servers to retrieve symbols used in your application.

You do not *need* to specify symbol servers if you do not wish to, and Memory Validator will work correctly without them.

Read on, or click on a setting in the picture below to find out more.



Symbol servers

Symbol servers are entirely optional, but are useful for obtaining symbols from a centralized company resource or for obtaining operating symbols from Microsoft.

The default symbol server is the Microsoft symbol server used for acquiring symbols about Microsoft's operating system DLLs. You may also wish to add some symbol servers for any software builds in your organisation.

A symbol server is defined by at least the following:

- the symbol server dll to be used to handle the symbol server interaction

- a directory location where symbol definitions are saved
- the server location - a url

Each symbol server can be enabled or disabled allowing you to keep multiple symbol server configurations available without constantly editing their definitions.

You can define up to four symbol servers and more than one can be enabled at a time.

Symbol Server Errors

Any symbol server entry shown in red indicates there is a problem with parts of the definition of that symbol server.

In the image shown above the symbol server at http://127.0.0.42:8000 cannot be reached. It is either offline or does not exist.

Managing symbol servers

- **Add...** › displays the symbol server dialog described below

- **Remove** › remove selected symbol server(s) in the list
- **Remove All** › remove all symbol servers

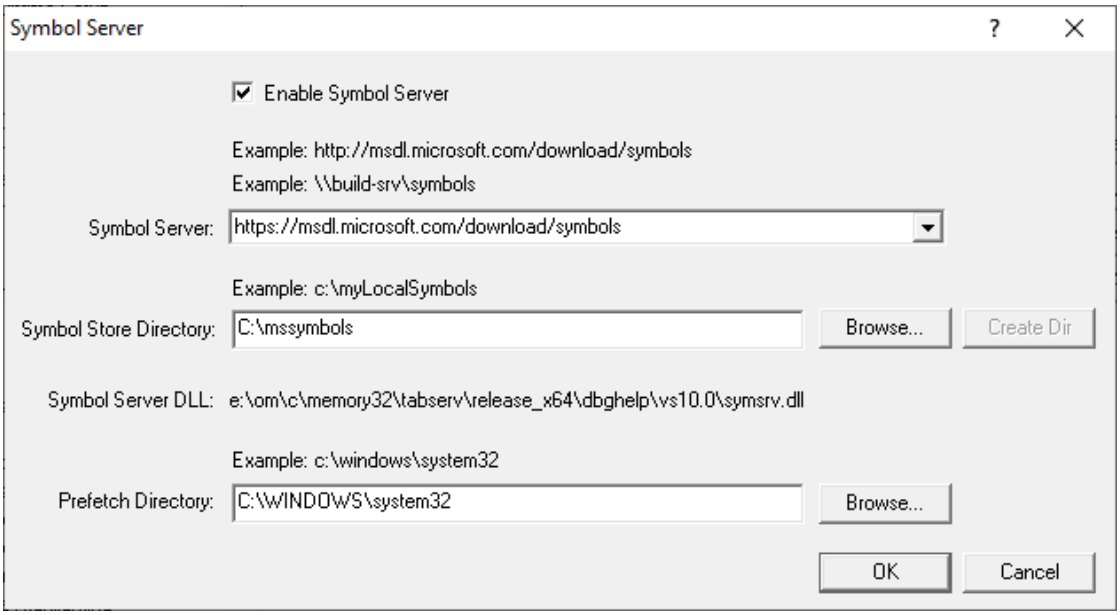
- **Enable All** › enables all symbol servers in the list
- **Disable All** › disables all symbol servers

> You can also enable or disable an item in the list via the yellow check box at the left of each row.

To edit the details for a symbol server, just double click the entry in the list to show the symbol server dialog again.

Symbol server dialog

The dialog initially appears pre-populated with some default values and allows you to set up or edit the definition of a symbol server. Some of the default values can be changed.



- **Enable Symbol Server** › enable or disable this server

The following three entries must be set to enable the **OK** button and define the symbol server.

> OK button not enabled?The OK button will only be enabled when the following entries have a valid value: - Symbol Server DLL names a dll present in the Memory Validator install directory. - Symbol Store Directory has been specified and exists. - Symbol Server URL has been specified (this value will not be checked for correctness).

- **Symbol Server** › select a predefined public symbol server or enter the URL of the symbol server you wish to use - the Microsoft server is initially set as the default
- **Symbol Store Directory** › enter or **Browse** to set the directory that will contain local copies of the downloaded symbols
    - **Create Dir** › creates a directory if you entered a directory name that does not exist yet

The **Symbol Server DLL** is set based on the Symbol Lookup settings you have chosen.

You can optionally associate a directory to scan when you are prefetching symbols (below)

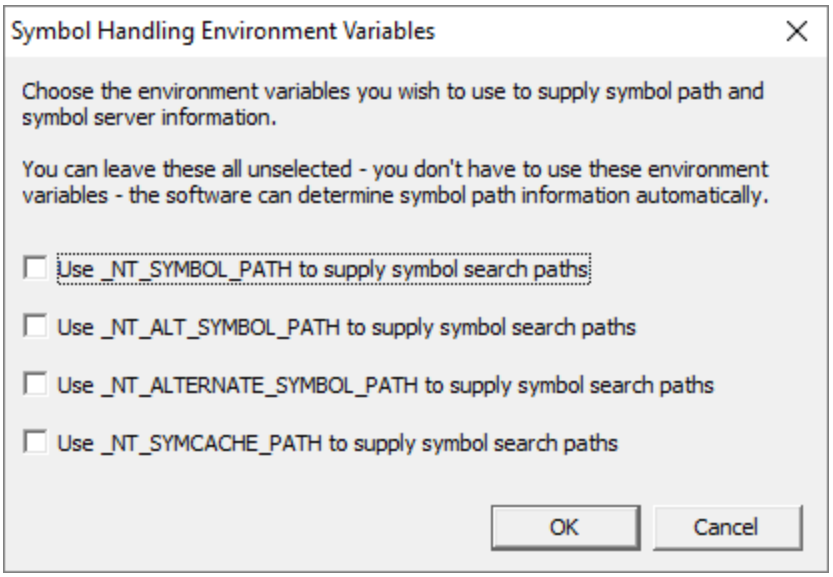
- **Prefetch Directory** › specify the directory to scan for symbols

Environment variables related to symbols

If you wish, you can set some environment variables to supply symbol paths.

- **Configure Symbol Handling Environment Variables** › opens the dialog below

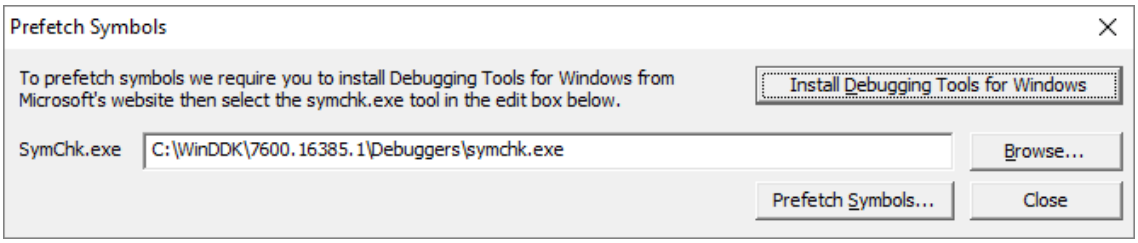
Check the desired options - if any.



Pre-fetching symbols

To avoid delays when using symbol servers, you can trigger the retrieval of symbols (by running SymChk.exe) to collect symbols for all executable files specified in the exe/dll which you associated with each symbol server.

- **Prefetch Symbols...** › open the Prefetch Symbols dialog below to continue



Prerequisites for pre-fetching symbols

The pre-fetching of symbols requires the installation of Microsoft's Debugging Tools.

You may already have Debugging Tools if you've previously installed the Windows Driver Kit (DDK or WDK) or the Windows SDK.

- **Install Debugging Tools for Windows** › opens a web page (as above) to download and install the x86 or x64 Debugging Tools for Windows

After installing the Debugging Tools, you must specify the location of SymChk.exe from the installed area.

- **SymChk.exe** › enter or **Browse** to SymChk.exe location

  A typical path might be `C:\WinDDK\7600.16385.1\Debuggers\symchk.exe`

Getting the symbols

Note that prefetching symbols may consume a large amount of disk space and download bandwidth.

You should ensure that you have at least 2 or 3Gb of disk free space, because of the total size of the download packages.

- **Prefetch Symbols...** › runs SymChk.exe to get all the symbols

  The symbols for each symbol server are stored in the associated symbol store directory.

If no symbol servers are specified in the symbol server settings above, you'll see a warning dialog and no symbols will be fetched.

Command line pre-fetching of symbols with the SymChk utility

The section on Pre-fetching symbols above is a convenient alternative to manually using the SymChk,exe utility.

To avoid delays when using symbol servers, you can pre-fetch symbols using the SymChk.exe command line tool that is part of Microsoft's Debugging Tools.

You may want to add the folder of the Debugging Tools for Windows package to the PATH environment variable on your system so that you can access this tool easily from any command prompt.

Example:

To use SymChk.exe to download symbol files for all of the components in the `c:\windows\System32` folder, you might use the command:

```
symchk.exe /r c:\windows\system32 /s SRV*c:\symbols\*http://msdl.microsoft.com/download/sym
```

where

`/r c:\windows\system32` finds all symbols for files in that folder *and any subfolders*

`/s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols` specifies the symbol path to use for symbol resolution.

In this case, `c:\symbols` is the local folder where the symbols will be copied from the symbol server.

➡ To obtain more information about the command-line options for SymChk.exe, type `symchk /?` at a command prompt.

Other options include the ability to specify the name or the process ID (PID) of an executable file that is running.

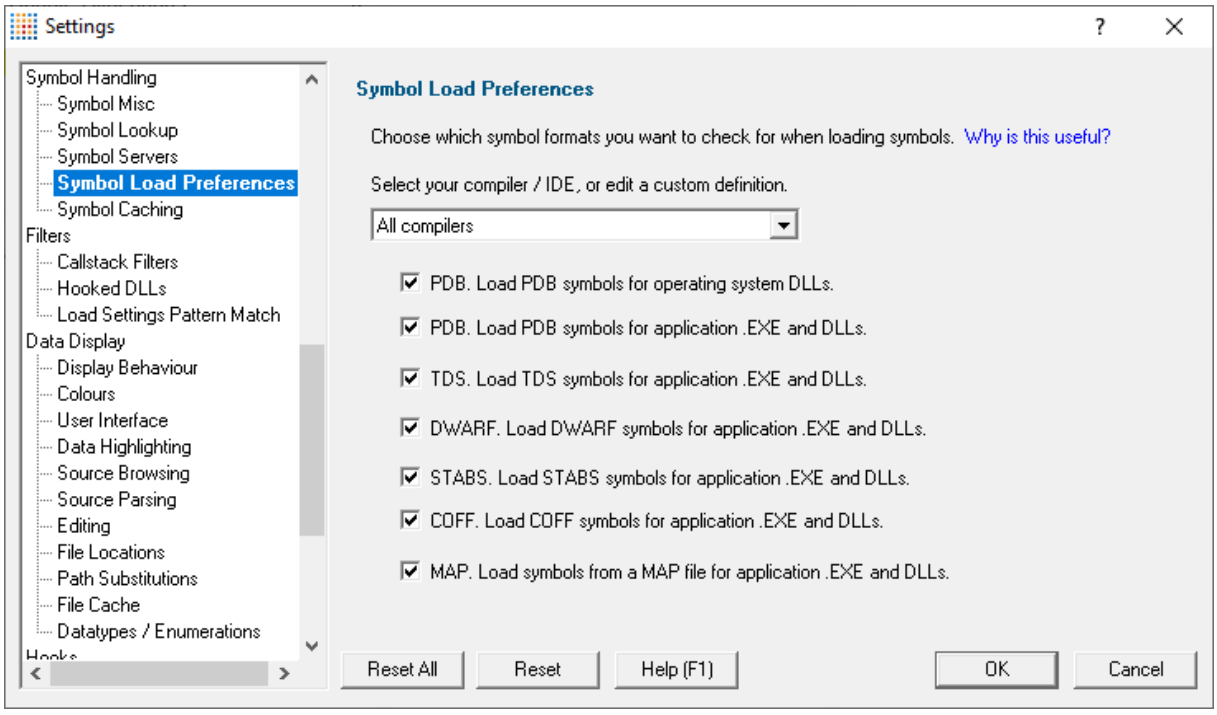
Reset All - Resets **all** global settings, not just those on the current page. This includes removing any symbol servers added.

Reset - Resets the settings on the current page. This includes removing any symbol servers added.

3.12.1.5.4 Symbol Load Preferences

The **Symbol Load Preferences** tab allows you to configure which debug information types are looked for and which are ignored.

This can save some time fetching symbols each time a DLL is loaded.



- **Select your compiler / IDE...** › choose a preset definition for a compiler / IDE, or edit one of four custom symbol load preferences

The present definitions are:
  - **I don't know which compilers** › choose this if you don't know which compilers were used to build the software
  - **All compilers** › choose this to let Memory Validator fetch the symbols
  - **Visual Studio** › choose this if you're only using Visual Studio
  - **Visual Basic 6** › choose this if you're only using Visual Basic 6

- **Delphi** › choose this if you're only using Delphi
- **C++ Builder** › choose this if you're using C++ Builder on 32 bit Windows
- **C++ Builder 32 bit** › choose this if you're using C++ Builder to build 32 bit applications
- **C++ Builder 64 bit** › choose this if you're using C++ Builder to build 64 bit applications
- **MingW / gcc / g++ / QtCreator / Dev C++** › choose this if you're using MingW / gcc / g++
- **QtCreator / Dev C++** › choose this if you're using QtCreator
- **Dev C++** › choose this if you're using Dev C++
- **Rust** › choose this if you're using Rust
- **Salford Fortran 95** › choose this if you're using Salford Fortran 95
- **Custom 1** › choose this to edit a definition you can reuse
- **Custom 2** › choose this to edit a definition you can reuse
- **Custom 3** › choose this to edit a definition you can reuse
- **Custom 4** › choose this to edit a definition you can reuse

Editing a definition

Once a definition has been selected the appropriate check boxes next to each debug information type are populated.

You can edit these selections, for example to include or exclude PDB debug information for operating system DLLs, or allow Memory Validator to search for COFF debug information, whatever is optimal for the way you are working.

Custom definitions

Only the custom definitions will be remembered if they are edited.

The four custom definitions will be remembered, so the next time you choose them you'll get the definition you edited. If you choose one of the preset definitions and edit it, you'll use the edited definition, but if you then change to a different preset (or a custom definition) and then back to the original preset you'll get the preset definition, not your edited version of the preset definition.

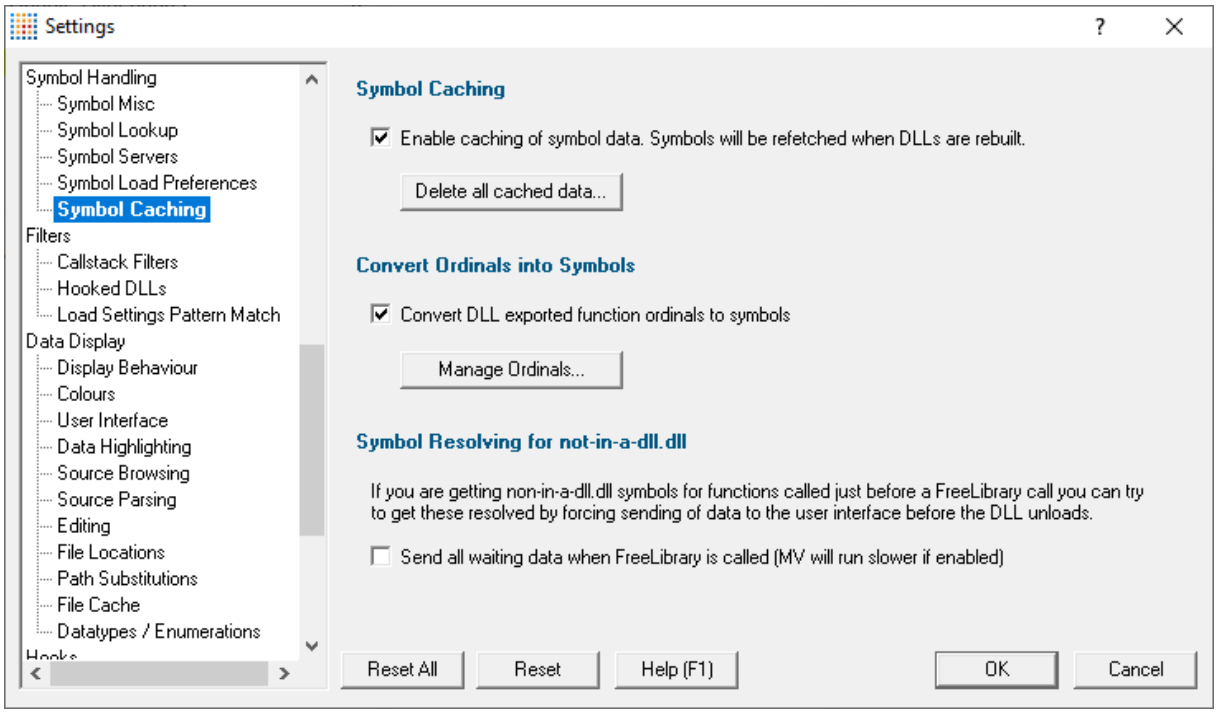
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.5.5 Symbol Caching

The **Symbols Caching** tab allows you to set settings for caching, ordinal handling and resolving symbols that are "not-in-a-dll.dll".

The default settings are shown below:



Symbol Caching

Retrieved symbols for each DLL are normally cached, only being refetched if the DLL is rebuilt or the version of DbgHelp.dll changes.

- **Enable caching of symbol data** › enable symbol caching (the default)

  Caching the symbols is slightly faster as calls to DbgHelp.dll can be omitted and symbol information accessed directly.

- **Delete all cached data** › removes all cached data member information from disk

Convert ordinals into symbols

- **Convert DLL exported function ordinals to symbols** › enable the ordinal to function name mapping

  You'll need to tick this to enable the use of mapped names defined in the list. If you don't, you won't see the names being used.

- **Manage Ordinals...** › shows the Ordinal Handling dialog to manage which .def files are associated with which DLLs

Premature module unloading and not-in-a-dll.dll

In some situations a module will be unloaded by the target program before any addresses for the DLL have been resolved into symbols, causing the symbols to be shown as **not-in-a-dll.dll**.

This can be prevented by ensuring all data is processed before a module is unloaded:

- **Send all waiting data when FreeLibrary is called** › avoid scenarios where symbols get displayed as not-in-a-dll.dll (off by default)

  There is a performance hit with this option as the target program will be waiting until all symbols have been sent.

Reset All - Resets **all** global settings, not just those on the current page.

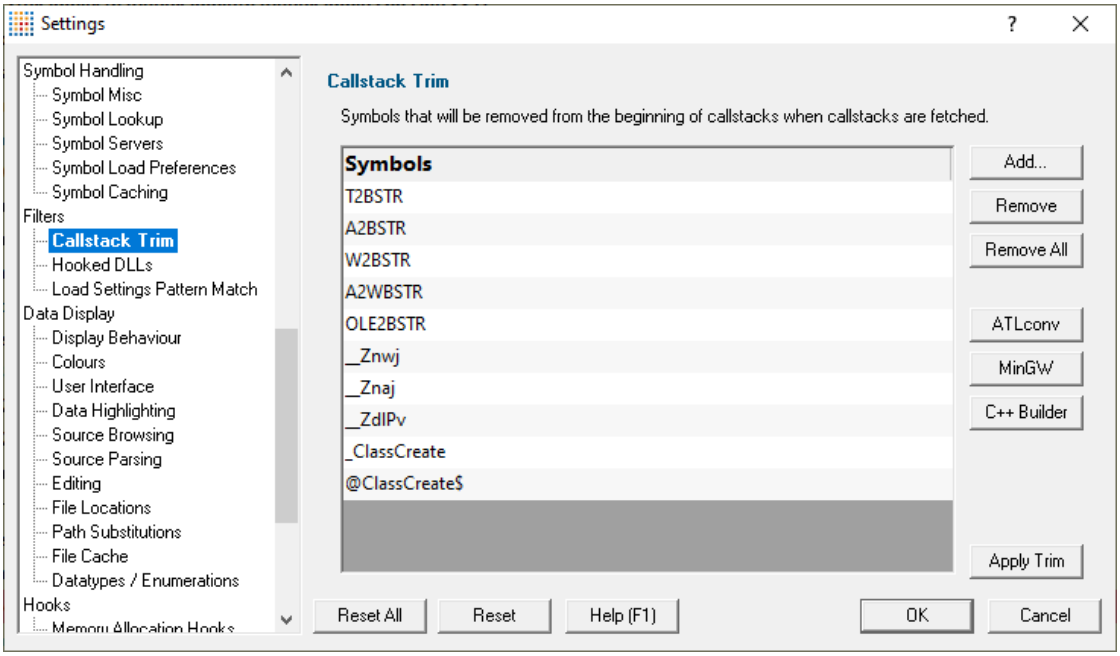
Reset - Resets the settings on the current page.

3.12.1.6 Filters

3.12.1.6.1 Callstack Trim

The **Callstack Trim** tab allows you to specify symbols that should not appear in callstacks collected by Memory Validator.

The default options are shown below:



Callstack trim filters

Callstack trim filters can help simplify the display of callstacks, making it easier to see *effective* rather than *actual* allocation locations.

There are two principal use case for this feature:

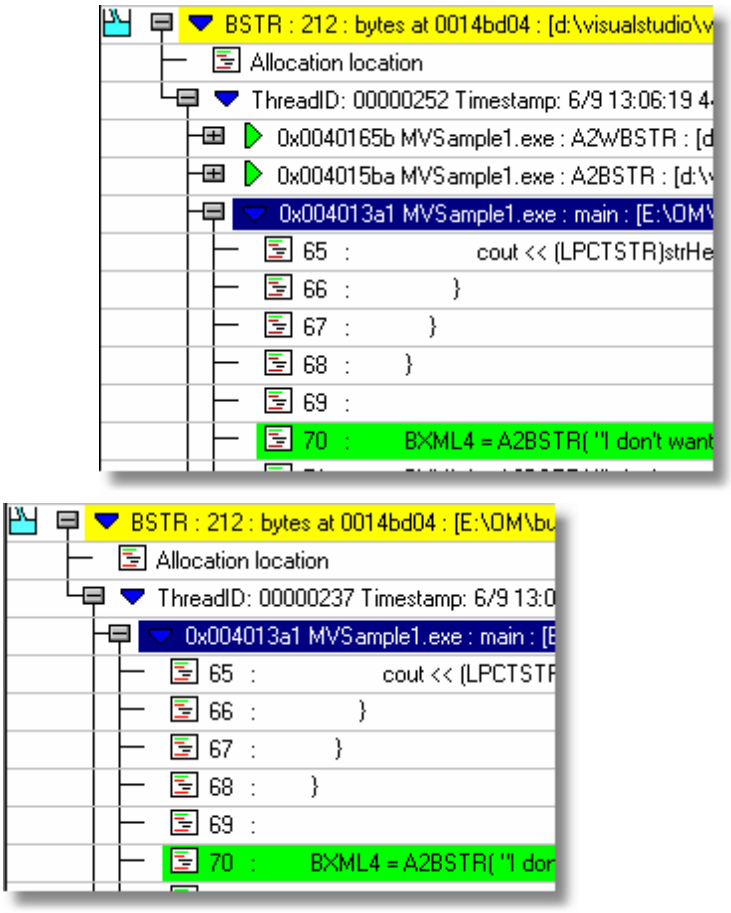
1. when an inline function definition allocates memory which is then used by your application.

2. when some functions wrap a function that is allocating memory but you don't want the wrapper functions appearing in your callstack. An example of this might be an in-house memory allocator that wraps the malloc/realloc/free functions.

By removing the symbol from the callstack, the memory appears to be allocated at the point of *calling* the inlined function, not inside it.

By way of example, the two images below show a callstack for allocating a `BSTR` using an inline function `A2BSTR` which calls another inlined function `A2WBSTR`, defined in the ATL header file atlconv.h.

The second version has had the inline functions (`A2BSTR` and `A2WBSTR`) filtered out so as not to appear in the callstack.



Using callstack trim filters

Add a symbol to the list of symbols to be filtered out of the beginning of callstacks:

- **Add** › An entry is added to the list › **enter the name** of the symbol › press **return** or click anywhere outside of the field to confirm (or press escape to cancel)
- **Remove** › removes any select entries in the list
- **Remove All** › removes all symbol names, clearing the list

To edit an existing symbol, double click the list entry.

Batch adding of callstack trim filters

For a couple of specific cases, you can add multiple callstack filters at once, although note that these are already in the list by default.

- **ATLconv** › add the default inline symbols used in the atlconv.h header file (`T2BSTR`, `A2BSTR` etc)
- **MinGW** › add the default memory allocation and deallocation symbols used in the MinGW compiler (`__Znwj`, `__Znaj`, and `__ZdlPv`)
- **C++ Builder** › add the symbols used in the C++ Builder compiler (`_ClassCreate`, and `@ClassCreate$`)

When will the callstack trim filters take effect?

The callstack trim filters will be used the next time you monitor an application using Memory Validator.

Trim existing callstacks

If you need to trim the callstacks in the current session you can do that using the Apply Trim option.

- **Apply Trim** › apply the callstack trim filters to callstacks that have already been collected. This will cause any displays with callstacks to refresh.

Reset All - Resets **all** global settings, not just those on the current page.

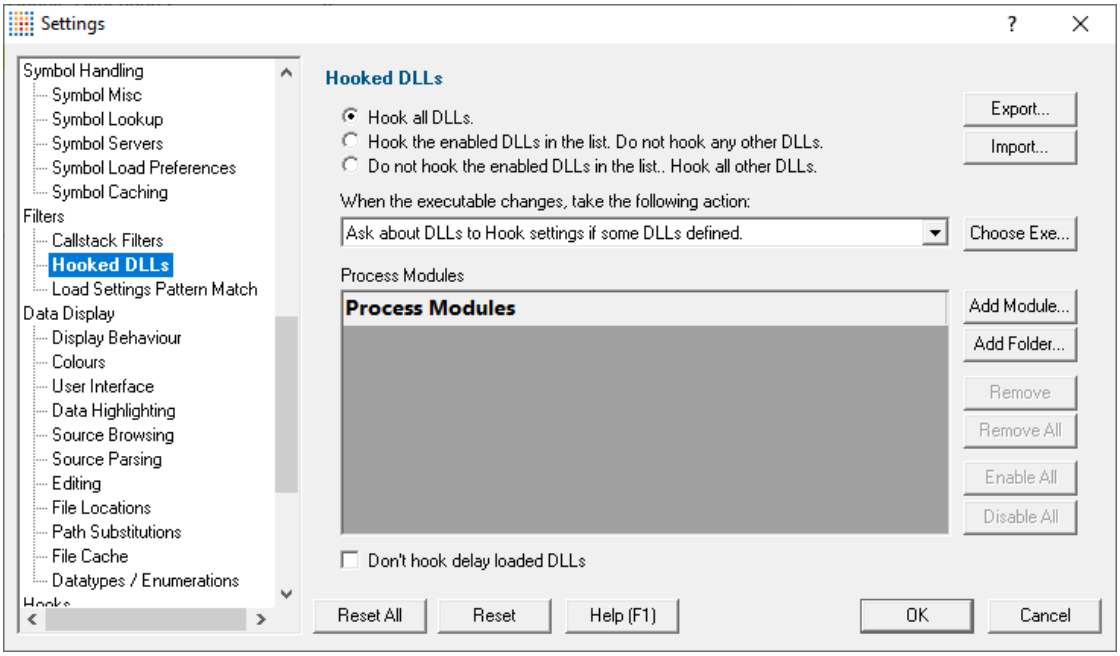
Reset - Resets the settings on the current page.

3.12.1.6.2 Hooked DLLs

The **Hooked DLLs** tab allows you to specify hooks for functions that Memory Validator does not initially know about.

The default settings are shown below:

Read on, or click on a setting in the picture below to find out more.



Which DLLs to hook - the hooking rule

By default, Memory Validator will try to hook all DLLs and .EXEs used by your application, but you can choose to list only those which should be included or excluded

- **Hook all DLLs** › hook everything - ignoring the settings in the list
- **Hook the enabled DLLs in the list** › hook only the ticked modules listed
- **Do not hook the enabled DLLs in the list** › *ignore* all the ticked modules in the list, and hook everything else

Populating the process modules list

The process modules list should specify the following items to be included or excluded from hooking in the target application

- DLLs
- .EXEs
- folders *containing* DLLs and .EXEs

Initially the list is empty as the default option is to hook all DLLs and ignore the list. You can add modules to the list by:

- automatically adding modules on which your application is dependent
- manually adding modules or folders

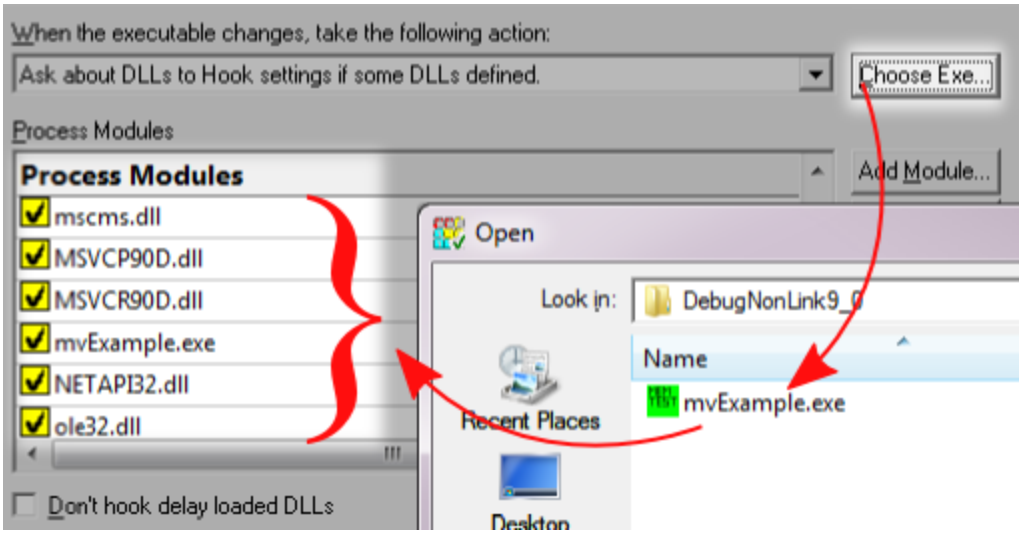
- editing modules or folders already in the list

modules whose inclusion is controlled by this list will override the hook insertion settings on the Data Collection > Collect tab.

Automatic module addition

You can automatically populate the list with all the dependent modules for your application:

- **Choose Exe...** > navigate to your application and click **Open** > all the process modules appear in the list



Manual module addition

You can also manually add one or more modules or a folder to the list.

- **Add Module** > navigate to the DLL or EXE and click **Open** > all the selected items are added
- **Add Folder** > navigate to the folder and click **OK** > the folder is added to the list

Manual addition might be useful for example if you use LoadLibrary() to load a DLL rather than linking it, as this would not be picked up automatically by the *Choose Exe...* method.

By default, all the modules are ticked in the yellow checkboxes.

Note that ticked modules or folders are either **in**cluded or **ex**cluded depending on the hooking rule above

Altering existing module names

Although you can't add blank entries to the list and edit them, you *can* edit existing items in the list by double clicking on an entry:

- enter only the module name, not the path
- you can use wildcards like MFC*.dll, but only for DLLs, not folders

Managing the process modules list

The usual controls apply for removing or changing the enabled state of items in the list:

- **Remove** › removes selected items in the list
- **Remove All** › removes all items, clearing the list
- **Enable All** › ticks all items in the list for applying to the hooking rule
- **Disable All** › unticks all items in the list, meaning they won't apply to the hooking rule

Alternatively, press Del to delete selected items, and Ctrl + A to select all items in the list first.

Exporting and importing

Since the list of hooked DLLs (and the rule being applied) can be quite complicated to set up and optimise, you can export the settings to a file and import them again later. This is useful when switching between different target applications.

- **Export...** › **choose or enter** a filename › **Save** › outputs the hooking rule and the list of modules to the file
- **Import...** › **navigate** to an existing *.mvx file › **Open** › loads the hooking rule and the list of modules

The exported file can be used with the -dllHookFile command line option.

Optionally hooking delay loaded DLLs

- **Don't hook delay loaded DLLs** › prevents hooking of delay loaded DLLs. The default is to hook these.

  What is 'delay loading'?

  Delay loading a DLL is when it it is implicitly linked, but not actually loaded until your code references a symbol contained in the DLL.

  Delay loading can speed up startup time, but unhandled exceptions may cause your program to terminate if the DLL can't be found when needed during the run time.

Launching new Applications

When specifying DLLs to hook, and launching different applications, it can be quite easy to forget to change the hooked DLLs for the new program. This might be the case when performing unit tests, for example.

Using the wrong list of hooked DLLs for a program will likely cause incorrect coverage results, so you can opt to be warned about the DLLs being hooked whenever the target application changes between sessions (using the dialog below).

The choices in the drop down list are only applicable when the application changes:

- **Ask about DLLs to Hook settings if some DLLs defined**

  You'll only be asked about the settings if you defined some DLLs in the list *and* if the hooking rule is not set to *hook all DLLs*

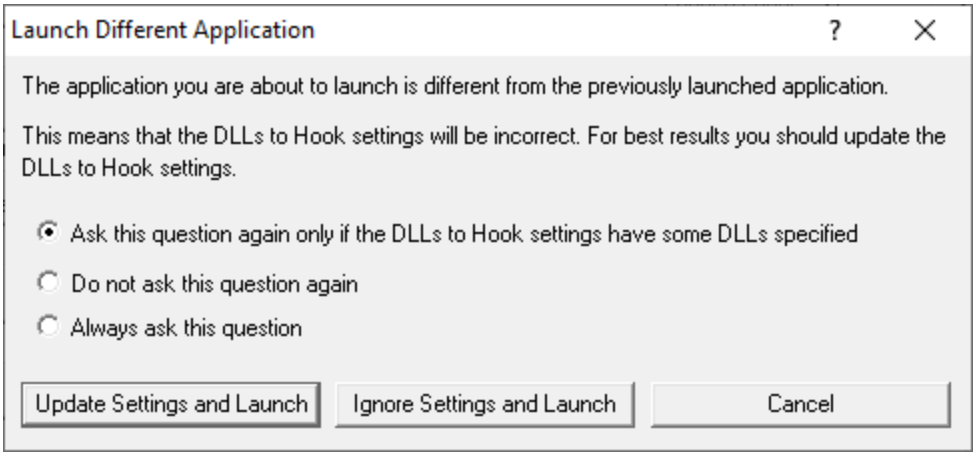
- **Always ask about DLLs to Hook settings**

  You'll always be asked about the settings - *whatever* the other settings are.

- **Never ask about DLLs to Hook settings**

The 'Launch Different Application' dialog

When being asked about the hooked DLL settings, you'll see the following dialog:



You can update the settings; ignore them and launch anyway, or just cancel the launch:

- **Update Settings and Launch** › **edit** the settings › click **OK** › the application will be launched
- **Ignore Settings and Launch** › the application will be launched without updating the settings
- **Cancel** › won't launch the application

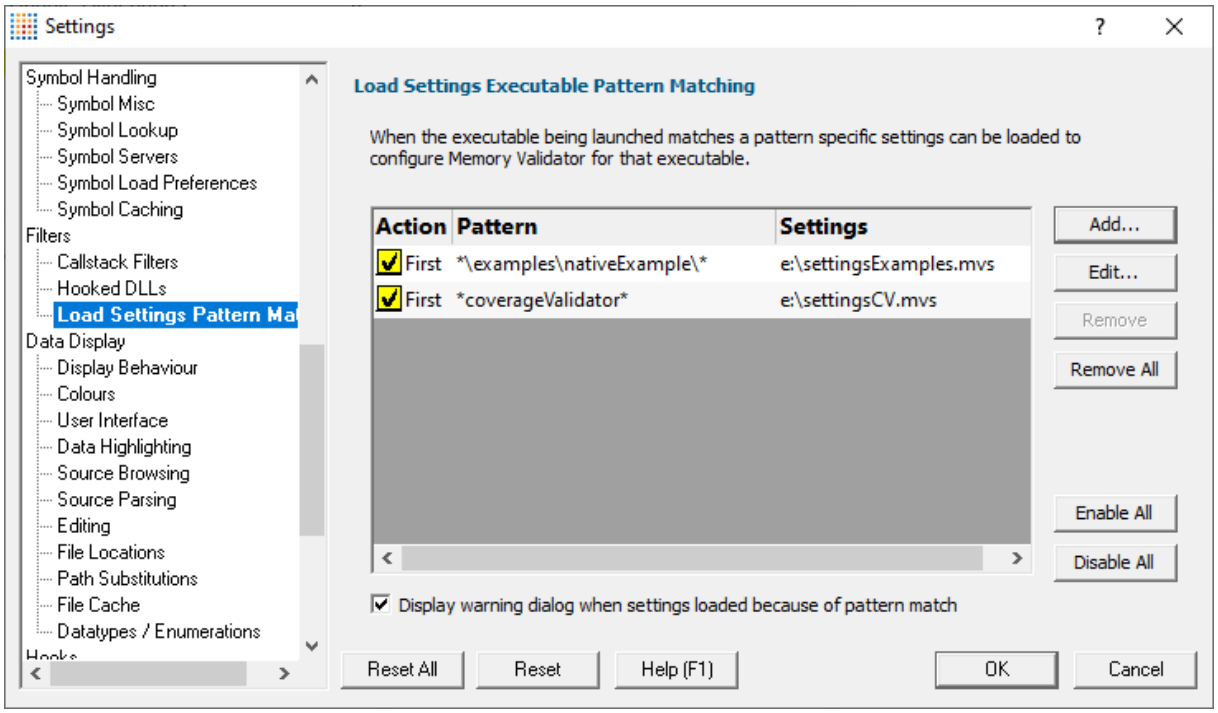
To change when you are asked this question, just choose the appropriate option in the dialog.

Reset All - Resets **all** global settings, not just those on the current page. This includes removing any process modules added here.

Reset - Resets the settings on the current page. This includes removing any process modules added here.

3.12.1.6.3 Load Settings Pattern Match

The **Load Settings Pattern Match** tab allows you to configure loading of different settings depending on the executable being launched (or relaunched).



The grid shows one pattern match per line.

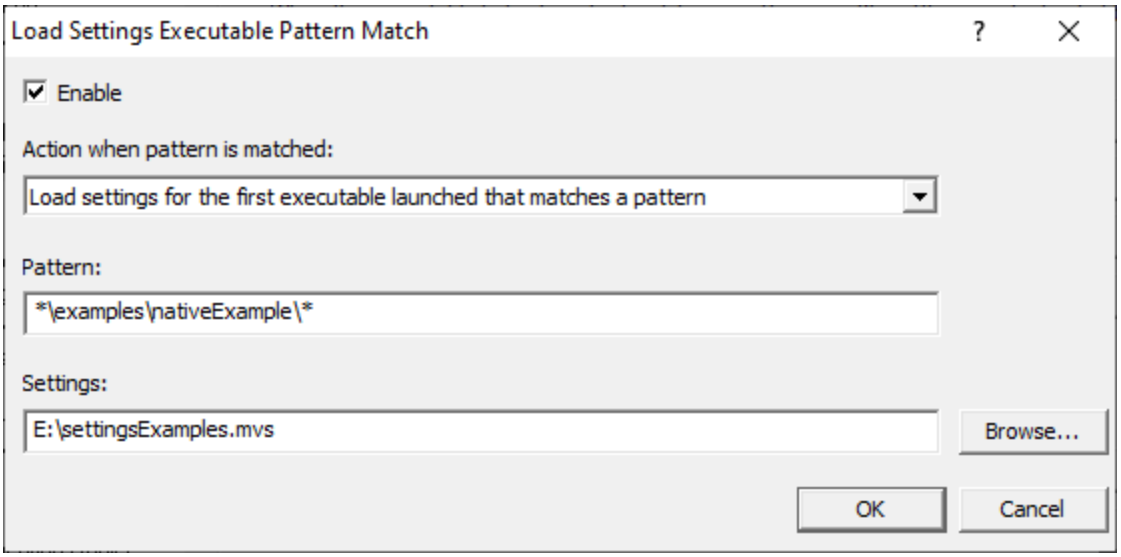
The buttons alongside allow you to Add, Edit and Remove patterns that you have created. You can also enable and disable them all.

- **Add...** › display the pattern match dialog to create a pattern to match.
- **Edit...** › display the pattern match dialog to edit the selected pattern.
- **Remove...** › delete the selected pattern.
- **Remove All...** › delete all selected patterns.
- **Enable All...** › enable all patterns.

- **Disable All...** › disable all patterns.

Pattern Match Dialog

The pattern match dialog allows you to create and edit pattern matches.



- **Enable** › enable or disable this pattern.
- **Action** › how to evaluate if this pattern is matched.
  - **Load settings for first executable...** › the first executable that matches a pattern causes the settings to be loaded.
  - **Load settings for each executable...** › each executable that matches a pattern causes the settings to be loaded provided it is not the same executable that previously loaded the settings.
  - **Load settings for every executable...** › every executable that matches a pattern causes the settings to be loaded.

  When a different pattern matches the first/each status is reset.
- **Pattern** › a text pattern, including the * wildcard, to match an executable path.

  Examples:
  ```
  *\examples\nativeExample\*
  c:\tests\*
  e:\dev\myProject\release\*.exe
  ```
- **Settings** › the full path to the settings you want to load if the action and pattern match an executable.

How does the pattern matching work?

It is probably easiest to demonstrate how pattern matching works with some examples.

Let's assume we have two patterns:

`*\examples\nativeExample\*` that will load `e:\settingsExamples.mvs`
`*coverageValidator*` that will load `e:\settingsCV.mvs`.

We'll cover each of the possible action criteria for a sequence of application launches, showing which settings are loaded and why.

- **Load settings for first executable...** › the first executable that matches a pattern causes the settings to be loaded.

Launched application	Settings loaded	Reason
e:\examples\nativeExample\release\nativeExample.exe	e:\settingsExamples.mvs	1st application, new pattern
e:\examples\nativeExample\release\nativeExample.exe		repeat application, same pattern
e:\examples\nativeExample\debug\nativeExample.exe		2nd application, same pattern
c:\program files (x86)\software verify\Memory Validator x86\coverageValidator.exe	e:\settingsCV.mvs	1st application, new pattern
e:\examples\nativeExample\release\nativeExample.exe	e:\settingsExamples.mvs	1st application, new pattern

- **Load settings for each executable...** › each executable that matches a pattern causes the settings to be loaded provided it is not the same executable that previously loaded the settings.

Launched application	Settings loaded	Reason
e:\examples\nativeExample\release\nativeExample.exe	e:\settingsExamples.mvs	new application, new pattern
e:\examples\nativeExample\release\nativeExample.exe		repeat application, same pattern
e:\examples\nativeExample\debug\nativeExample.exe	e:\settingsExamples.mvs	new application, same pattern
c:\program files (x86)\software verify\Memory Validator x86\coverageValidator.exe	e:\settingsCV.mvs	new application, new pattern
e:\examples\nativeExample\release\nativeExample.exe	e:\settingsExamples.mvs	new application, new pattern

- **Load settings for every executable...** › every executable that matches a pattern causes the settings to be loaded.

Launched application	Settings loaded	Reason
e:\examples\nativeExample\release\nativeExample.exe	e:\settingsExamples.mvs	Every application
e:\examples\nativeExample\release\nativeExample.exe	e:\settingsExamples.	Every application

e:\examples\nativeExample\debug\nativeExample.exe	mvs e: \settingsExamples. mvs	Every application
c:\program files (x86)\software verify\Memory Validator x86\coverageValidator.exe	e:\settingsCV.mvs	Every application
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples. mvs	Every application

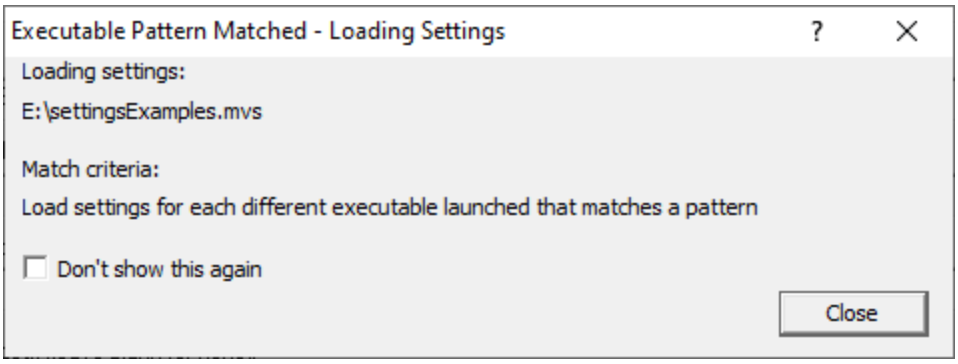
Warning

When a pattern is matched and the action criteria are satisfied the specified settings will be loaded.

A warning can be displayed at this point to remind you that the settings are being changed.

- **Display warning dialog...** > the warning dialog will be displayed when the pattern match criteria are met.

The warning dialog looks like this:



Reset All - Resets **all** global settings, not just those on the current page.

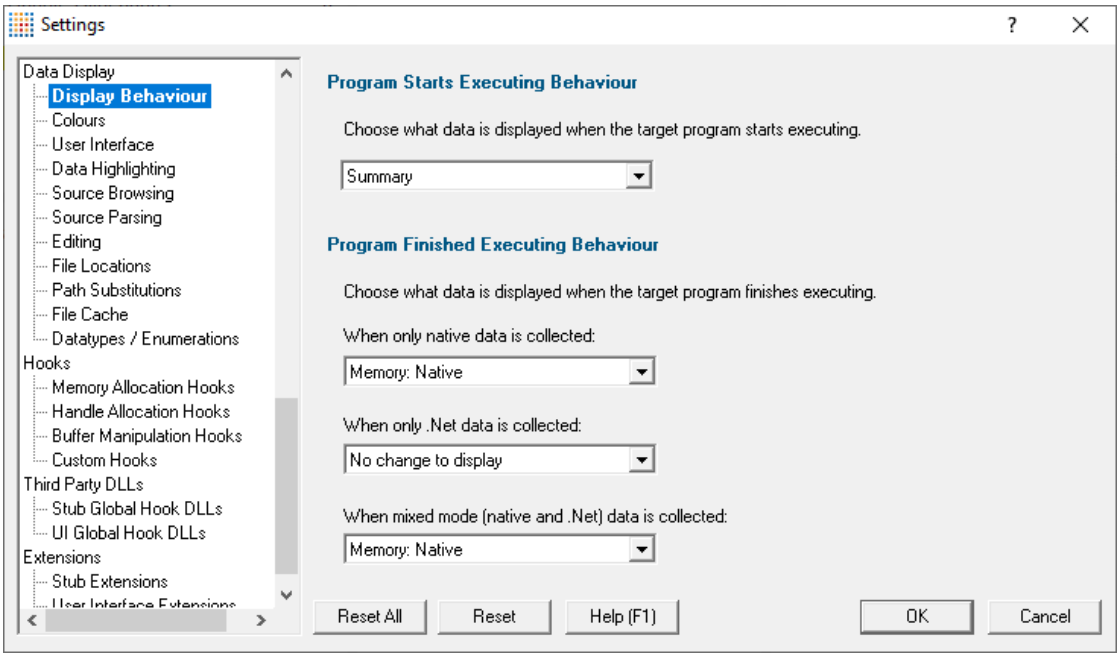
Reset - Resets the settings on the current page.

3.12.1.7 Data Display

3.12.1.7.1 Display Behaviour

The **Display Behaviour** tab allows you control how which displays are shown when a program starts executing and when a program finishes executing.

The default options are shown below:



Memory Validator can change the current display to any of the following displays.

- **Summary** › the main display
- **Memory : Native** › display native memory allocations
- **Memory : .Net** › display .Net memory allocations
- **Timeline** › timeline of all memory and handle allocations
- **Statistics : Types** › statistics about allocation types
- **Statistics : Sizes** › statistics about allocation sizes
- **Statistics : Locations** › statistics about allocation locations
- **Statistics : Generations** › statistics about allocation generations
- **Statistics : Ages** › statistics about allocation ages
- **.Net : Snapshots** › .Net snapshots
- **.Net : Heap Dumps** › .Net heap dumps
- **.Net : Leak Analysis** › .Net leak analysis
- **Analysis : Hotspots** › allocation hotspots
- **Analysis : Coverage** › allocation coverage
- **Analysis : Query** › allocation query
- **Analysis : Pages** › memory layout information
- **Analysis : Virtual** › virtual memory data and visualisation
- **Diagnostic : Diagnostic** › diagnostic information
- **Diagnostic : Stdout** › text collected from stdout
- **Diagnostic : Environment Variables** › environment variables from the program under test
- **Diagnostic : Child Processes** › processes launched by the program under test

Program Starts Executing Behaviour

When Memory Validator starts monitoring the behaviour of an application the current display can automatically be switched to any of the displays listed above.

There is also the option not to change the display.

Program Finished Executing Behaviour

When Memory Validator has finished processing all the information from the target application the current display can automatically be switched to any of the displays listed above.

There is also the option not to change the display.

The type of display that may interesting for collected data depends on the type of program that was executed. Native, .Net or Mixed mode. To accommodate this we provide one setting for each of the three program types.

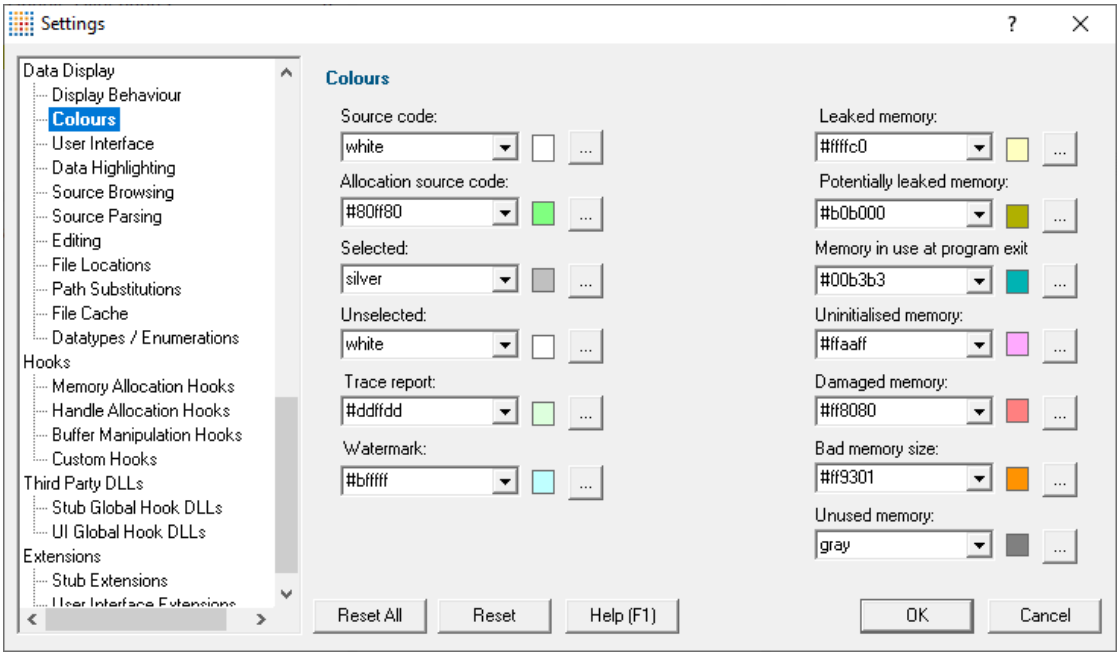
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.2 Colours

The **Colours** tab lets you choose the colours used to display each type of data item collected by Memory Validator.

The default colours are shown below:

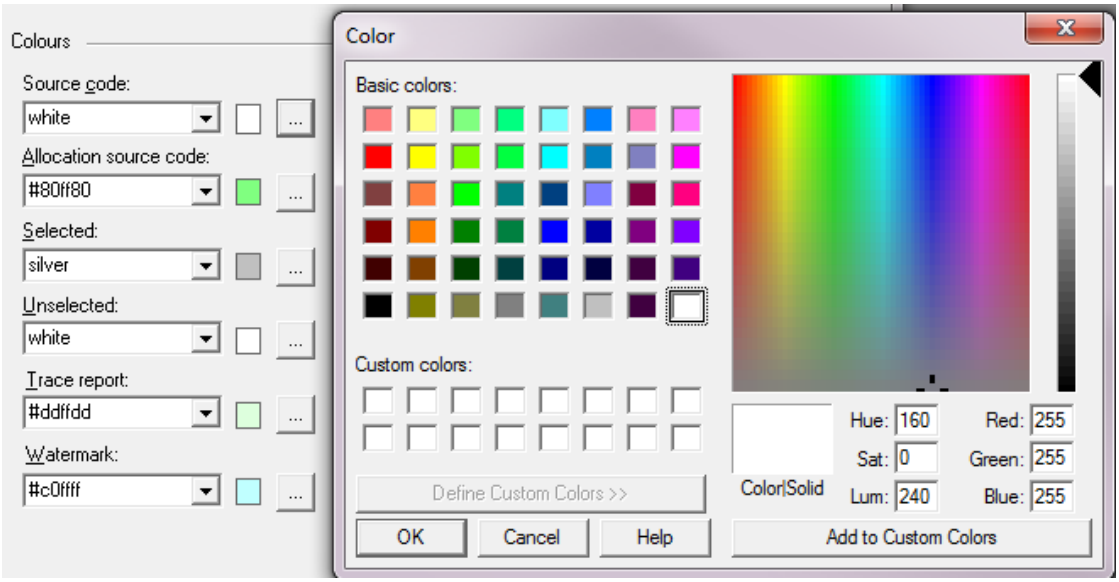


Changing display colours

For each colour you can choose a predefined colour or make your own:

- Use the drop-down list  › pick one of 16 predefined colours below

- Click the  button › edit the colour using the standard colour dialog:



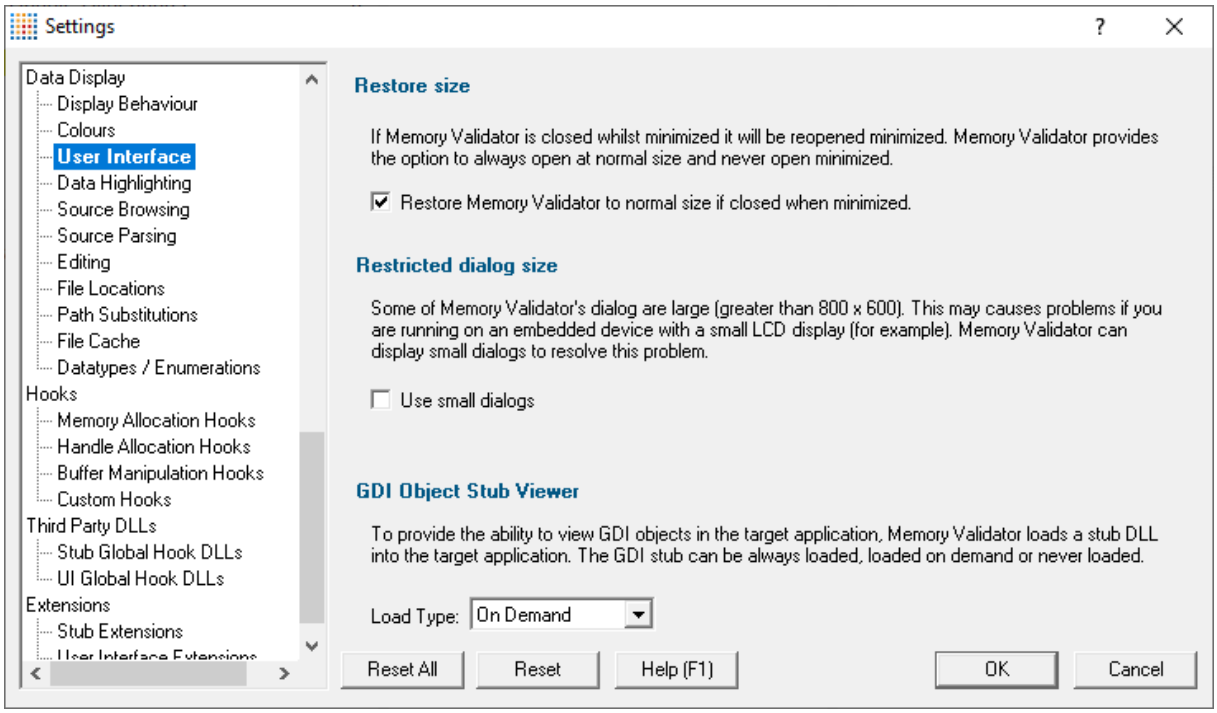
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.3 User Interface

The **User Interface** tab allows you to change some user interface characteristics including those that may help with using Memory Validator on devices with a smaller display.

The default settings are shown below:



Restoring window size when starting up

If Memory Validator is closed while minimized to the taskbar, the default behaviour when restarting is to restore the main window to normal size.

You can choose to keep it minimized on restart instead:

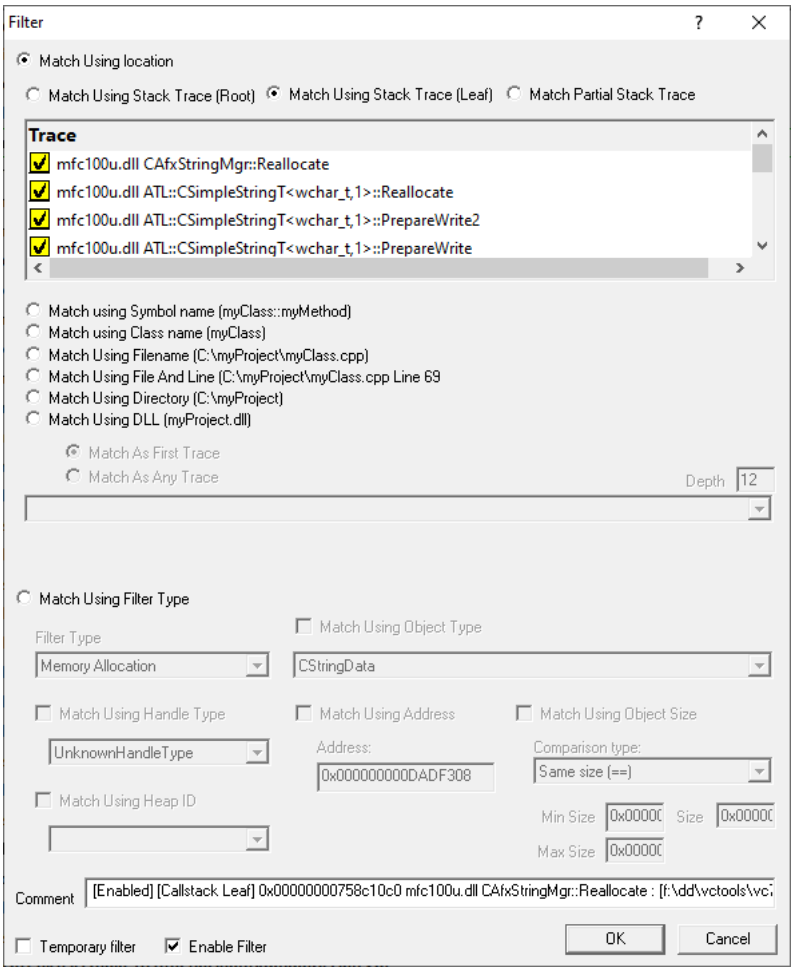
- **Restore Memory Validator to normal size if closed when minimized** › unticking will restart in the same state as it was closed

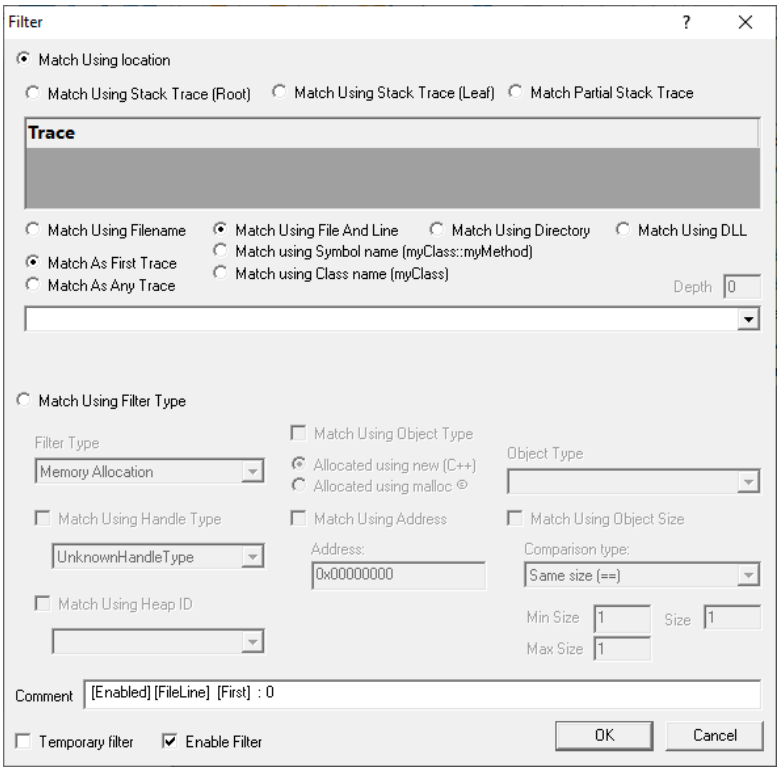
Restrict dialog size

When using small display devices, for example an 800x600 screen, some of the larger dialogs will not fit on the screen, but you can restrict dialog sizes in such environments:

- **Use small dialogs** › restrict dialog sizes

For example, the images below show the relative sizes of the Filter dialog where the height reduces from 730 to 592 pixels when using small dialogs:



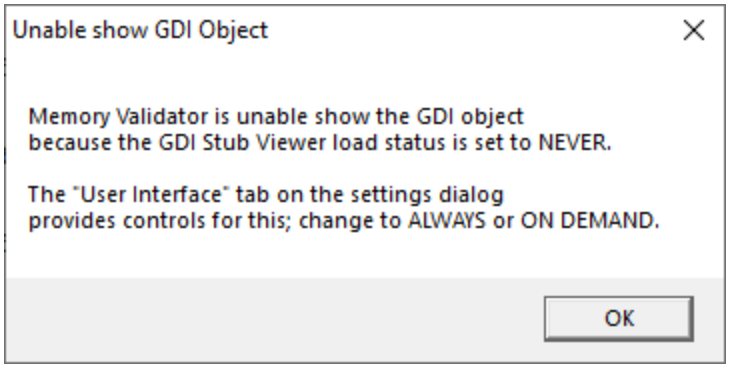


GDI object stub viewer

From the menus on the Memory or Types tab, you can view a GDI resource in a dialog. This is enabled by loading a GDI Viewer into the stub in the target application.

This GDI stub viewer can be loaded at startup; when needed, or not at all:

- **Load Type** › choose when the GDI Viewer is loaded, the options are:
  - **Always** › the viewer is loaded when first attaching to the target application
  - **On Demand** › the viewer is loaded when the first request to view a GDI object is made, if at all
  - **Never** › the GDI stub is never loaded and requests to view GDI objects will be blocked using the following message



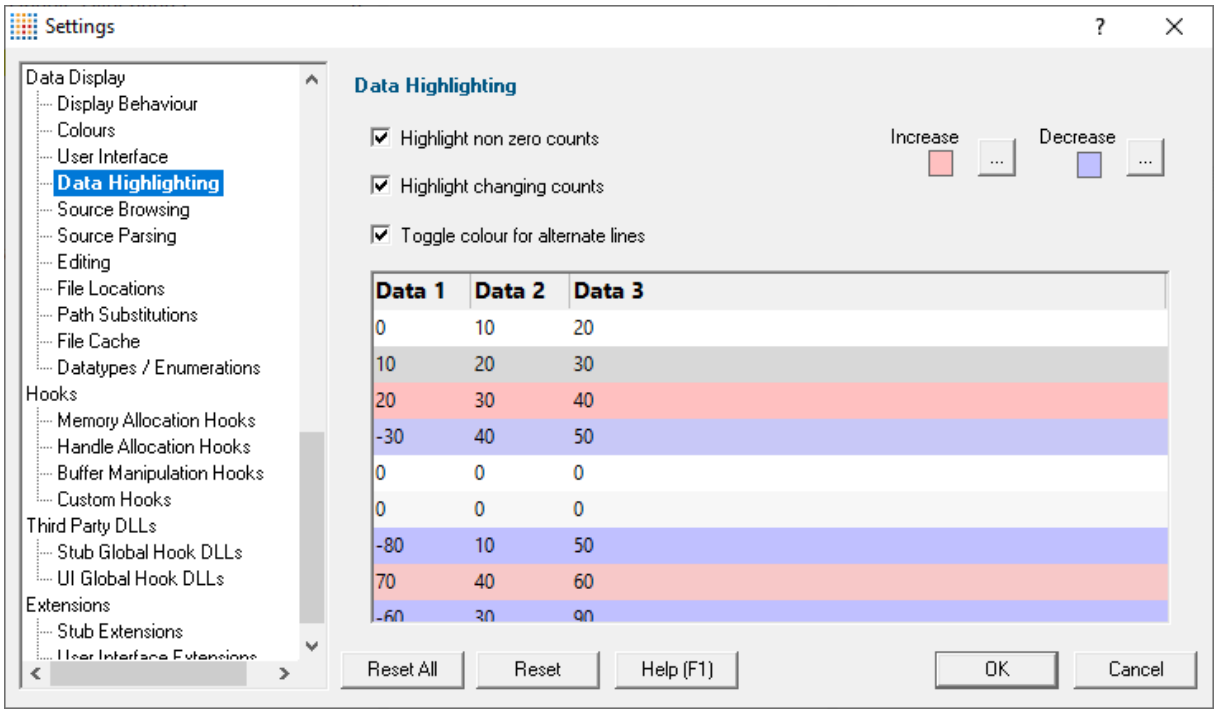
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

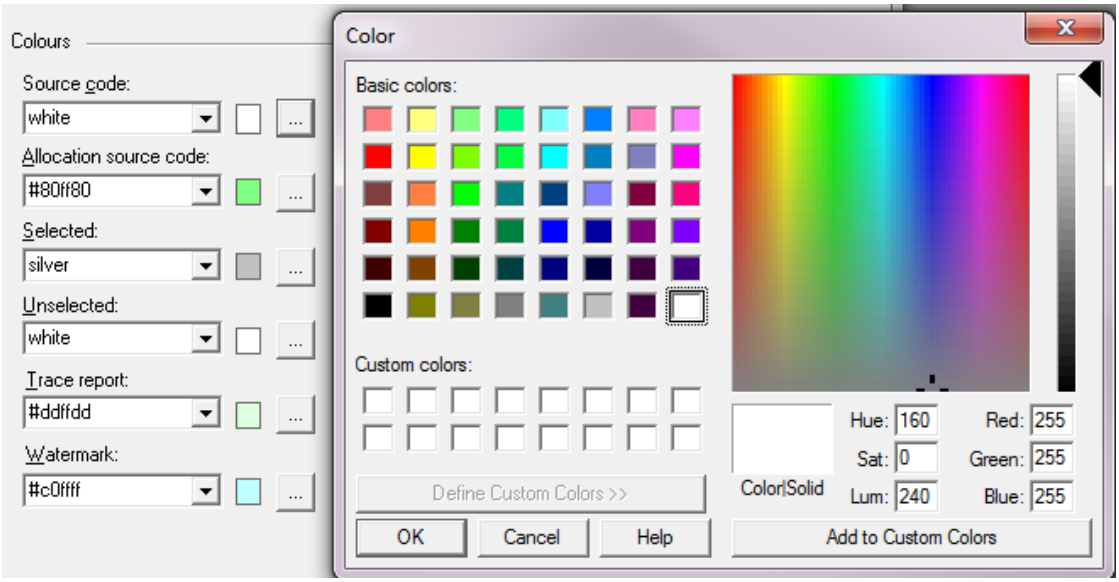
3.12.1.7.4 Data Highlighting

The **Data Highlighting** tab allows you to change how data is displayed and highlighted on the various grid and tree displays.

The default settings are shown below:



- Click the ... button › edit the increase and decrease colours using the standard colour dialog:



These colours are used to colour the background of highlighted lines.

- **Highlight non zero counts** › lines with values > 0 are highlighted with a light grey background
- **Highlight changing counts** › lines with increasing values are highlighted with the increasing colour. Lines with increasing values are highlighted with the decreasing colour

- **Toggle colour for alternate lines** › alternate lines have the background colour subtly changed so that longer lines are easier to read

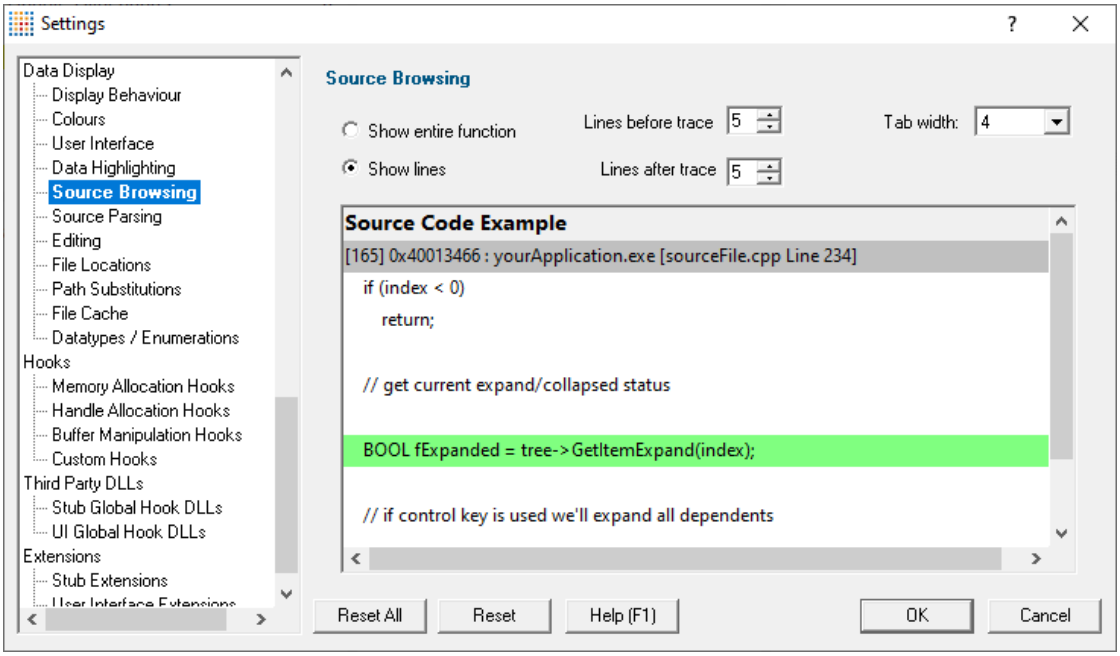
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.5 Source Browsing

The **Source Browsing** tab allows you control how much source code is displayed and some basic formatting.

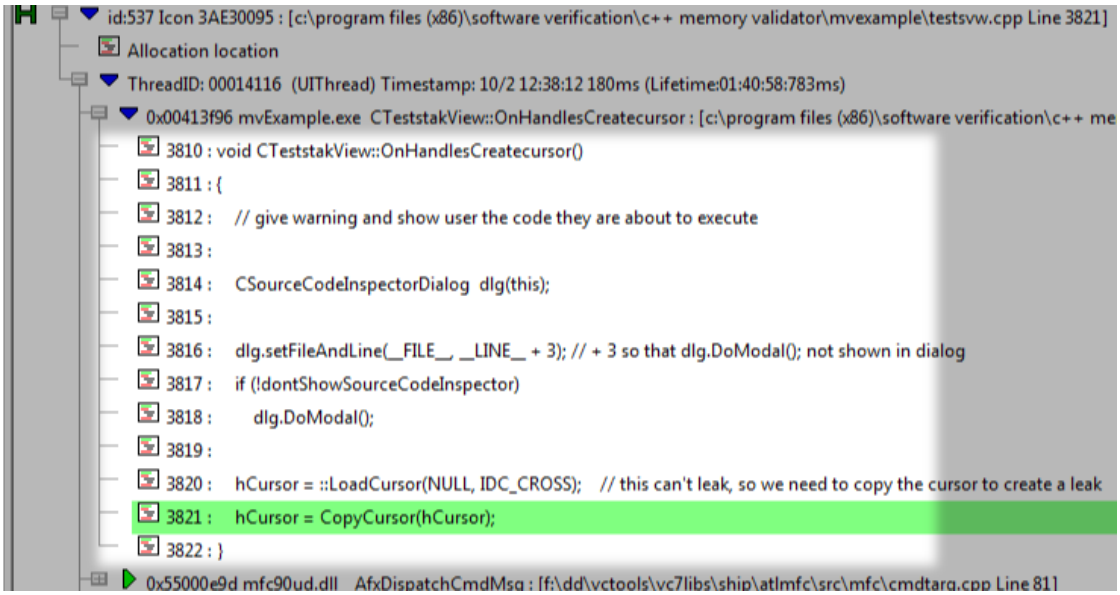
The default options are shown below:



Source browsing

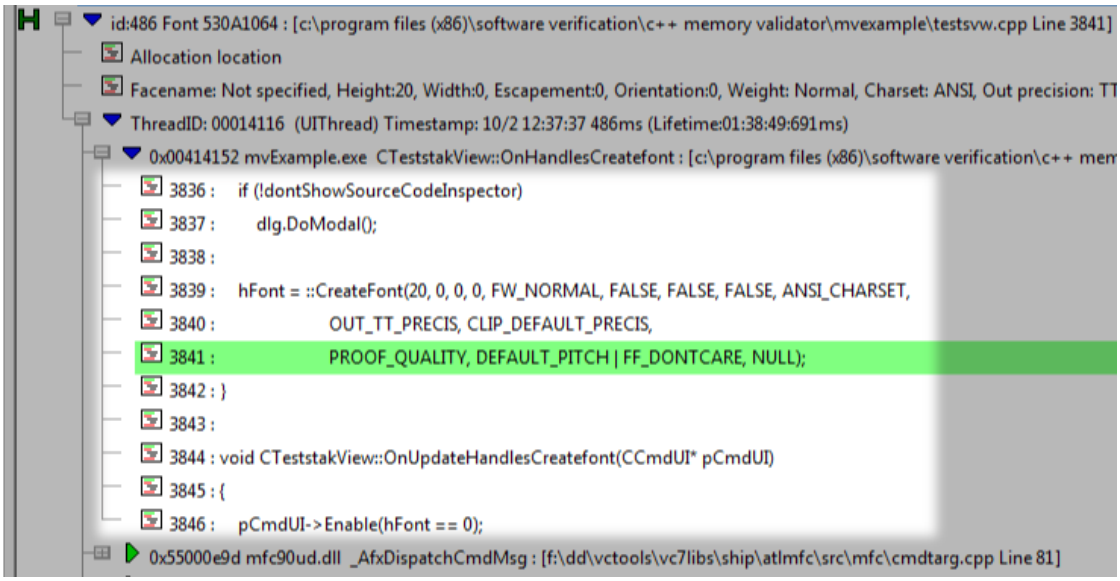
When expanding a callstack to view the source code, you can choose to see the whole function or a few lines either side of the line of interest.

- **Show entire function** › shows the whole function source as shown here:



- **Show lines** › shows a given number of lines before and after the point of interest:
  - **Lines before trace** › number of lines before, from 0 to 100
  - **Lines after trace** › number of lines after, from 0 to 100

  The default is to show 5 lines above and below, as shown here



Changing these settings will affect any new callstacks which are expanded, but will not reformat any that have already been opened.

Source browsing - how much to show?

Showing the entire function is more likely to show the full context of the allocation or deallocation line, but if you have particularly long functions it may become cumbersome to browse the callstack data!

Because of the unpredictable lengths of showing entire functions, this is not the default setting.

Showing a set number of lines reduces the amount of source display to something that is consistent and manageable, but you may see parts of neighbouring functions that are not relevant (as above), and you may not see enough of the preceding lines to determine the allocation context.

Tab size formatting

When formatting the source code being displayed you can control the tab size

- **Tab width** › set the tab size between 1 and 16 characters

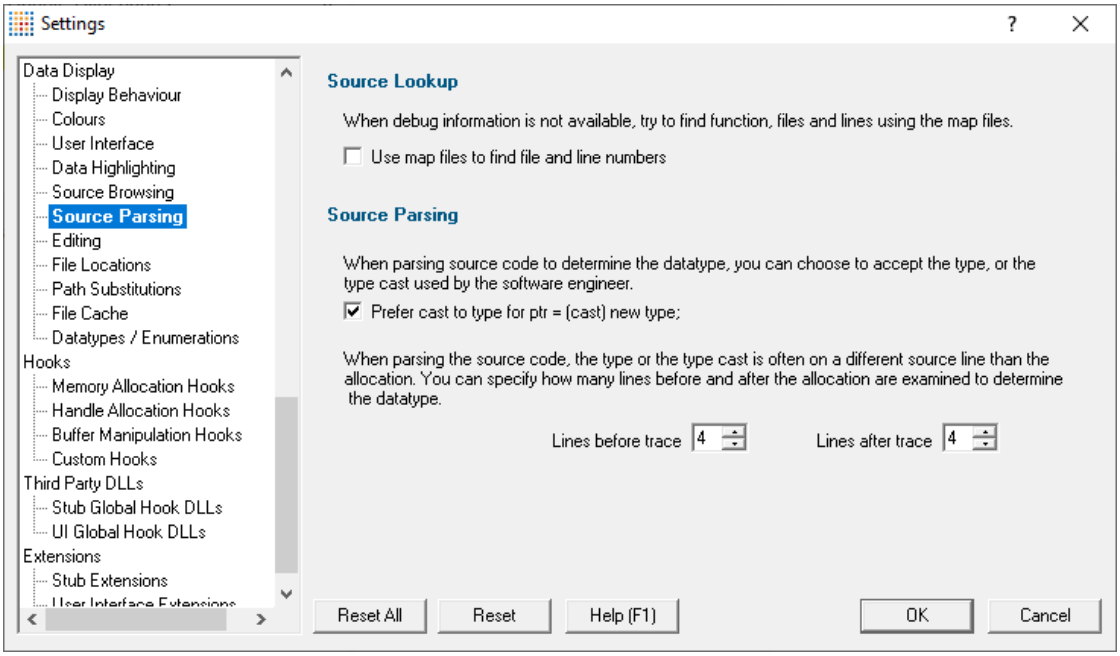
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.6 Source Parsing

The **Source Parsing** tab allows you control how source code is parsed and what the default behaviour is when symbol debugging information is not found.

The default options are shown below:



Source lookup

When debugging information is missing or unavailable, you can choose one or more of the following methods to try and resolve the symbols:

- **Use map files to find file and line numbers** › use linker .map files. Faster than using .Bsc files
- **Map Ordinals to function names** › use linker definition .def files. Fast, though not usually needed in addition to the other two options
- **Use Bsc files to find file and line numbers** › use compiler .Bsc source code browser files

➜ See also: topics on File Locations and Ordinal Handling,

Source parsing for data types

Normally Memory Validator can get data type information from the debug heap, but this is not always possible (see below for reasons why parsing may be necessary).

Parsing the source code is an alternative way to determine the data type of an allocation, and the following settings are then used:

- **Lazy source code parsing** › the data type will be determined when it needs to be displayed rather than at the time of detecting the allocation

  With 'lazy' parsing, the object type statistics on the Types page will be incorrect, but Memory Validator may appear to execute faster.

- **Prefer cast to type...** › uses the *cast* type at the source code location in preference to the original type

When parsing source code, a few lines before and after the specified line will be examined to try and detect the appropriate type. This is because the debug information doesn't always align exactly with the line in the source code that would be considered correct by a person viewing it.

- **Lines before / after trace** › specify how many lines of source around the allocation point are examined in order to try and determine the data type

Why source code parsing may be necessary

As mentioned above, Memory Validator can normally get data type information from the debug heap.

However, the type information in the debug heap is not always present, depending on how the source code was compiled, and in particular, that information is of no help for allocations made using non CRT functions such as `HeapAlloc()`.

For some Win32 handle functions, the handle type is defined by the function, so the type doesn't need to be determined, but for those allocations where a type could not be found, Memory Validator will try to ascertain the type by examining the source code.

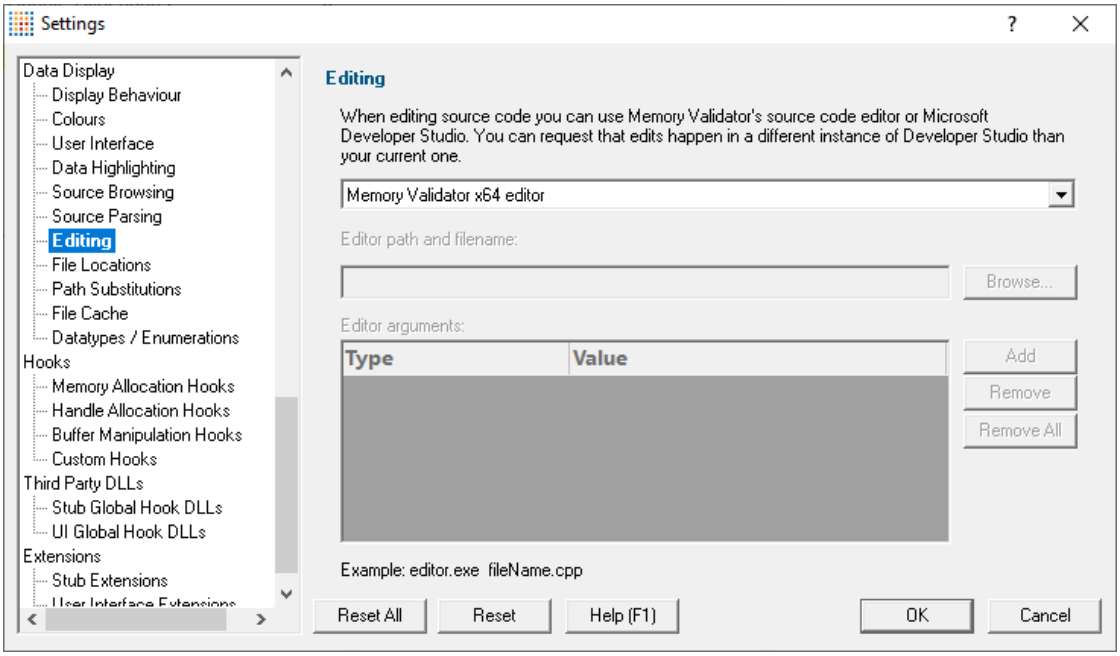
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.7 Editing

The **Editing** tab allows you to configure which editor Memory Validator will use for editing source code.

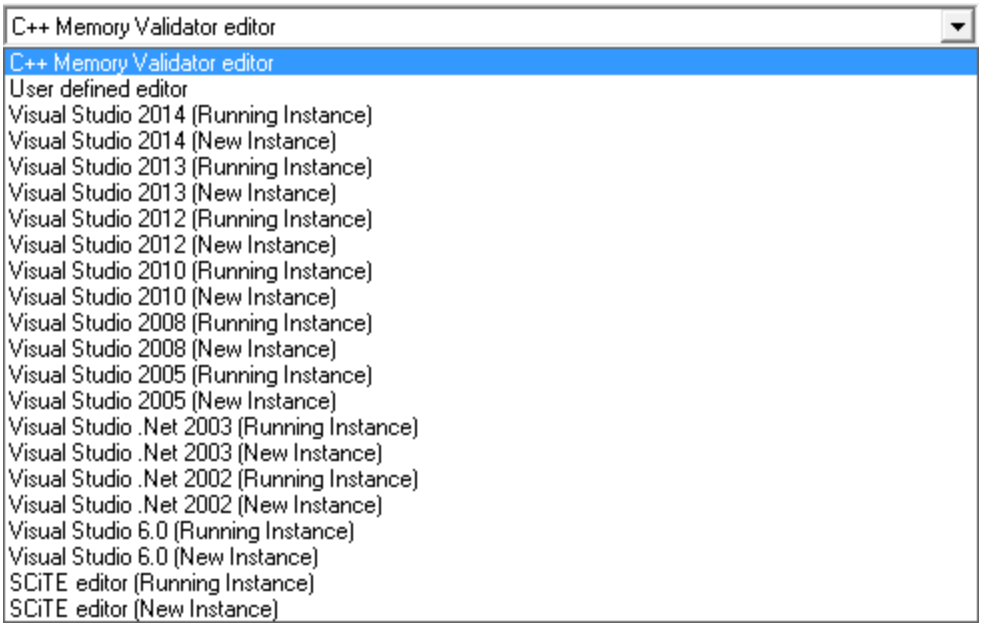
The default settings are shown below:



Editing source code

There are several places in Memory Validator where you can choose to edit source code, for example, while viewing callstacks or other source code fragments.

By default, source code is opened in a provided source code editor using syntax colouring, but you can change where you edit code via the drop-down menu:



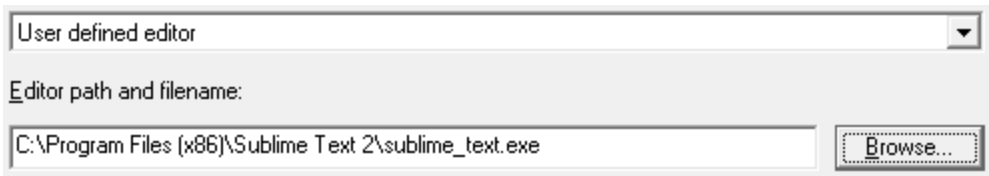
When choosing one of the editors listed, you can request a currently open instance (e.g. the same one you are using to develop your application), or to open a new instance.

➜ SCiTE is included in the list of editors, but there are many text editors that can be used for source code on windows. Wikipedia has a comparison of editors including their programming feature support

Editing with your preferred editor

We've all got our favourite editors! To use yours:

- Select **User defined editor** from the list of options › enables the fields below
- Enter the **Editor path and filename** or just **Browse** › choose the executable for your preferred editor



Now when you want to edit source code, that editor will be opened, but typically you'll need to specify some command line arguments with which to start the editor.

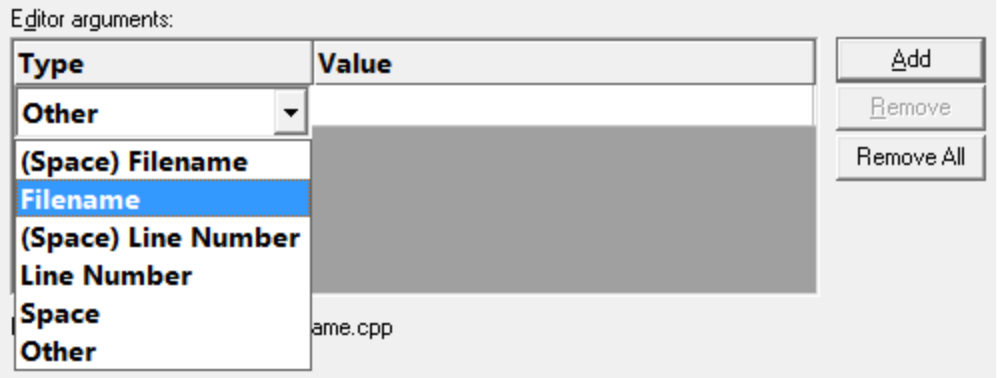
Starting your preferred editor with command line arguments

By default, just the file name is passed as a command line argument to the editor.

Depending on the editor, you may need to tailor the arguments, especially if you want the file opened at the allocation line for example.

The arguments can be specified by adding them to the table provided, one at a time and in the order required

- **Add** › adds a row to the **Editor arguments** table › select an argument **Type** from the following options



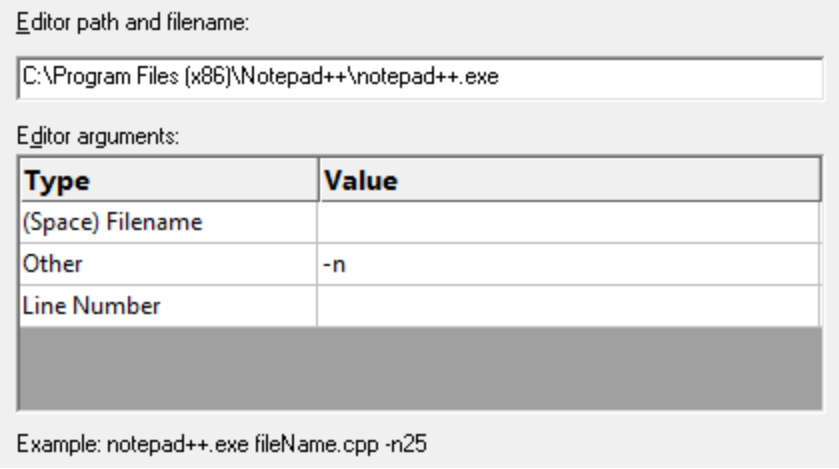
The possible arguments include

- **(Space) Filename** › appends a space followed by the filename
- **Filename** › appends just the filename
- **(Space) Line Number** › a space followed by the line number
- **Line Number** › just the line number.
- **Space** › a space.
- **Other** › appends the text typed in the **Value** column of the list. Press return after entering the value.

Only the last option needs an entry in the Value column

The *Other* option is currently also prefixed by a space. You will need to press **Return** after entering the value otherwise the entry won't get recognized.

The example below configures NotePad++ to edit a file at the required line using the -n switch



As you modify the arguments an example command line is shown below the list.

Managing the command line arguments

Edit a Type or Value by double clicking the entry. The usual controls apply for removing list items:

- **Remove** › removes selected arguments in the list
- **Remove All** › removes all arguments, clearing the list

Alternatively, press Del to delete selected items, and Ctrl + A to select all items in the list first.

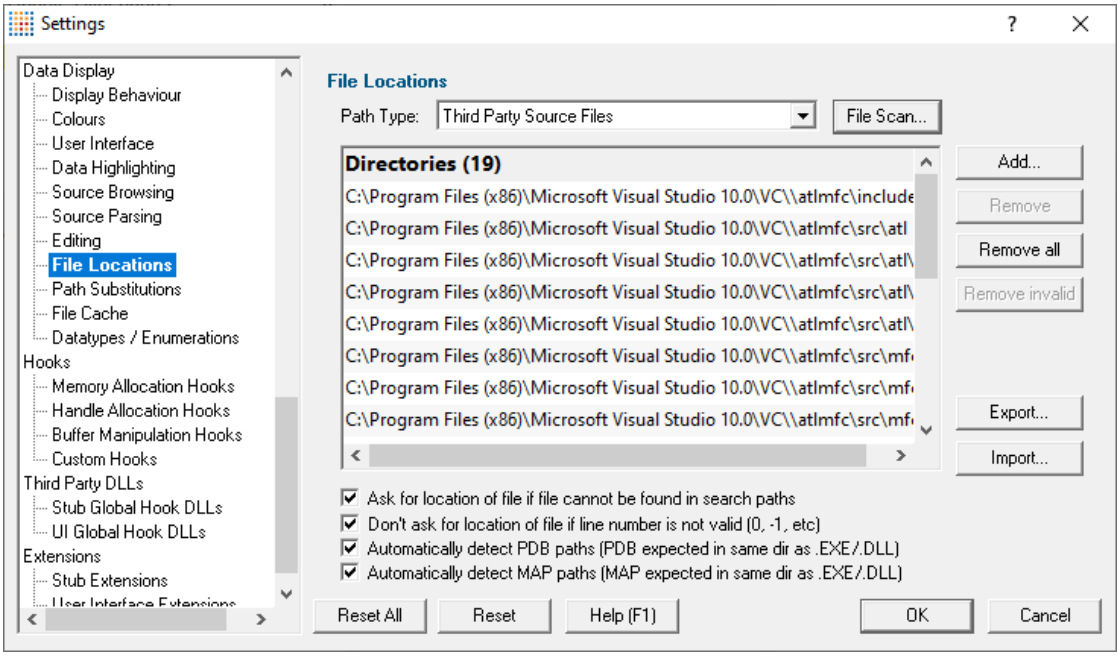
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.8 File Locations

The **File Locations** tab allows you to specify which directories Memory Validator should look in for source code files, PDB files, and others.

The default settings are shown below:



File locations

Sometimes the information Memory Validator has access to consists of the file name, but not the directory.

When this happens Memory Validator scans a set of directories that it knows about in order to find the file.

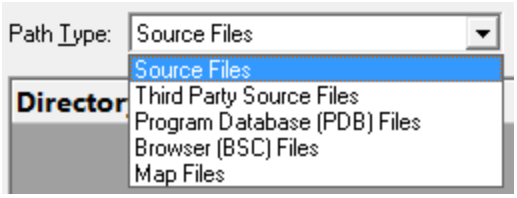
The options below allow you to specify those directories that should be searched for source files, PDB files, MAP files, etc.

If a file can't be found, you'll get prompted for its location, but you can control this below as well.

Setting directories for a path type

There are five path types, and a separate list of directories to scan for each one.

- **Path Type** › select the type of file with which you want to modify the list directory



You don't *have* to specify any directories if you don't want to or if you just don't have them. Nor do you have to give directories for *all* the path types.

Prompting for file locations

Whenever a file still cannot be found, then the default action is for a dialog to ask you where it is.

To avoid frequent user interruption, it is recommended that the directories for source code files (yours and third party) are specified, enabling Memory Validator to browse source code and detect data types in the background.

If however, you don't want to be prompted for locations, you can disable that too.

- **Ask for location of file...** › untick to stop prompting for file locations

When prompting is switched on, it can happen that the line in question is invalid anyway!

The default is not to prompt for invalid lines, but if you want to know when that happens, just switch that behaviour off.

- **don't ask for location of file if line number is not valid...** › untick to be prompted for invalid lines anyway

PDB (program database) file paths

Normally PDB search paths are automatically generated, based on the same directories that .exe and .dll files are found in:

- **Automatically detect PDB paths** › automatically detect PDB locations (the default)

However, it is recommended that you specify paths for PDB (program database) files, especially if your build environment dictates that PDB files are kept in different directories to their binaries.

If you don't automatically generate PDB paths and you don't specify any paths for PDBs, the search path will be defined as the current directory plus any paths found in the following environment variables:

- _NT_SYMBOL_PATH
- _NT_ALTERNATE_SYMBOL_PATH
- SYSTEMROOT

MAP file paths

Normally MAP search paths are automatically generated, based on the same directories that .exe and .dll files are found in:

- **Automatically detect MAP paths** › auto-detect MAP locations - the default, and strongly recommended

  Note that you must also enable the use of MAP files on the Source Parsing settings.

Manually adding path type directories

Once you have chosen your path type you can modify the list of files for each path type in the following ways:

- **Add** › appends a row to the directory list › enter the directory path

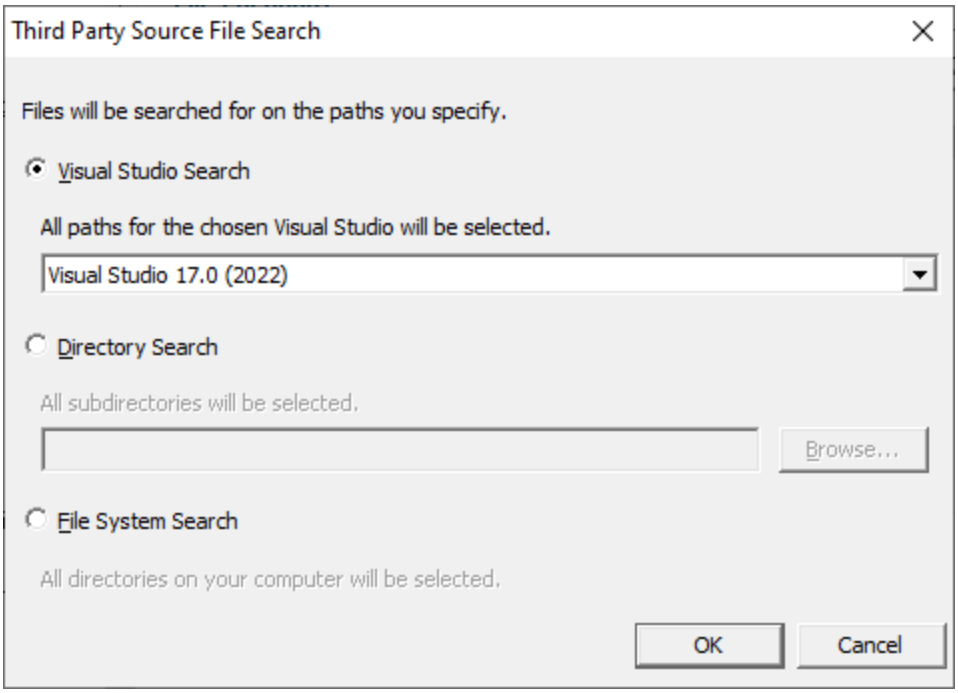
Edit a directory path by double clicking the entry. The usual controls apply for removing list items:

- **Remove** › removes selected items from the list
- **Remove all** › clears the list
- **Remove invalid** › removes all items that are not valid directories from the list

Alternatively, press Del to delete selected items, and Ctrl + A to select all items in the list first.

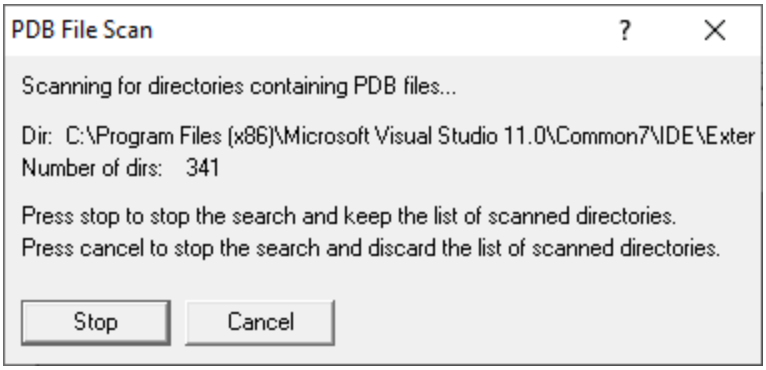
Scanning for directories to add

The **File Scan...** button displays the File Search dialog to provide three ways of specifying the files to scan.

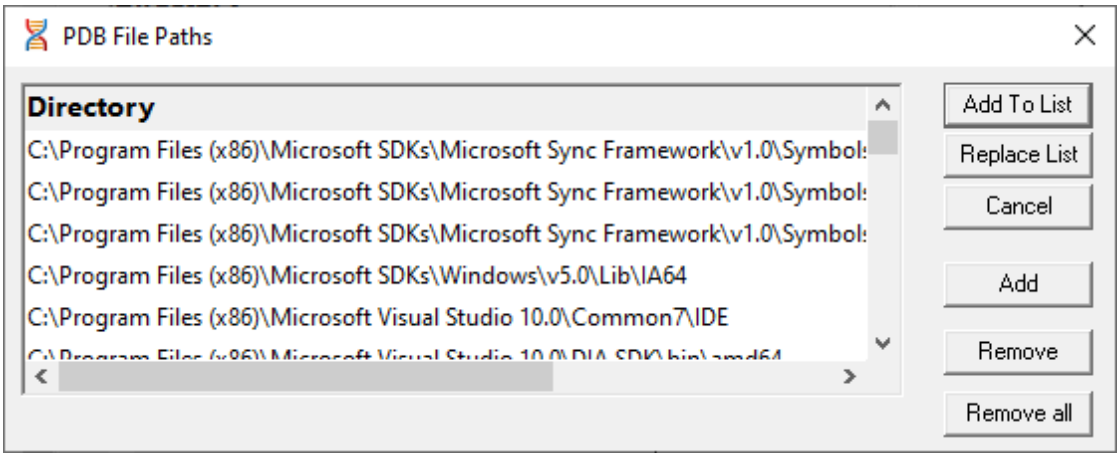


- **Visual Studio Search** › choose the version of Visual Studio › **OK** › starts a scan for directories related to that version of Visual Studio
- **Directory Search** › **Browse...** displays a directory browser › **navigate** to a location you want to scan within › **OK** › starts a scan for directories
- **File System Search** › **OK** › starts a scan of all drives for directories containing files

All options will bring up a **File Scan** dialog indicating number of relevant directories found, and giving you a chance to **Stop** or **Cancel** the scan at any time:



Once the scan is complete you'll see the **File Paths** dialog showing you the scan results:



You can modify the list of resulting directories by adding, removing or editing, exactly as for the path type list above.

Once you're happy with the scan results, either append or replace the path type directories with the scan results.

- **Add To List** › adds the scan results list to the path type directories and closes the File Paths dialog
- **Replace List** › replaces the path type directories with the scan results
- **Cancel** › discard the scan results and close the dialog

Exporting and Importing

Since the list of path types and their file locations can be quite complicated to set up and optimise, you can export the settings to a file and import them again later. This is useful when switching between different target applications.

- **Export...** › **choose or enter** a filename › **Save** › outputs all the path types and their file locations to the file
- **Import...** › **navigate** to an existing *.mvx file › **Open** › loads the hooking rule and the list of modules

The exported file can be used with the -fileLocations command line option.

Export file format

The file format is plain text with one folder listed per line. Sections are denoted by a line containing `[Files]` (for source code files), `[Third]` (for third party source code files), `[PDB]` etc.

Example:

```
[Files]
c:\work\project1\
```

```
[Third]
d:\VisualStudio\VC98\Include
[PDB]
c:\work\project3\debug
c:\work\project3\release
[MAP]
c:\work\project3\debug
c:\work\project3\release
```

Checking directory scanning order

To see the order in which the *DbgHelp.dll* process checks directories to find symbols, see the diagnostic tab with the filter set to *DbgHelp debug*.

Reset All - Resets **all** global settings, not just those on the current page.

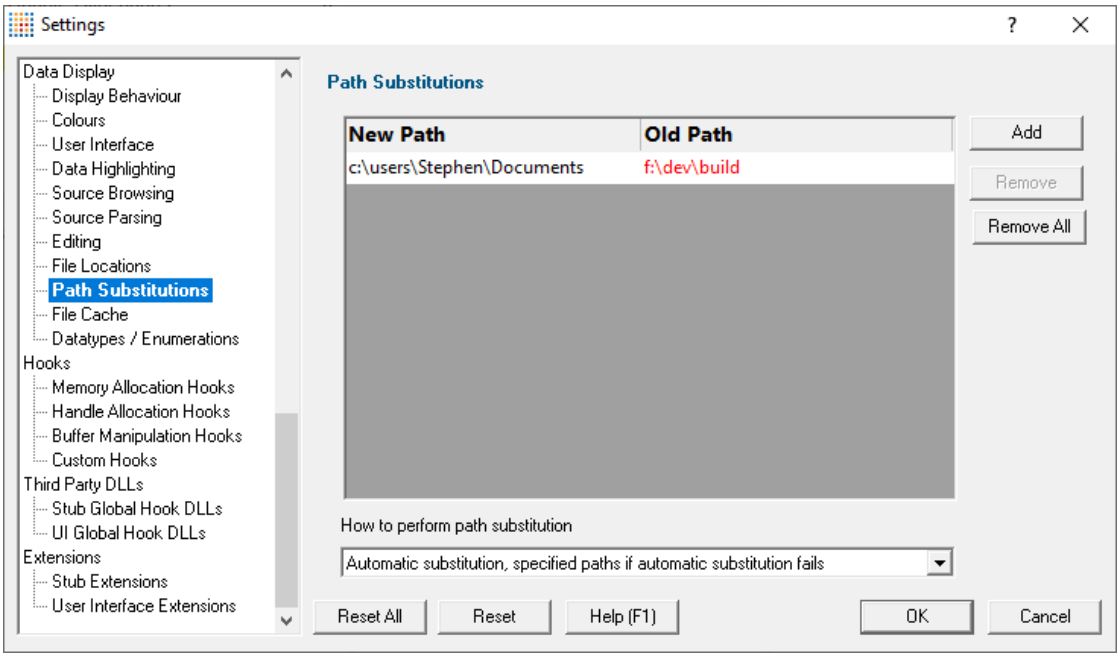
Currently, the four checkbox items at the bottom of this page are **not** reset as part of the global settings.

Reset - Resets the settings on the current page.

3.12.1.7.9 Path Substitutions

The **Path Substitutions** tab allows you to specify file path substitutions to handle copying builds from build machines to development or test machines .

The default settings are shown below:



Path Substitutions

Some software development schemes have multiple rolling builds of their software, often enabled by using substituted disk drive naming schemes.

When you download the build to your development machine for development and testing, debugging information may reference disk drives that don't exist on your machine, for example, drive X: while your machine only has C:, D:, and E: drives.

Or you may just be copying a build from a drive on a development machine to a subdirectory on a drive on your test machine.

These options let you remap the substitution so that the Memory Validator looks in the correct place for the source code.

- **Add** › adds a row to the **File Paths Substitutions** table › enter the new path that will replace the old path in the **New Path** column › click in the Old **Path** column › enter the path that is being replaced

  For example, you might enter c:\users\stephen\documents for the new path and `f:\dev\build` for the old path.

You can double click to edit drives and paths in the table, or remove items:

- **Remove** › removes selected substitutions from the list
- **Remove All** › removes all substitutions from the list

Alternatively, press Del to delete selected items, and Ctrl + A to select all items in the list first.

Example: Changed disk drive

Project originally located at	m:\dev\build\testApp
Project copied to	e:\dev\build\testApp
New Path	e:\
Old Path	m:\

Example: Project copied to a new location

Project originally located at	f:\dev\build\testApp
Project copied to	C:\Users\Stephen\Documents\testApp
New Path	C:\Users\Stephen\Documents
Old Path	f:\dev\build

The slashes do not have to match, a forward slash will match a backslash when comparing path fragments. This is deliberate - to improve ease of use with libraries built by different compilers (LLVM and compilers that use it use forward slashes, whereas Visual Studio etc use backslashes).

Path Substitution Method

Path substitution can be turned off, use only manually specified paths, perform automatic path substitution based on best guesses based on information in the executable, or a combination.

Use the combo box to choose the appropriate path substitution method. The default is automatic path substitution and if that fails to try path substitution using the manually specified paths.

- **No path substitution** › path substitution does not happen
- **Only substitute specified paths** › path substitution uses the manually specified paths
- **Automatic substitution only** › path substitution is performed automatically using information in the executable
- **Automatic substitution, specified paths if substitution fails** › an attempt at automatic path substitution is made, if this fails path substitution is performed using the manually specified paths

The default is **Automatic substitution, specified paths if substitution fails**.

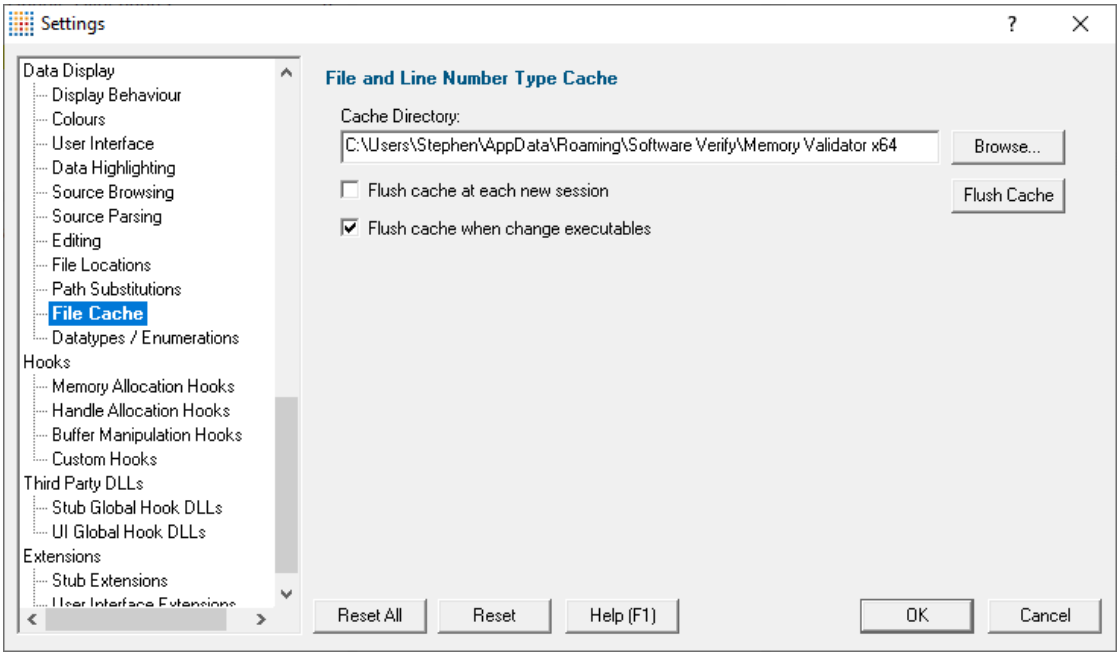
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.7.10 File Cache

The **File Cache** tab allows you to specify where cached information is stored and when it gets cleared.

The default settings are shown below:



Caching file locations

Memory Validator keeps a cache of known locations for files for which it needed to search, improving the speed at which files can be found.

- **Cache Directory** › type directly or **Browse** to find a directory for Memory Validator to cache its information

By default, the cache is only flushed when the executable changes between sessions

- **Flush cache at each new session** › tick to flush the cache every session

  This slightly slows down relaunch of the same executable, as the cache needs rebuilding.

- **Flush cache when executable changes** › untick to prevent the cache being flushed at all

When not automatically flushing, you can manually flush the cache if necessary

- **Flush Cache** › flush the cache now

  This is only possible when no sessions are in the session manager.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

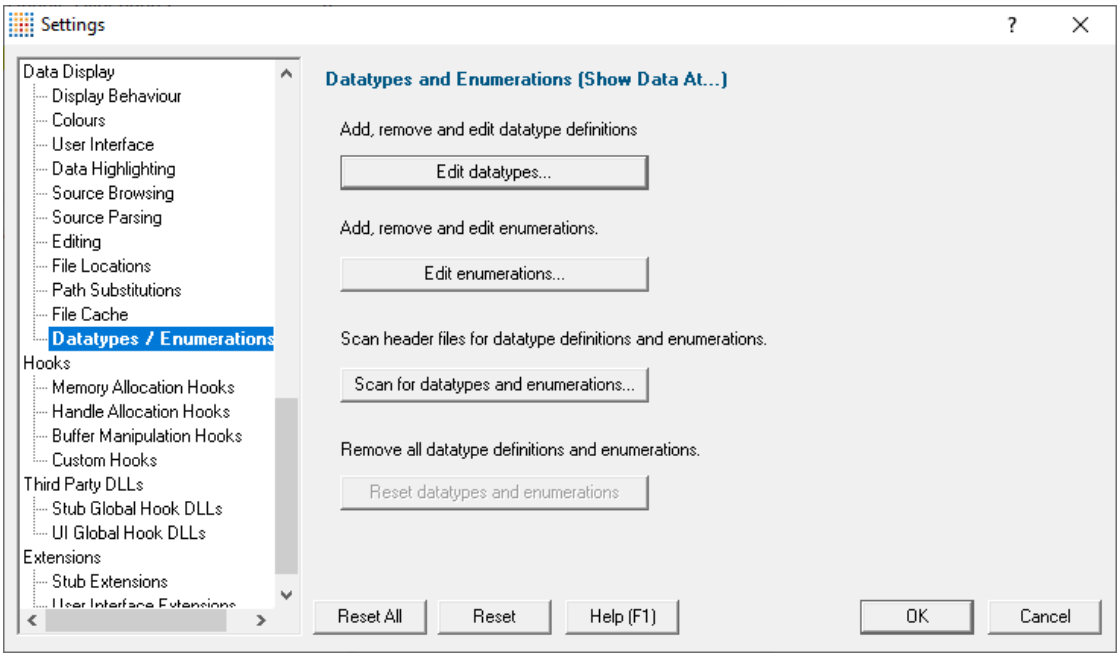
3.12.1.7.11 Datatypes and Enumerations

The **Datatypes and enumerations** tab allows you to:

- scan headers to look for datatypes including typedefs, structs and enumerations
- define or edit data not able to be determined automatically

The Show data at... feature of the Memory tab uses these definitions to inspect allocated memory using a meaningful interpretation of the data.

If you don't define any datatypes or definitions here then memory inspection will just show the same format as the Show Data at (bytes)... option.



- **Edit datatypes...** › shows the Datatypes dialog to edit existing datatypes or add new ones
- **Edit enumerations...** › shows the Enumerations dialog to edit existing enums or add new ones
- **Scan for datatypes and enumerations...** › shows the Scan for Datatypes dialog to extract definitions from header files
- **Reset datatypes and enumerations** › clears any definitions previously defined or found in headers

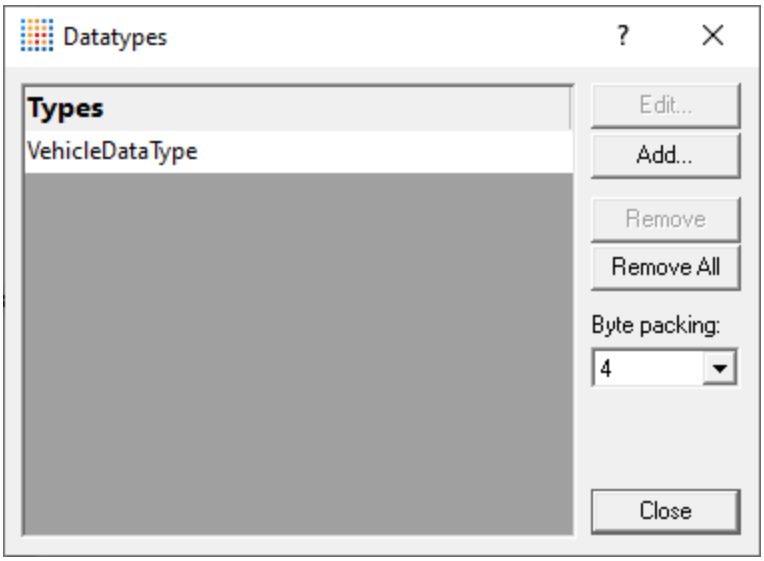
> Don't accidentally click the **Reset** button at the bottom of the page as that will reset *all* settings!

You can define standalone datatypes and enumerations, as well as those to be used in more complex datatypes.

Make sure you click OK not Cancel on the global settings dialog when you're done or you'll lose any changes.

Definitions will persist between sessions.

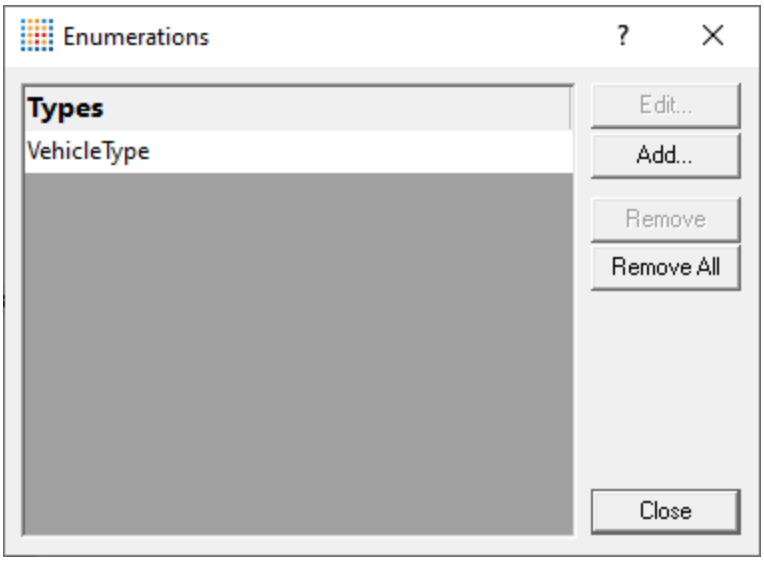
Datatypes Dialog



- **Add...** › shows the Define Data dialog to add a new definition
- **Edit...** › opens the Define Data dialog to modify or just review an existing definition
- **Remove** › removes any selected datatypes from the list
- **Byte packing** › sets the number of bytes into which the datatype is to be packed and aligned

Enumerations dialog

The enumerations dialog is almost identical to the Datatypes dialog above:



- **Add...** › shows the Define Enumeration dialog to add a new definition
- **Edit...** › opens the same dialog to modify or just review an existing enum

- **Remove** › removes any selected enumerations from the list

Defining a new datatype

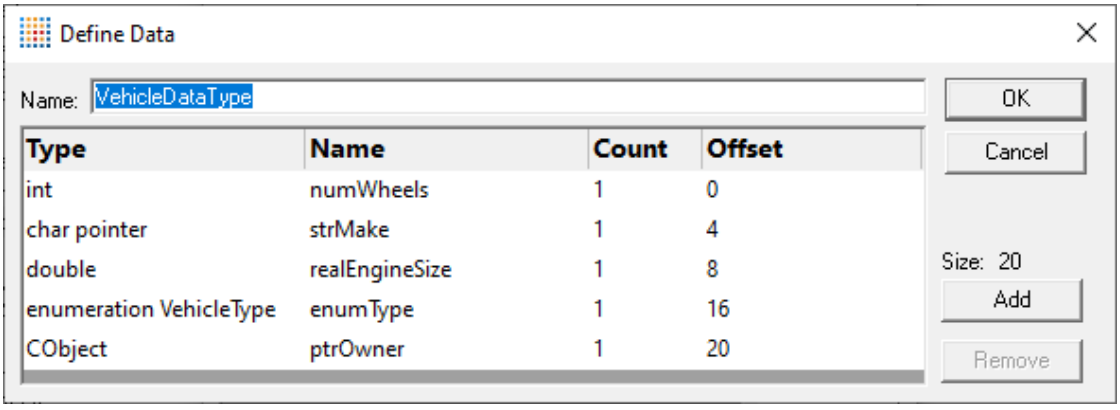
To define a new datatype you need to enter the name of the structure, and details about each member.

You'll need to define any referenced types first before you define a structure that uses it.

We'll demonstrate the datatype definition with the following examples:

```
enum VehicleType
{
   VehicleType_Car    = 0,
   VehicleType_Bus    = 1,
   VehicleType_Van    = 2,
   VehicleType_Lorry  = 3,
   VehicleType_Bicycle = 4,
   VehicleType_Motorbike = 5,
   VehicleType_Scooter = 6,
   VehicleType_Skateboard = 7
};

struct VehicleDataType
{
   int               numWheels;
   char*             strMake;
   double            realEngineSize;
   enum VehicleType  enumType;
   CObject*          ptrOwner;
};
```

- **Edit datatypes...** › **Datatypes dialog** › **Add** › shows the Define Data dialog



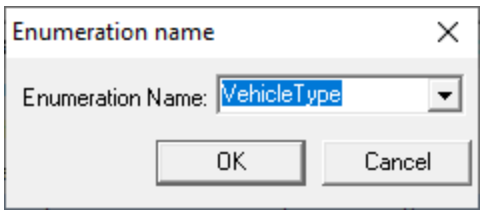
- **Name** › specify the name of the datatype, `VehicleDataType`

- **Add** › appends a new row to the data to define a new member

  Double click to show the drop down list in the **Type** column if not already shown.

  The list has many core datatypes and will include any datatypes you've already defined.

  If the type is an enumeration, you'll be prompted for the name of the enumeration which you should have already defined.



  Enter the name of the data member in the **Name** column.

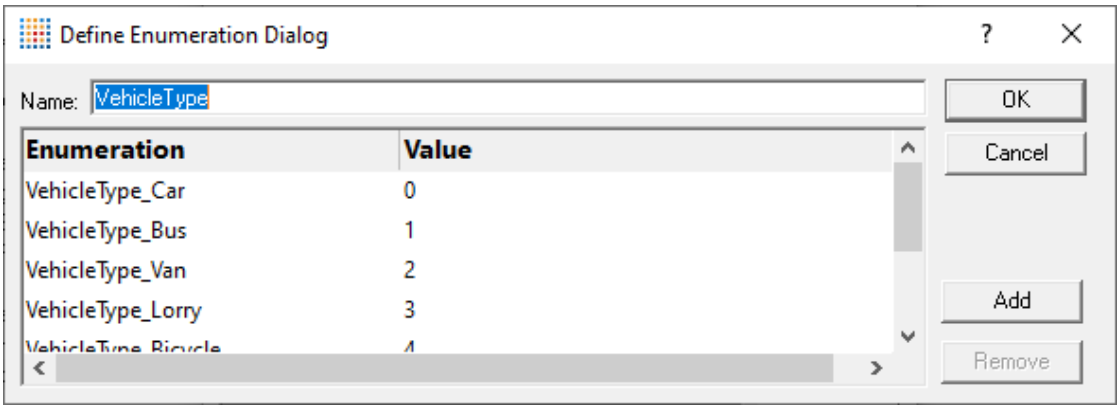
  If more than one, Specify how many items the member represents in the **Count** column, e.g. for an array.

  Values in the **Offset** column will be calculated automatically.

- **Remove** › removes any selected data members from the list

Defining a new enumeration

- **Edit enumerations...** › **Enumerations dialog** › **Add** › shows the Define Enumeration dialog



Any enumerations defined here can be used in the Define Data dialog above.

- **Name** › specify the name of the enum, `VehicleType`
- **Add** › appends a new row to the data to define a new member

  Enter the name of the member in the **Enumeration** column.

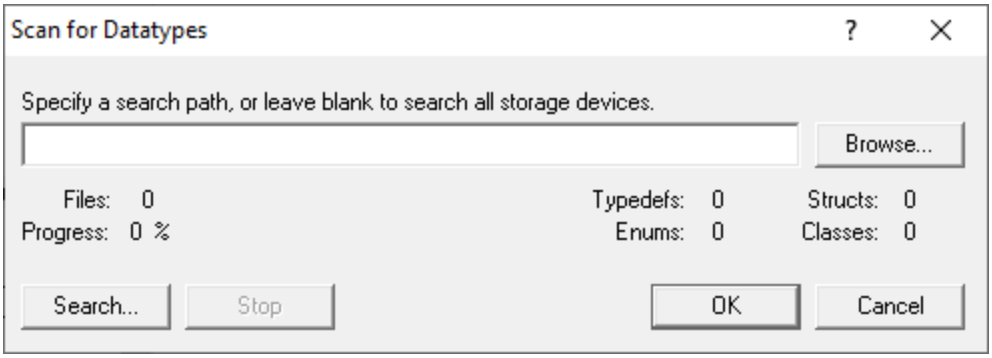
Set the corresponding value in the **Value** column

- **Remove** › removes any selected members from the list

Scanning for datatypes

You can scan header files to find definitions for datatypes or enumerations which are not otherwise detectable.

- **Scan for datatypes and enumerations...** › shows the Scan for Datatypes dialog



- **Browse...** › type or browse to enter a directory or drive to scan for header files from which to extract datatypes

  Leave this blank to scan all available drives.

- **Search...** › start header file scanning
- **Stop** › stop the search at any time
- **Cancel** › discard the results of the search and close the dialog

Any datatypes found during the scan are added to the lists in the datatypes dialog or the enumerations dialog.

At the time of writing, scans don't detect datatypes inside classes and 64 bit datatypes are not supported.

Reset All - Resets **all** global settings, not just those on the current page.

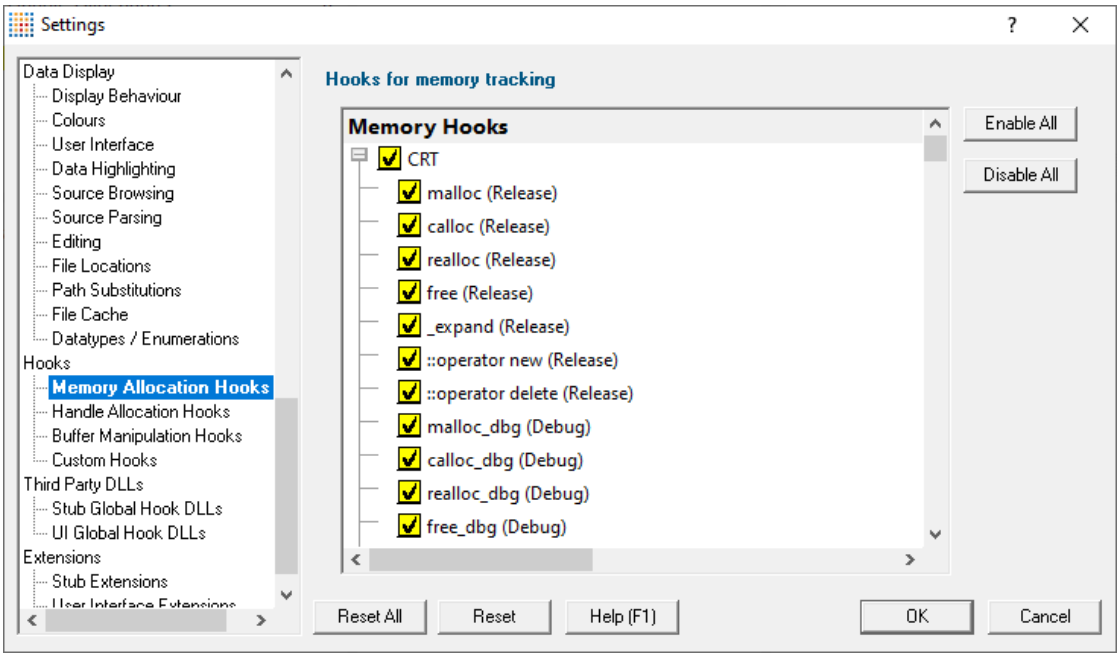
Reset - Resets the settings on the current page.

3.12.1.8 Hooks

3.12.1.8.1 Memory Allocation Hooks

The **Memory Allocation Hooks** tab allows you to enable and disable every hook or group of hooks that Memory Validator uses to track memory allocations.

The picture below doesn't show all the options, but the default is to have everything enabled (ticked):



Memory hooks

Enable or disable a hook by ticking or unticking the yellow boxes.

Note that because the C runtime is provided in both Release and Debug, some function names are present twice, with their status indicated in brackets.

- **Enable All** › ticks all the hooks, enabling everything
- **Disable All** › unticks everything

Memory hook groups

All the hooks in the following groups can be enabled and disabled at once by ticking the group checkbox

- **CRT** › the C runtime hooks for release and debug libraries `MSVCRT.DLL / MSVCRTD.DLL`
- **Memory allocation** › the `HeapAlloc`, `VirtualAlloc`, `GlobalAlloc` and `LocalAlloc` function groups

- **LocalAlloc allocations by other functions** › functions that use `LocalAlloc` to allocate workspace that should be freed by the caller
- **GlobalAlloc allocations by other functions** › functions that use `GlobalAlloc` to allocate workspace that should be freed by the caller
- **CoTaskMemAlloc allocations by other functions** › functions that use `CoTaskMemAlloc` to allocate workspace that should be freed by the caller
- **COM/OLE Allocators** › `CoTaskMemAlloc` function group and `IMallocSpy`
- **BSTR Allocations** › `SysAllocString` function group and `VariantClear`
- **Net API Allocations** › `NetApiBufferAllocate` function group
- **Misc Allocations** › miscellaneous functions that allocate and deallocate objects for various API areas such as security and encryption
- **Fortran 95 Allocations**
- **Delphi allocations**

➡ For details on the contents of each of these hook groups consult the Hook Reference.

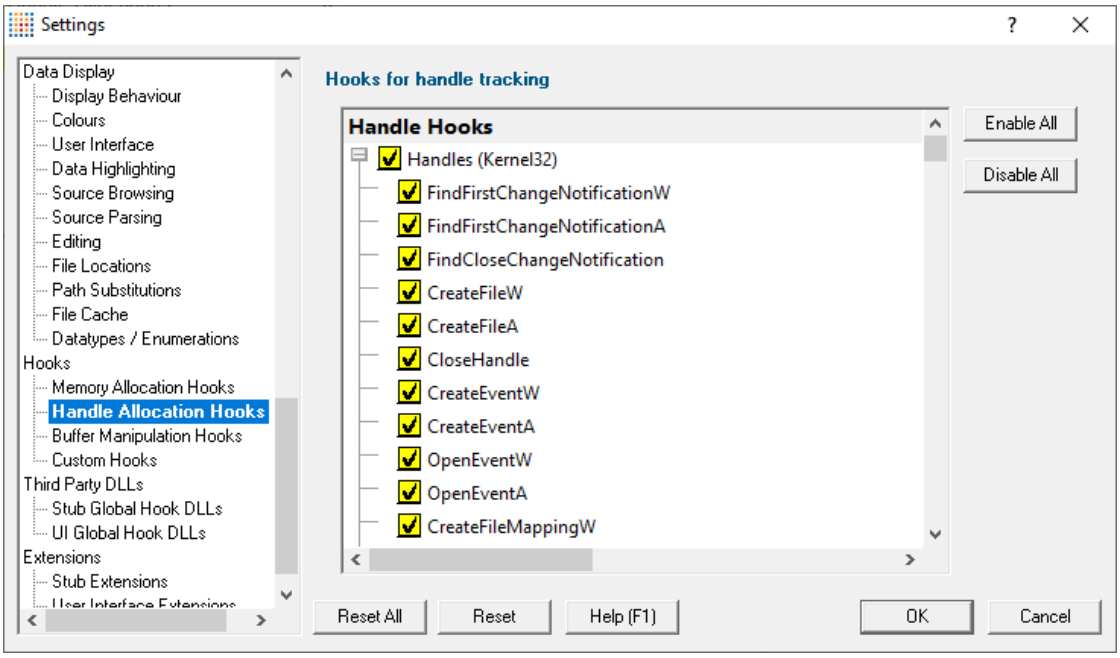
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.8.2 Handle Allocation Hooks

The **Handle Allocation Hooks** tab allows you to enable and disable every hook or group of hooks that Memory Validator uses to track handles.

The default is to have everything enabled (ticked):



Handle hooks

Enable or disable a hook by ticking or unticking the yellow boxes.

- **Enable All** › ticks all the hooks, enabling everything
- **Disable All** › unticks everything

Handle hook groups

All the handle creation and deletion hooks in the following groups can be enabled and disabled at once by ticking the group checkbox next to the group name

- **Kernel32**
- **Advapi32**
- **GDI32**
- **User32**
- **Shell32**
- **COMCTL**
- **Sockets**
- **WinHttp**
- **Printer**

➜ For details on the contents of each of these hook groups consult the Hook Reference.

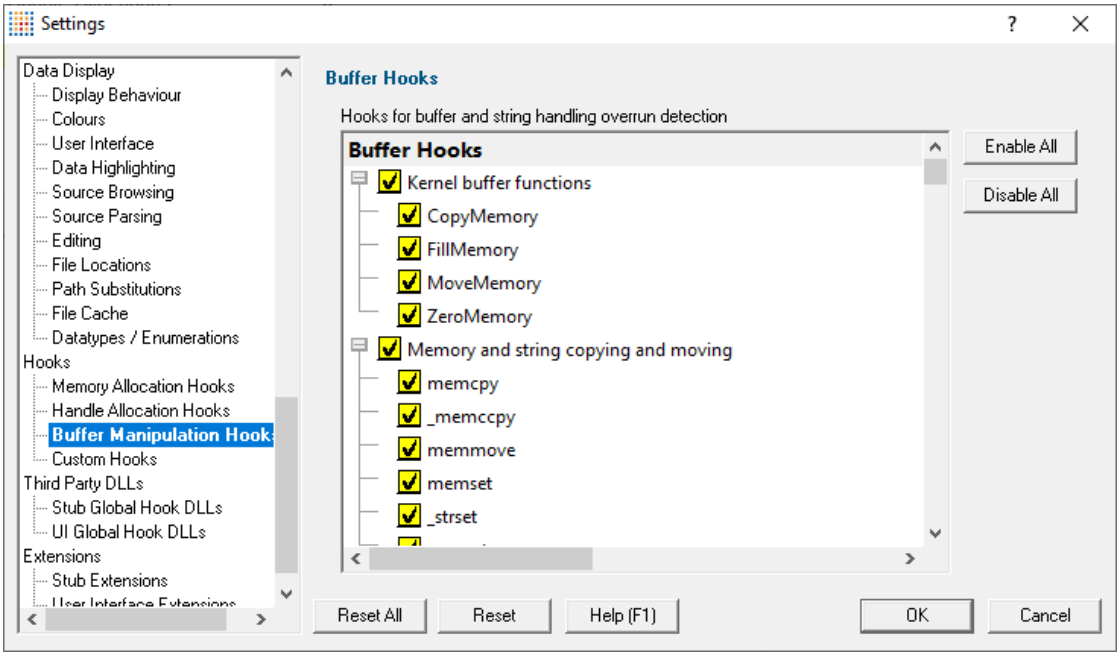
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.8.3 Buffer Manipulation Hooks

The **Memory Allocation Hooks** tab allows you to enable and disable every hook or group of hooks that Memory Validator uses to track buffer overruns and buffer underruns.

The default is to have everything enabled (ticked):



Buffer hooks

Enable or disable a hook by ticking or unticking the yellow boxes.

- **Enable All** › ticks all the hooks, enabling everything
- **Disable All** › unticks everything

Buffer hook groups

All the buffer manipulation hooks in the following groups can be enabled and disabled at once by ticking the group checkbox next to the group name

- **Kernel buffer functions**
- **Memory and string copying and moving**
- **Memory and string comparisons**
- **Internet, Path and Registry functions**

➜ For details on the contents of each of these hook groups consult the Hook Reference.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

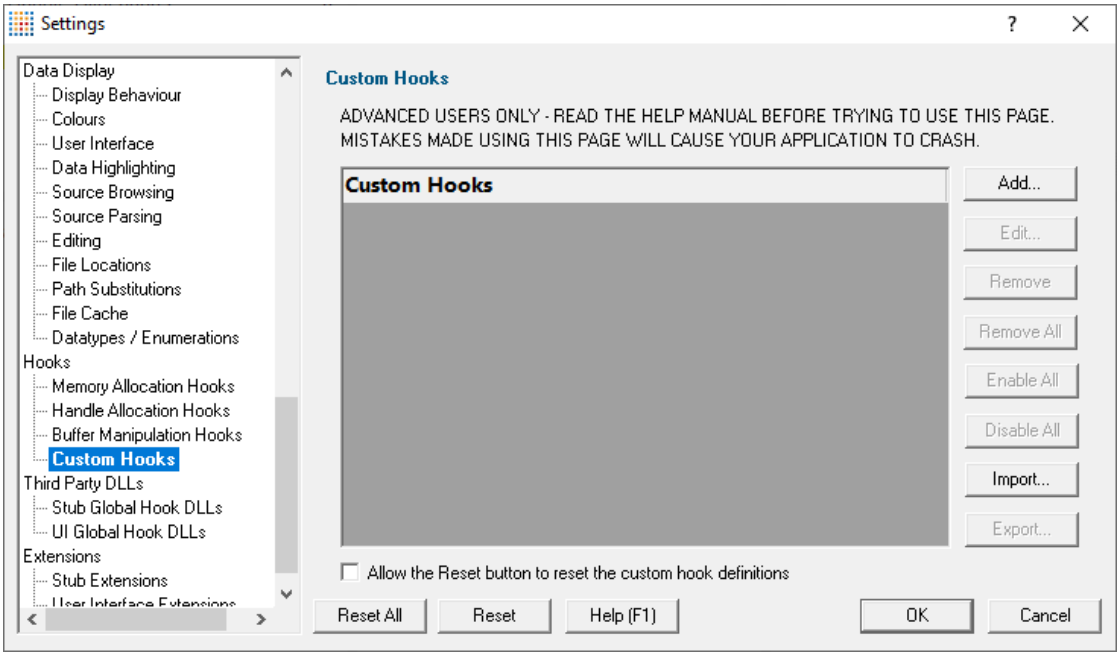
3.12.1.8.4 Custom Hooks

The **Custom Hooks** tab allows you to specify hooks for functions that Memory Validator does not initially know about.

For example, custom hooks might let you monitor APIs in third party products or in APIs that are released after Memory Validator was released.

This a very advanced topic.

Read on, or click on a setting in the picture below to find out more



Using custom hooks

Before adding custom hooks, you need to know the parameter list and return type of each function to be hooked and their calling conventions.

- **Add...** › displays the custom hook dialog (below)

For other options (edit,remove, and enable), see modifying existing custom hooks further below.

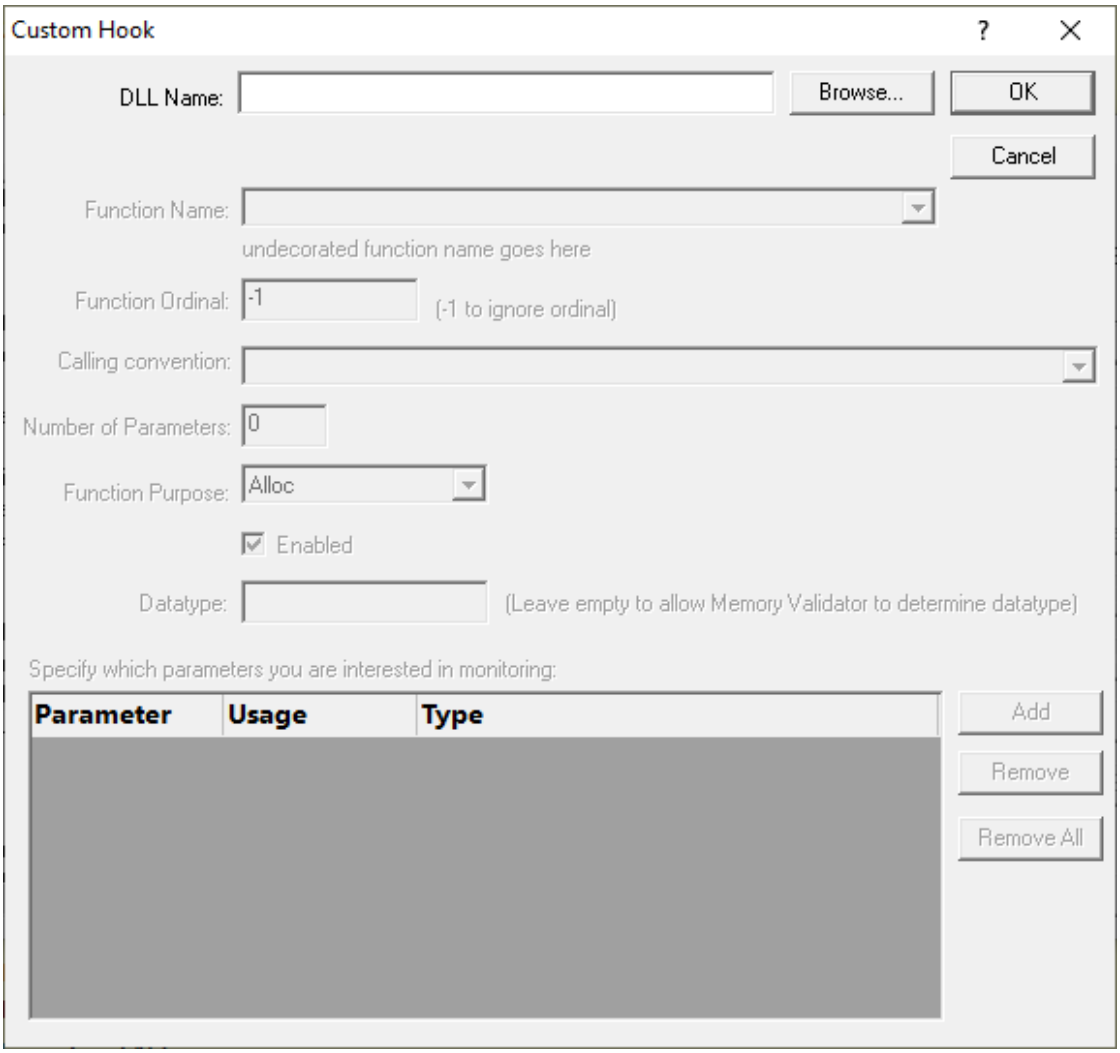
➜ This topic is also discussed in the Memory Validator tutorial, which can be found on the Help menu.

Custom hook dialog

The custom hook dialog allows you to set up or edit the definition of a custom hook, including its parameters.

Take care! Failure to specify the information correctly may crash your application. If in doubt do not attempt to use this feature of Memory Validator.

At the end of this topic are some examples of how this dialog might be filled out for a few different functions.



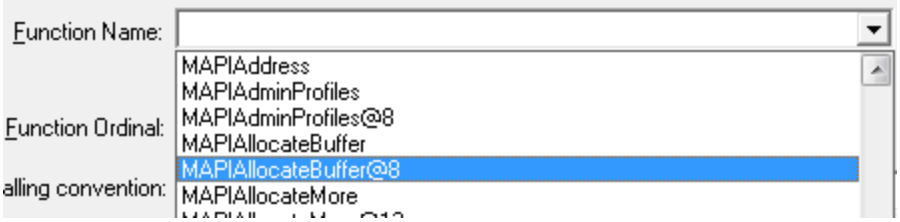
Custom hook definition

- **DLL Name** › **Browse** to navigate to a DLL › **Open** to enter the DLL name into the dialog

  Alternatively you can type the full path, or a relative path to the DLL into the **DLL Name** field.

- **Function Name** › choose the name of the exported function from the drop-down list

  The list is automatically populated with the exported functions after choosing a DLL name above



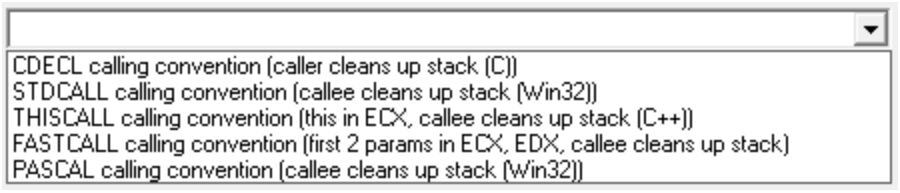
The function name must be exported from the DLL you specified in order to appear in the list. If the function is not exported it cannot be hooked.

- **Function Ordinal** › type the ordinal (decimal integer) if the function is imported by ordinal, otherwise enter -1 to ignore it

  For example `MAPIAllocateBuffer@8` in MAPI32.DLL has the ordinal 13 as found in the Export Address Table of the DLL

  ⊟ Finding the ordinal of an exported function in a DLLOne way to find the ordinal of a function exported from a DLL is to use the dumpbin.exe utility provided with Microsoft Visual Studio. e.g. dumpbin filename.dll /EXPORTSIn Visual Studio 9.0, the dumpbin.exe utility is found in VC\bin and is dependent on link.exe in the same directory and mspdb80.dll in the Common7\IDE directory.

- **Calling convention** › select the option that corresponds to the calling convention of the function



  Do not guess this, if you don't know! Using the wrong calling convention will crash your application

  ➜ See also, the history of calling conventions

- **Number of Parameters** › enter the number of parameters the function takes

  This is the number of parameters the function has, not the number of parameters that you want to monitor (below).

  Again - do not guess this! Using the wrong number of function parameters will crash your application when used with the stdcall calling convention.

- **Function Purpose** › set the function purpose - Alloc, AllocMore, Realloc, or Free

- **Enabled** › tick to enable hooking of the function

  Only takes effect each time your application starts, so changing this in the middle of a session has no effect.

- **Datatype** › enter the type of all allocations via this hook, or **leave empty** and Memory Validator will parse the source code to try and obtain the datatype

  Do not include parentheses in the datatype. `[]` and <> *are* acceptable though.

Custom hook function parameters

The parameters section of the dialog allows you to define which parameters and return value are monitored for the function.

You should specify all the other settings in the custom hook definition (see section above) prior to defining the parameters. This will ensure that the right menu choices are made available while setting up the parameters.

You don't need to monitor all the input parameters, or the return value - just monitor what you need.

Generally for each function purpose (that you will have defined above), different parameter values are usually required:

For example:

- **Alloc** an input size, and an output pointer or return value pointer
- **AllocMore** an input size and an input pointer
- **Realloc** an input size, an input pointer, and an output pointer or return value pointer
- **Free** an input pointer

In some cases, an input parameter may also serve as an output.

If you want to monitor additional parameters, we recommend that you set their type to **Miscellaneous** or **Pointer to Miscellaneous** as appropriate.

- **Add** › adds a row to the parameters table, firstly a return value, then parameters in order 0,1,2...
- **Remove** › removes selected parameter definitions from the list
- **Remove All** › clears the list of parameters

Defining custom hook function parameters

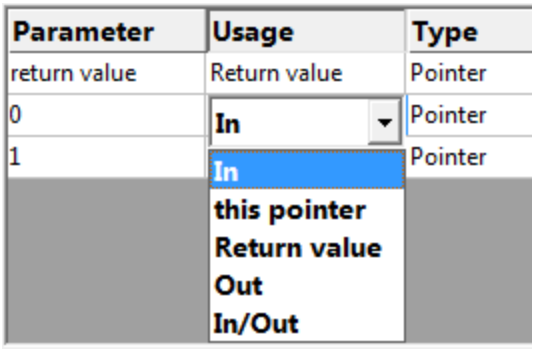
After adding a parameter definition, values can be edited by double clicking on the value in the table. A drop-down list will be displayed with appropriate choices in each column:

- **Parameter** › choose the parameter index in the range 0 to N-1 where N is the number of parameters you specified to the function

  You can only choose an index which is not already used by another parameter

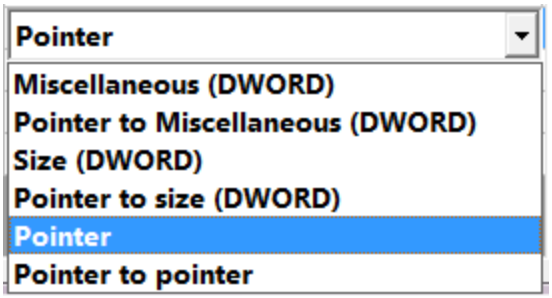
  You can't change the entry for a "return value"

- **Usage** › set how the parameter is used:



In is input only and **Out** is output only, while **In/Out** serves as both
Return value, **this** and any output parameters are usually pointers to something.

- **Type** › choose what the parameter type is:



All **Pointer to...** options are parameters that need dereferencing to read the values.

Memory Validator **x64** also supports the following values:

Size (QWORD)	a size
Pointer to size (QWORD)	a pointer to a size
Miscellaneous (QWORD)	a miscellaneous value
Pointer to Miscellaneous (QWORD)	a pointer to a miscellaneous value

Note that when specifying parameters as **Pointer to ...**, it's OK if that parameter is occasionally NULL - this will be identified and the pointer will not be dereferenced. Many Microsoft APIs allow NULL values for optional data, for example.

Modifying existing custom hooks

Once you have some custom hooks set up you can edit, remove or en/disable them in the following ways:

- **Edit...** or double click a hook in the list › opens the custom hook dialog to change the hook attributes

The usual controls apply for removing list items:

- **Remove** › removes selected items from the list

- **Remove All** › clears the list

  Alternatively, press Del to delete selected items, and Ctrl + A to select all items in the list first.

Enable or disable a custom hook by ticking or unticking the yellow boxes or change them all at once

- **Enable All** › ticks all the hooks, enabling them all
- **Disable All** › unticks everything
- **Import...** › Imports a custom hook XML file.
- **Export...** › Exports the custom hooks as an XML file. This can be useful for moving a custom hook definitions from one computer to another without copying the whole settings file.

Resetting custom hooks

Normally, the custom hooks will not be reset when you press the reset button, but you can make that the case if you wish:

- **Allow the Reset button to reset the custom hook definitions** › when ticked, custom hooks will be reset with all other global settings

Reset All - Resets **most** global settings including those on other pages, but not the settings on this page unless explicitly requested above.

Reset - Resets the settings on the current page.

These examples show how to use the custom hooks dialog

Custom hook dialog - example 1

A custom hook dialog for a function using the `__cdecl` calling convention is shown below.

The function prototype for `customAlloc1` in the DLL `testCustomDLL.dll` (provided) is:

```
extern "C" void *customAlloc1(DWORD   size);      // input param
```

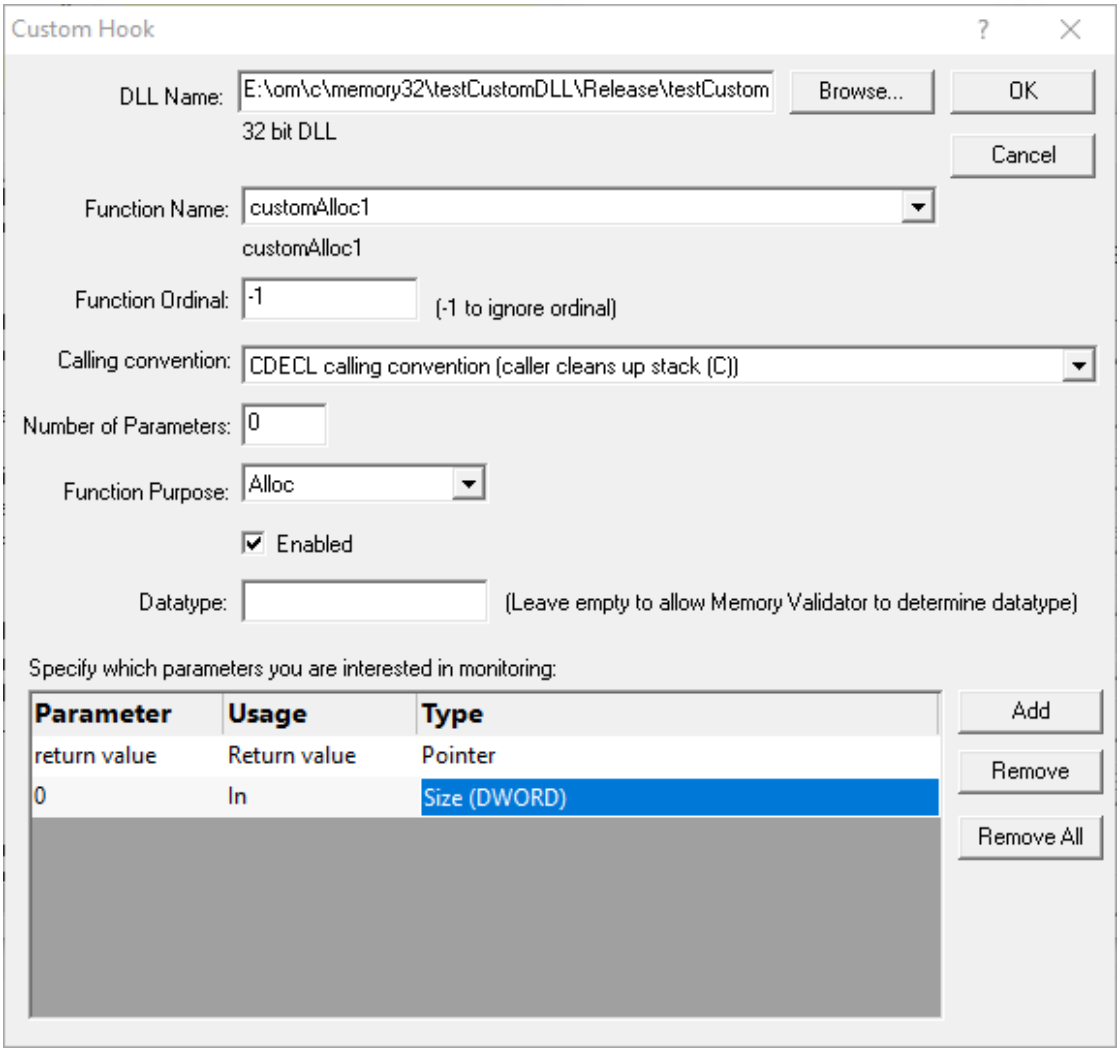
It has been specified using the `extern "C"` specifier so that the function name has no C++ name mangling decoration.

It uses the `__cdecl` calling convention, takes one input parameter and is marked as an allocator by the Alloc definition.

For datatype purposes it can be specified as returning the `BYTE` datatype.

The custom hook will monitor:

- the return parameter, marked as a pointer
- the single input parameter, marked as a size specifier.



Custom hook dialog - example 2

A custom hook dialog for a function using the `__stdcall` calling convention is shown below.

The function prototype for `?customStdCallAlloc3@@YGXKPAPAX@Z` is in the DLL `testCustomDLL.dll` is:

```
extern void __stdcall customStdCallAlloc3(DWORD    size,        // input param
                                           void    **ptr);      // output param
```

It has been compiled as C++, having C++ name mangling decoration.

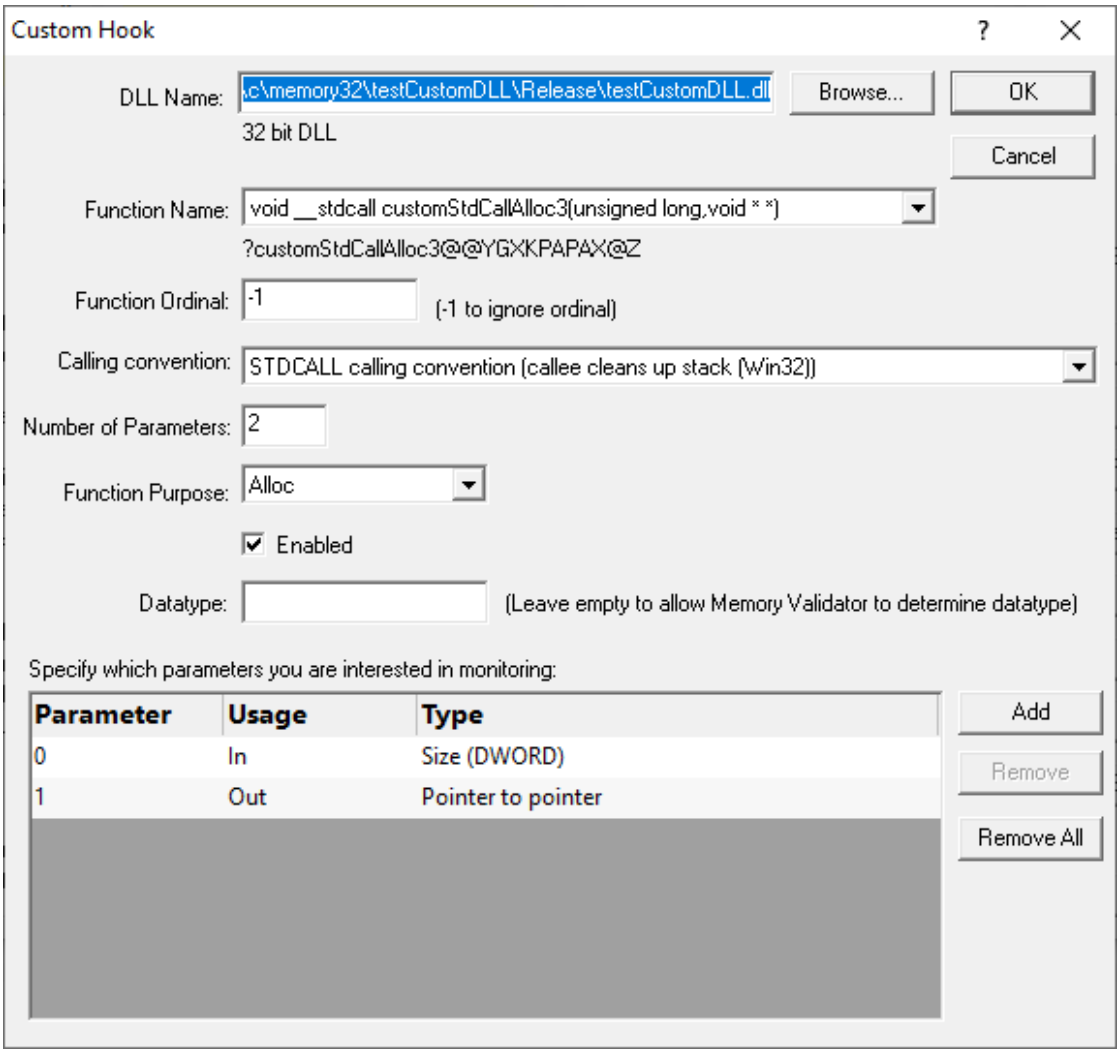
It uses the `__stdcall` calling convention, and takes two input parameters.

The function purpose as an allocator is defined by the Alloc setting.

For datatype purposes it can be specified as returning the `BYTE <STDCALL>` datatype.

The custom hook will monitor:

- the first input parameter, marked as a size specifier
- the second output parameter, marked as a pointer to pointer.



Custom hook dialog - example 3

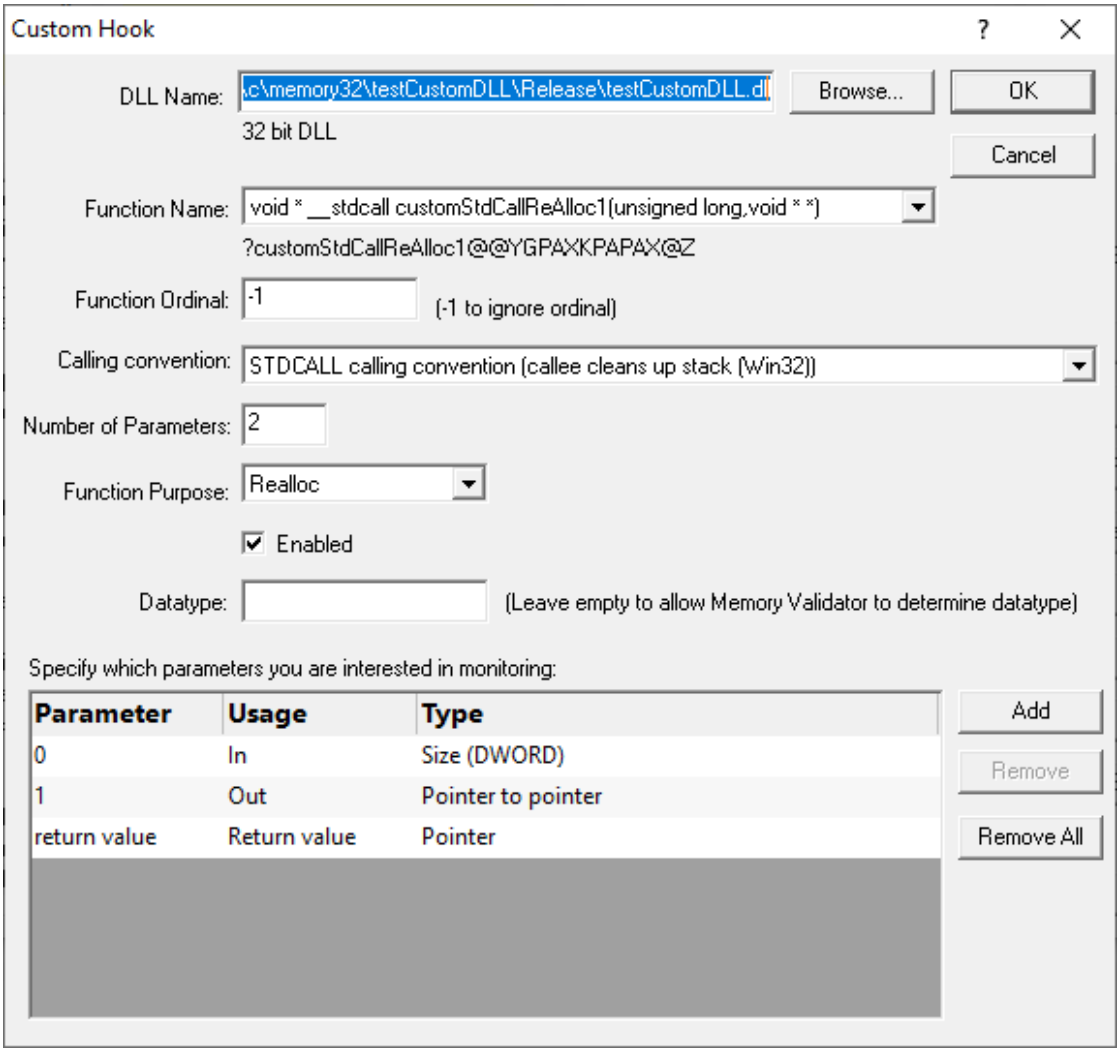
Similar to example 2, except we have a different function name and its purpose is now as a Realloc function, with a pointer return type.

The function prototype for `customStdCallReAlloc1` is:

```
extern void*  __stdcall customStdCallReAlloc1(DWORD   size,         // input param
                                               void    **ptr);      // output param
```

The custom hook will monitor:

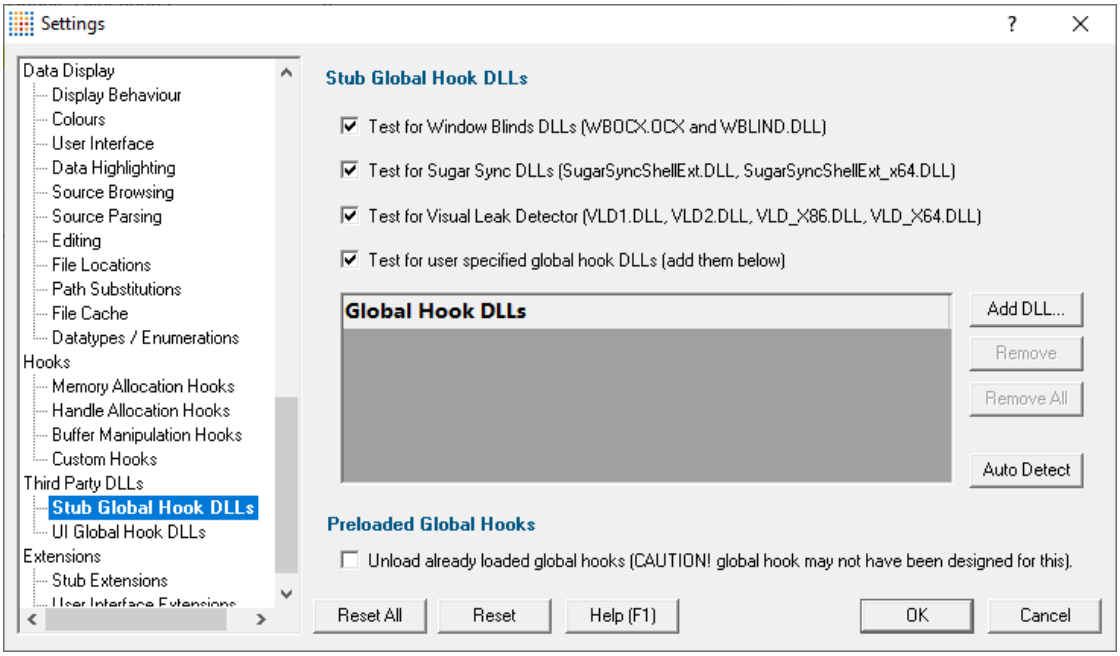
- the return parameter as a pointer
- the first input parameter, marked as a size specifier
- the second input parameter, marked as a pointer to pointer.



3.12.1.9 Third Party DLLs

3.12.1.9.1 Stub Global Hook DLLs

The **Stub Global Hook DLLs** tab allows you to detect and specify global hook DLLs that may not be wanted in the stub process.



About global hook DLLS

Some third party products such as storage devices and video cards are supplied with software to help integrate the hardware device into the computer desktop environment.

An example is the Iomega® Zip® drive. This uses a global hook via the IMGHOOK.DLL which allows the *browse for files* and *browse for folders* interfaces to correctly display all the storage devices on the computer, including the zip drive and any special options for the drive.

Now some global (or *system*) hook DLLs can interfere with the correct operation of Memory Validator when it inserts hooks into the target program, (although the IMGHOOK.DLL mentioned above doesn't).

The settings below allow you to specify and/or detect DLLs that should be treated as global hook DLLs.

Any DLL listed will fail to load into the target program when loaded via `LoadLibrary()` or `LoadLibraryEx()`.

For situations where the hook DLL is already present in the target program, it can optionally be forcibly unloaded. This may happen if it was loaded before Memory Validator attached to the process.

Managing global hook DLLs

- **Test for Window Blinds...** › test for Window Blinds DLLs loading into your application, and prevent them from loading

  Window Blinds DLLs WBOCX.OCX and WBLIND.DLL are not compatible with Memory Validator.

- **Test for Sugar Sync...** › test for Sugar Sync DLLs loading into your application, and prevent them from loading

  Sugar Sync DLLs SugarSyncShellExt.dll and SugarSyncShellExt_x64.dll are not compatible with Memory Validator.

- **Test for Visual Leak Detector...** › test for Visual Leak Detector DLLs loading into your application, and prevent them from loading

  Visual Leak Detector is not compatible with Memory Validator. If you are linked to Visual Leak Detector you'll need to create a build without Visual Leak Detector to use with Memory Validator.

- **Test for user specified...** › test for user specified DLLs loading into your application, and prevent them from loading
- **Add DLL...** › browse and select one or more DLLs › **Open** › adds the chosen DLLs to the **Global Hook DLLs** list
- **Remove** › removes any selected DLL from the list
- **Remove All** › removes all DLLs from the list

Auto detecting global hook DLLs

Memory Validator can detect any DLLs in its own process that are not ones it uses itself. Such DLLs are likely to be global hook DLLs:

- **Auto Detect** › automatically detect DLLs which may be global hook DLLs, *adding* them to the **Global Hook DLLs** list

Additionally, you can request unloading of any of the listed global hook DLLs that are detected as already loaded into the target process when Memory Validator attaches to it:

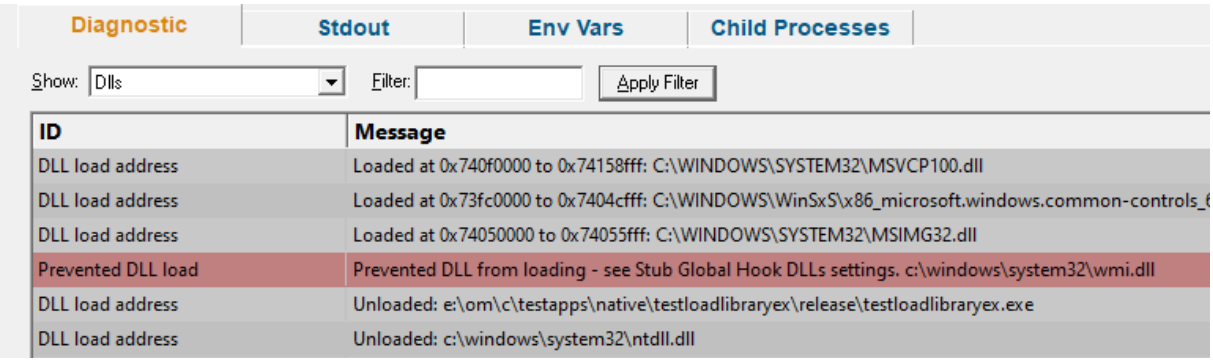
- **Unload already loaded global hooks** › when ticked, forces an unload of any listed global hook DLLs

   Use this with caution, as not all global hook DLLs may have been designed or intended for this!

Viewing the diagnostic information

If a DLL is prevented from loading because of these settings, or is allowed to load because of these settings, there will be an entry on the Diagnostic tab.

To view this data, go to the Diagnostic tab, select the Diagnostic sub tab, then set the Show combo box to "Dlls". All information about DLLs will be shown. Scroll through the list looking for "Prevented DLL load" in the left hand column. The right hand column will indicate if a DLL was prevented from loading, or allowed to load. The DLL name will also be shown.

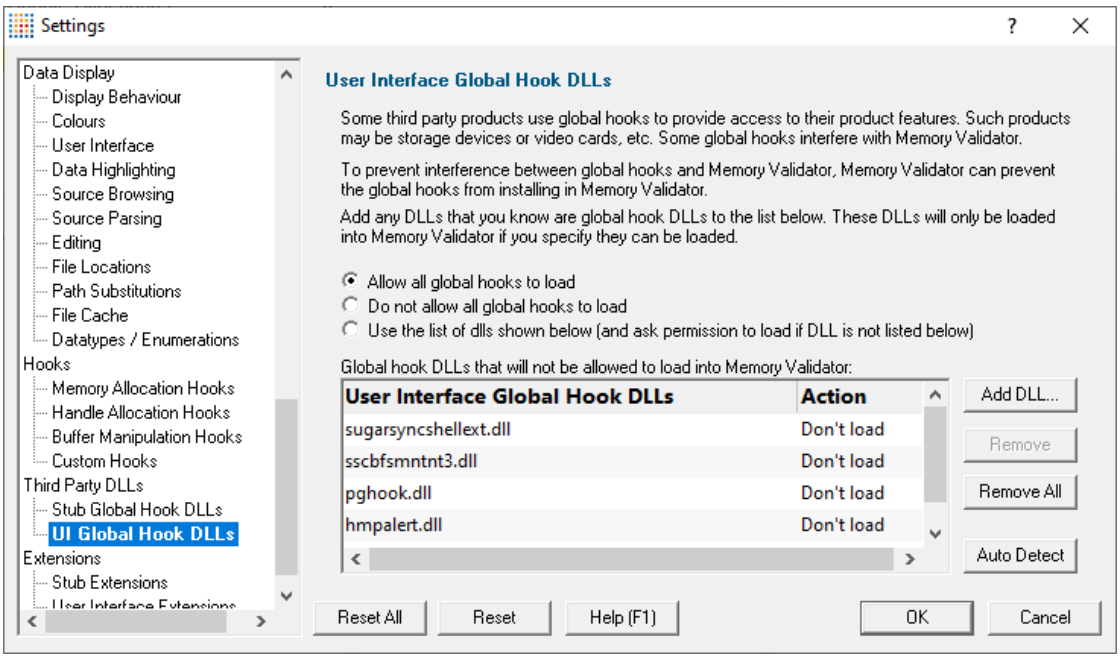


Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.9.2 User Interface Global Hook DLLs

The **User Interface Global Hook DLLs** tab allows you to detect and specify global hook DLLs that may not be wanted in the Memory Validator user interface process.



About global hook DLLS

See the similar topic on stub global hooks to read about global hook DLLs

The user interface hook DLL loading rule

The default behaviour is not to allow the global hooks to load, but you can change this if necessary

- **Allow all global hooks to load** › allows all global hook DLLs to load into Memory Validator
- **Do not allow any global hooks to load** › prevent any global hook DLLs from loading (the default)
- **Use the list of dlls shown** › provide per-DLL control over which DLLs load or don't load via the **User Interface Global Hook DLLs** list

  Any global hook DLLs not listed will result in the user being asked for permission to load a DLL via the Global Hook Warning Dialog below

Managing user interface global hook DLLs

- **Add DLL...** › browse and select one or more DLLs › **Open** › adds the chosen DLLs to the **Global Hook DLLs** list

  Having added a DLL to the list, you can change whether the DLL is allowed to load or not, by double clicking in the second column and changing the value: **Load** or **Don't load**
- **Remove** › removes any selected DLL from the list
- **Remove All** › removes all DLLs from the list

Auto detecting global hook DLLs

Memory Validator can detect any DLLs in its own process that are not ones it uses itself. Such DLLs are likely to be global hook DLLs:

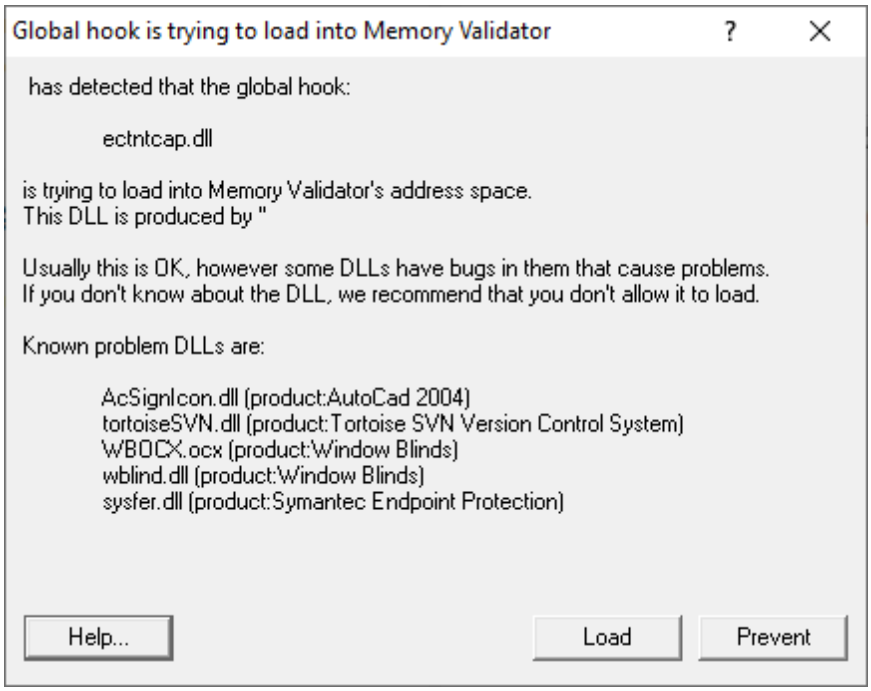
- **Auto Detect** › automatically detect DLLs which may be global hook DLLs, *adding* them to the **Global Hook DLLs** list

Global Hook Warning Dialog

When the global hook loading rule above is set to **Use the list of dlls shown**, the **Allow load** column controls whether the hook DLL is loaded.

When a global hook is loaded that is *not* on the list of known global hooks, the user is presented with a warning dialog like that shown below.

The user can then accept or block the global hook from loading. The dialog lists a couple of known problematic DLLs.



- **Help** > displays this help page
- **Yes** > lets the DLL load
- **No** > blocks the DLL

The response is automatically recorded in the **Global Hook DLLs** list, so that you won't be asked again.

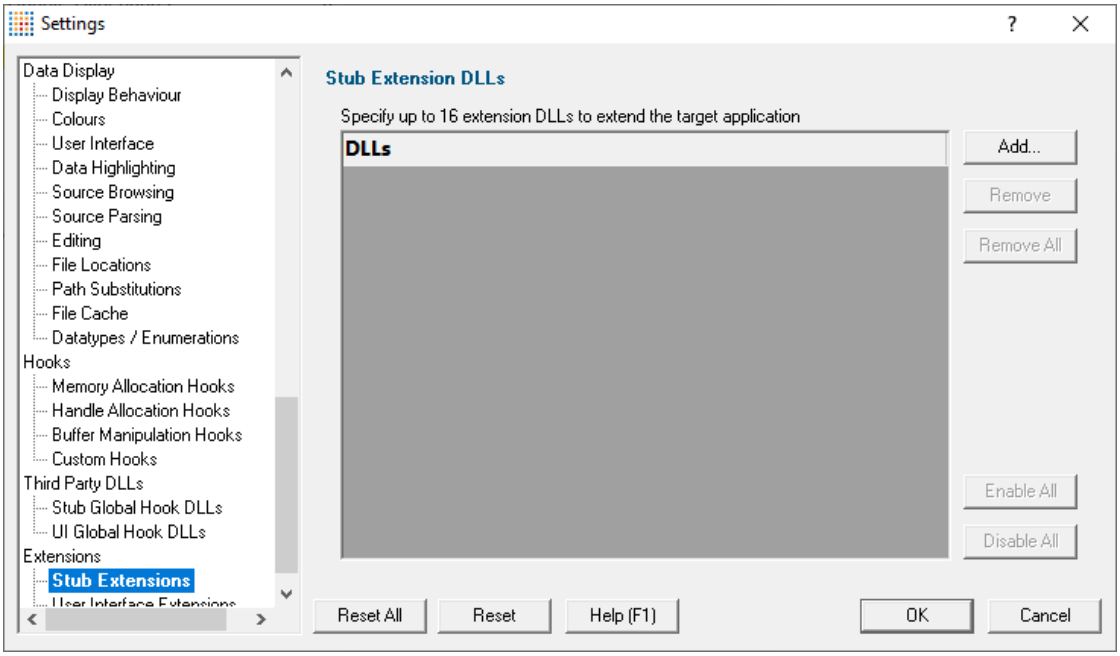
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.12.1.10 Extensions

3.12.1.10.1 Stub Extensions

The **Stub Extensions** tab allows you to specify DLLs which can be used to extend the Memory Validator stub.



Stub extension DLLs

You can specify up to 16 extension DLLs which can be used to extend the functionality of the Memory Validator stub - i.e. the target application.

Standard options control the list:

- **Add...** › browse and select one or more DLLs › **Open** › adds the chosen DLLs to the **DLLs** list
- **Remove** › removes any selected DLL from the list
- **Remove All** › removes all DLLs from the list

Enable or disable a DLL by ticking or unticking the yellow boxes or change them all at once

- **Enable All** › ticks all the DLLs, enabling them all
- **Disable All** › unticks everything

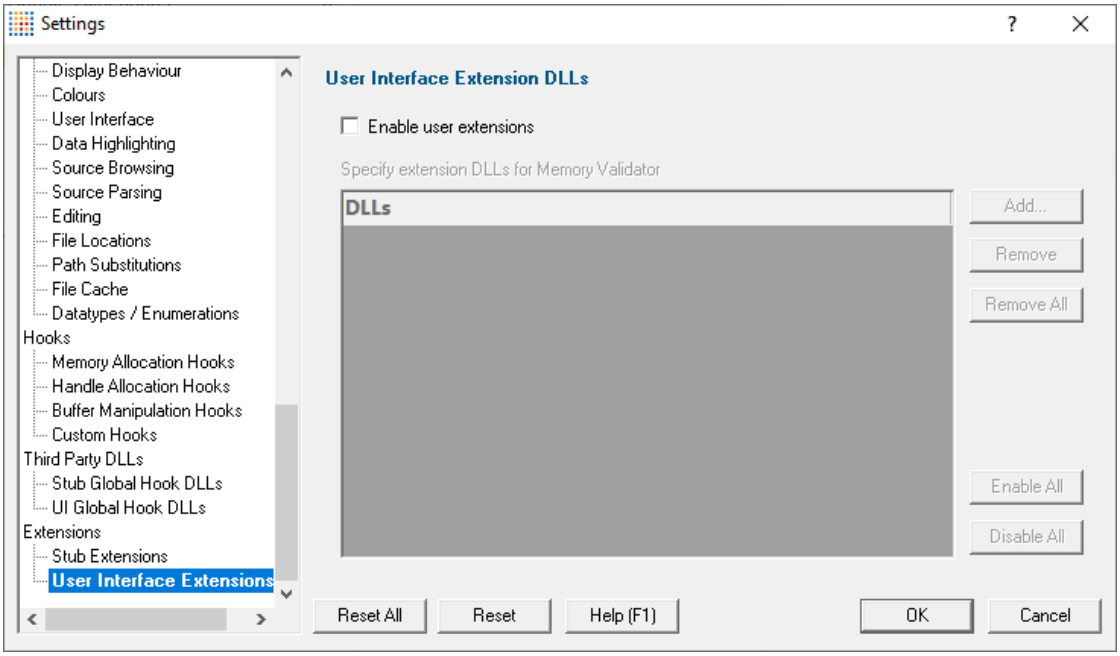
➡ See also user interface extension DLLS

Reset All - Resets **most** global settings including those on other pages, but does not clear the DLLs in this list other than those added since the settings page was last opened.

Reset - Resets the settings on the current page.

3.12.1.10.2 User Interface Extensions

The **User Interface Extensions** tab allows you to specify DLLs which can be used to extend the Memory Validator user interface.



User interface extension DLLs

Similar to the stub extensions, but without the limit of 16, you can specify extension DLLs which can be used to extend the functionality of the Memory Validator user interface.

- **Enable user extensions...** › allows you to turn your extensions on/off without having to enable/disable or remove them all.

Standard options control the list:

- **Add...** › browse and select one or more DLLs › **Open** › adds the chosen DLLs to the **DLLs** list
- **Remove** › removes any selected DLL from the list
- **Remove All** › removes all DLLs from the list

Enable or disable a DLL by ticking or unticking the yellow boxes or change them all at once

- **Enable All** › ticks all the DLLs, enabling them all
- **Disable All** › unticks everything

➜ See also stub extension DLLs

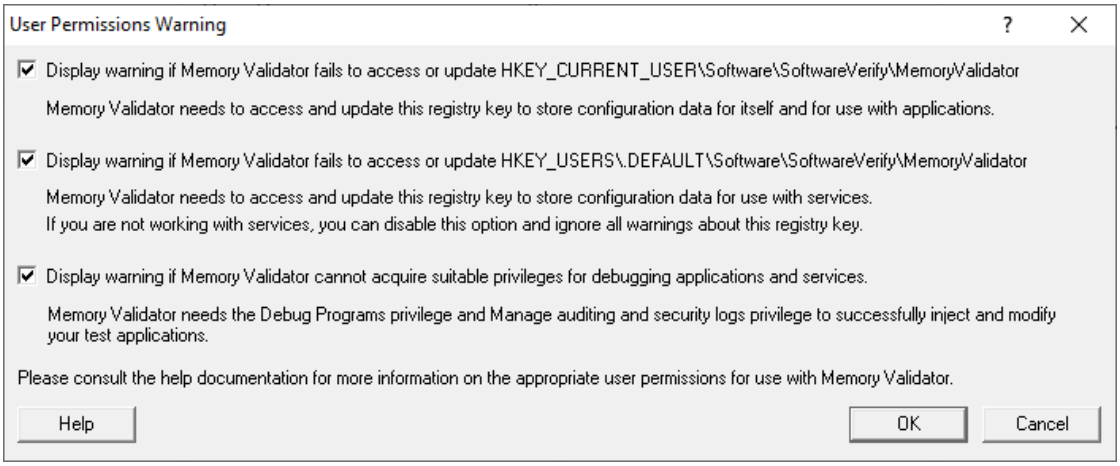
Reset All - Resets **most** global settings including those on other pages, but does not clear the DLLs in this list other than those added since the settings page was last opened.

Reset - Resets the settings on the current page.

3.12.2 User Permissions Warnings

You may see (or want to see) warning dialogs when Memory Validator receives an error accessing the registry or obtaining debugging privileges.

Settings menu › **User Permissions Warnings...** › shows the User Permissions Warnings dialog below



The Help button displays the User Permissions help topic.

➜ See also, the answer to the question about creating Power User accounts on Windows XP.

3.12.3 Ordinal Handling

Ordinal values and .def files

Some DLLs, including some from Microsoft, may export their functions by ordinal value, instead of by the usual readable name.

However, having access to the module's original .def file means those ordinal values can be used to look up the symbol names to display in Memory Validator.

The .def file will contain the function names and ordinal values, allowing a DLL's exported ordinal value to be mapped to the symbol name.

For example, here's a small section of mfc90.def showing the ordinal values 332 to 335 and a selection of decorated names:

```
??0CBrush@@QEAA@K@Z @ 332 NONAME
??0CBrush@@QEAA@PEAVCBitmap@@@Z @ 333 NONAME
??0CByteArray@@QEAA@XZ @ 334 NONAME
??0CChevronOwnerDrawMenu@@QEAA@XZ @ 335 NONAME
```

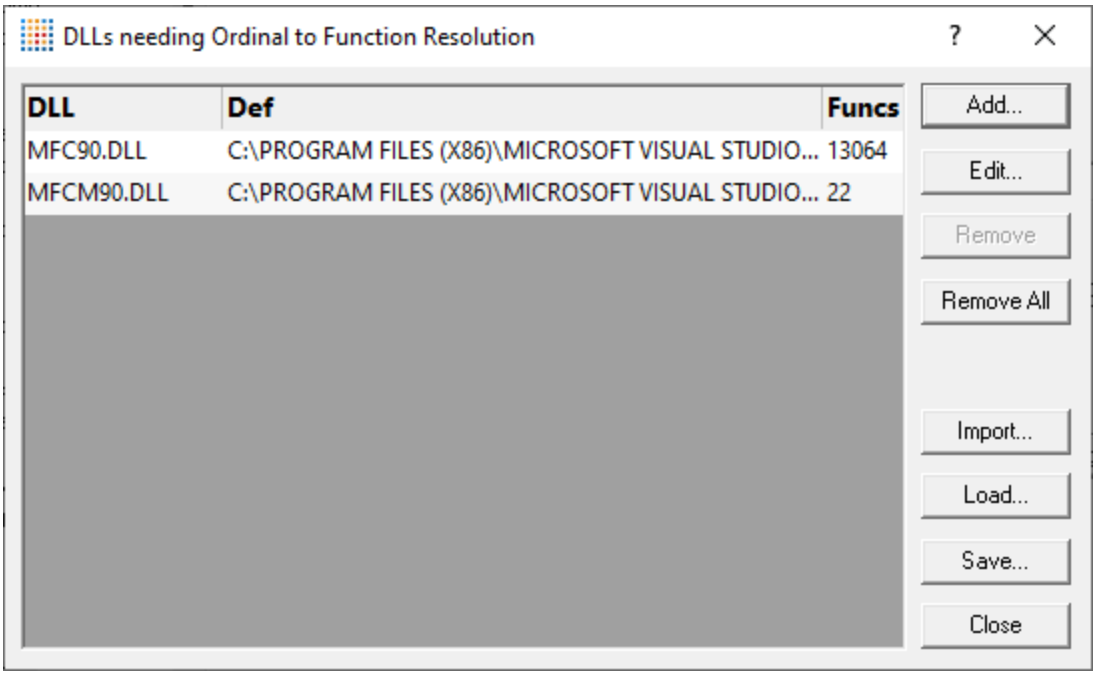
You can use the ordinal handler dialog (below) to associate a def file with a DLL. These associations will persist between sessions.

If you need different ordinal configurations for different DLL usage, you can export this mapping between DLL and .def to a file, to be used at a later date.

The ordinal handler dialog

The ordinal handler dialog lets you manage which .def files are associated with which DLLs.

Settings menu › **Edit Settings...** › **Symbols** › **Symbols Misc** page › **Manage Ordinals** › shows the ordinal handler dialog



- **Add...** › shows the ordinal-to-function converter dialog, described below ,so you can add a new mapping
- **Edit...** › opens a selected mapping in the ordinal-to-function converter dialog

Double clicking an item in the list also does this.

- **Remove** › removes the selected associations from the list

- **Remove All** › clears all the associations in the list

- **Import...** › choose a previously saved set of associations to ***add*** to the list
- **Load...** › as for Import, but ***replaces*** the contents of the list
- **Save...** › saves the associations in the list to a .ord file of your choice

Switching usage of mapped names on and off

The option to switch the ordinal mapping on or off is in the global settings dialog:

Settings menu › **Edit Settings...** › **Advanced** › **Symbols Misc** page › **Convert DLL exported function ordinals to symbols** › enable the ordinal to function name mapping

You'll need to tick this to enable the use of mapped names defined in the list above. If you don't, you won't see the names being used.

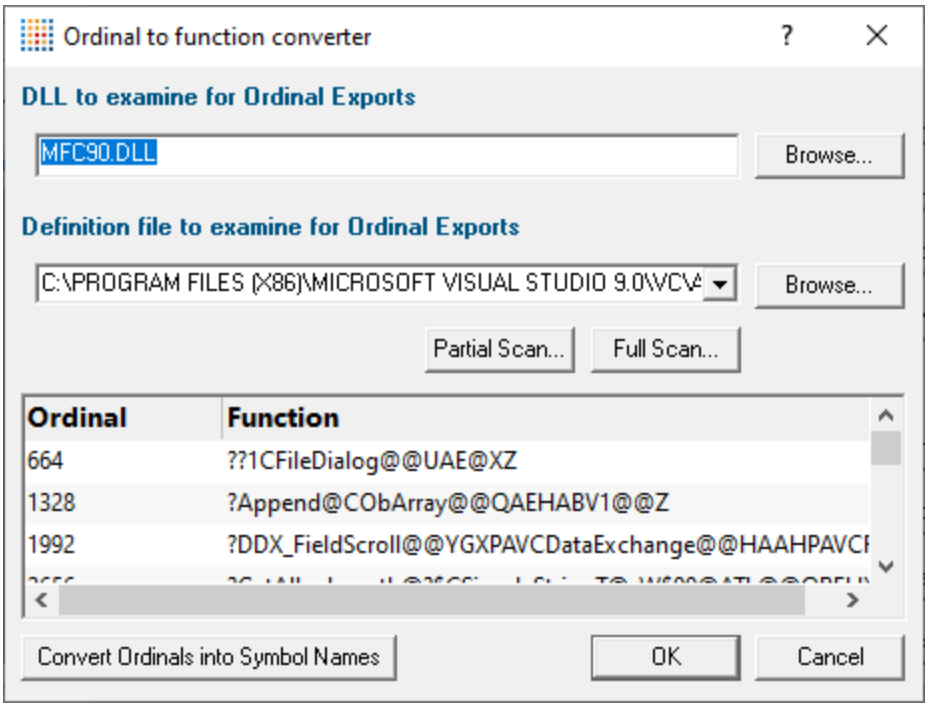
The ordinal-to-function converter dialog

After clicking Add... on the Ordinal Handler Dialog (above) you'll see the ordinal-to-function converter dialog below.

The basic process for adding a new mapping is to:

- choose a DLL
- find its matching .def file
- convert the ordinal values to symbol names
- add the file association to the Ordinal Handler dialog

This example shows the Microsoft MFC90.dll associated with its .def file:



Associating the DLL and .def files

In the ordinal-to-function converter dialog:

- **DLL to examine...** > type or browse to enter the DLL you want to map
- **Definition file to examine...** > type or browse to enter the .def file you want to associate with the DLL, or choose one from the drop-down list

  If a .def file matching the name of the .dll exists in the same directory as the DLL, this will be set automatically.

- **Convert Ordinals into Symbol Names** > convert the exported ordinals from the chosen DLL into symbol names using the specified .def file

  All the ordinals and symbol names found will be displayed in the list at the bottom.

- **OK** > closes the dialog and adds the association between the DLL and the .def to the ordinal handler dialog

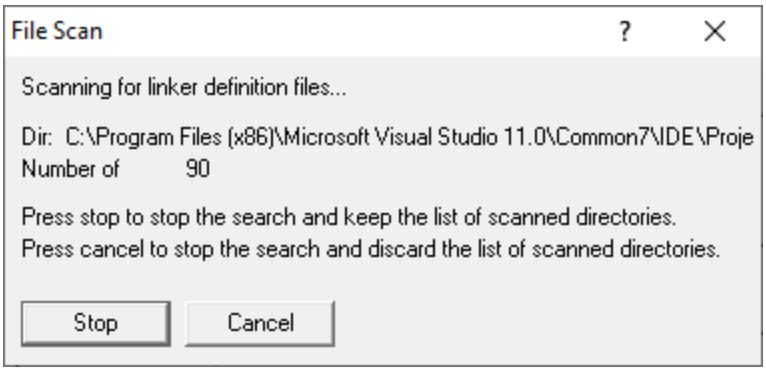
Scanning for .def files

The .def files aren't always in the same directory as the DLLs and may be hard to find, so a search option is available.

Any .def files found during a scan can be used to extend or replace the drop-down list of the **Definition file to examine...** option above.

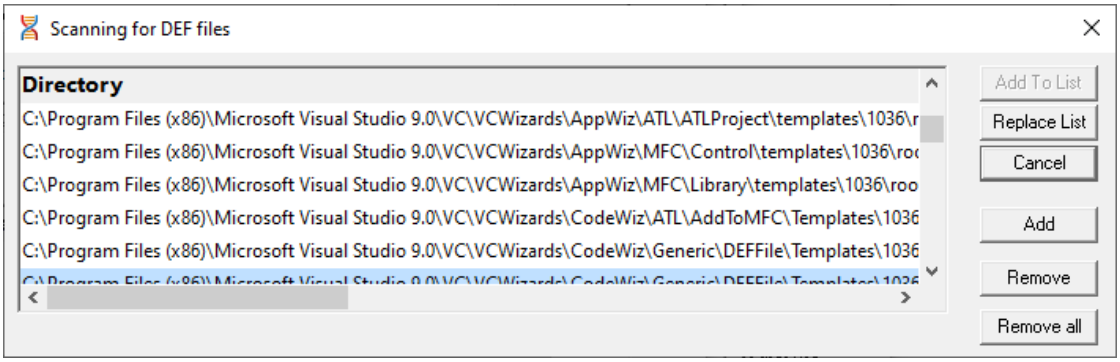
- **Partial Scan...** › choose a folder and scan it for .def files
- **Full Scan...** › scan all drives for .def files

While scanning, a progress dialog shows the search location and the number of .def files found:



- **Stop** › stop the scan and show results found so far
- **Cancel** › stop the scan and discard any results

When the scan is complete a dialog shows any .def files found:



- **Add To List** › extend the drop-down list of the **Definition file to examine...** option by adding all the search results in the list
- **Replace List** › as above but replaces the list rather than extending it
- **Add** › adds an entry to the list so you can manually enter a file path
- **Remove** › remove any selected items from the list
- **Remove All** › clears the list
- **Cancel** › closes the dialog, discarding the list of results

3.12.4 Loading and saving settings

Saving and loading settings files

Memory Validator settings can be saved to a file and restored at any time.

Settings menu › **Save Settings...** › save settings to a file

Settings menu › **Load Settings...** › load a previously saved settings file

Loading settings via command line option

Different settings files can be used for automated regression testing by using the -settings command line option and the required filepath to load the settings when Memory Validator starts up.

3.13 Managers

The Managers menu provides a handful of powerful tools to manage or inspect data collected by Memory Validator.

The tools include:

- session management
- global, session, local and thread filters
- named heaps
- watermarks and bookmarks

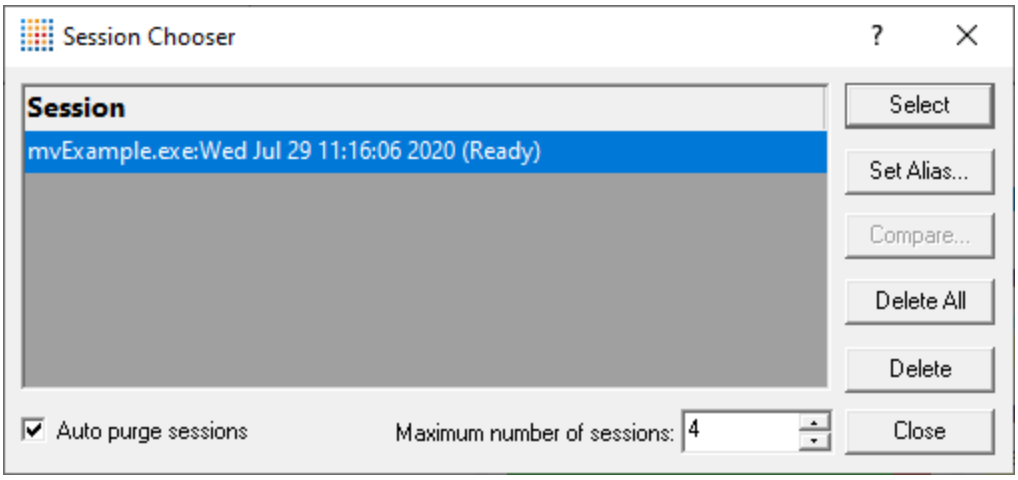
3.13.1 Session Manager

Managing multiple sessions

Memory Validator can manage multiple sessions at once.

As well as the actively running session, open sessions may include those run since Memory Validator started, or sessions that have been saved earlier and reloaded.

Managers menu › **Session Manager...** › shows the Session Chooser dialog below, highlighting the current session



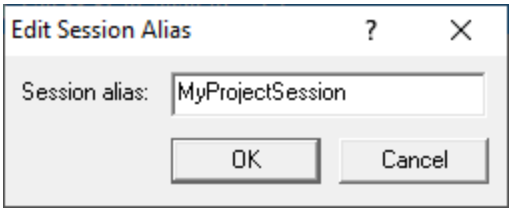
Each time a session is started or loaded it is added to this list using the name of the executable program and the date and time the session started.

Managing the sessions

- **Select** › makes the selected entry the *current* session, i.e. the one for which data will be displayed

  Some tab views may update immediately, others may need a manual refresh

- **Set Alias...** › opens the Edit Session Alias dialog so you can give the session a more useful name



- **Delete** › removes the selected session

  You can't delete a session that is actively collecting data.

- **Delete All** › removes all the loaded sessions

  If one of the session is actively collecting data, this will be disabled.

- **Close** › closes the dialog (as opposed to closing any selected sessions!)

Comparing loaded sessions

When two different sessions are loaded they can be compared as part of a manual regression test.

- **Compare...** › shows the Compare Sessions dialog for manual regression testing.

Limiting the number of sessions

You can choose to limit the maximum number of sessions open at once, at which point each time a new session is added, the oldest session will automatically be removed.

- **Auto purge sessions** › ensures that the number of loaded sessions is limited to the maximum (below)
- **Maximum number of sessions** › sets the maximum number of sessions allowed if auto-purge is on

3.13.2 Filters

Filtering of data

Memory Validator has a powerful filtering mechanism that allows you to exclude unwanted data from the displays.

Filtering happens at different levels of granularity and the filters themselves can be defined to be broad or extremely targeted.

You can manage different groups of filters; save and load them, and move filters between groups;

Filters do not control what is collected, only what is displayed, so you can refocus on different areas of data at any time.

Filter Groups

Filter groups allow related filters to be grouped together.

For example, you might have a group of filters excluding data relating to memory leaks in 3rd party products that you have no control over.

The Thread Filter manager gives you control over which thread data you see.

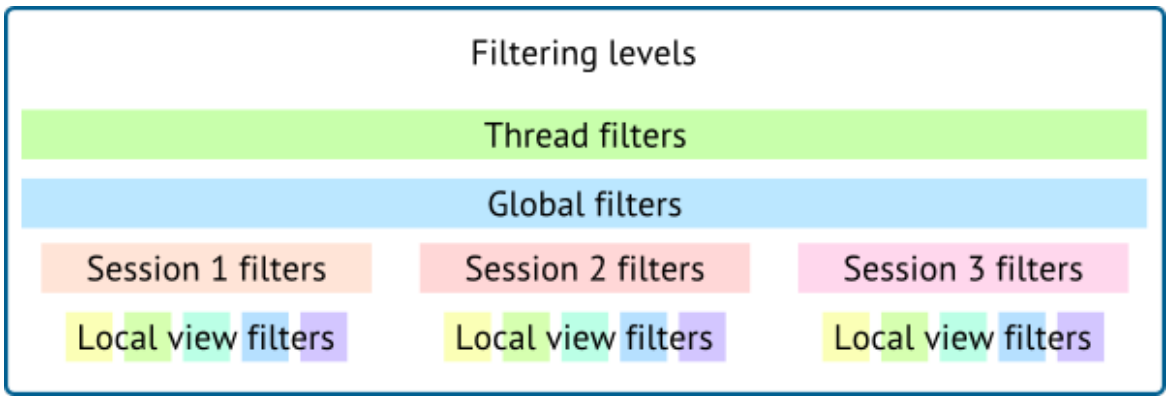
The Filter Manager dialog lets you manage the high level global filters and session filters

For local filters, the relevant tab views have a Filter... button to the left of each display.

Filtering Level

Filtering of displayed data happens at four different levels:

- **Thread filters** where for example you might enable only the UI thread of your application
- **Global filters** that will be applied to all sessions and all tab views
- **Session filters** are applied only to a particular session for which data is recorded and displayed
- **Local filters** affecting each individual tab view



Filter Types

Filters can be one of three types:

- **Instant filters** are derived directly from an existing callstack and saved with the session

  Typically these are created by right clicking on a callstack in the display and opting to create an instant filter based on the selected data.

- **Temporary filters** are usually instant (callstack) filters, and will not be saved anywhere
- **Custom filters** let you decide exactly what to exclude using the Define Filter dialog

Any persistent filter can be made temporary and vice versa.

While you'd normally make custom filters persistent, you *can* make it temporary if you wish.

3.13.2.1 Thread Filters

Thread filters

The thread level is the highest level at which filtering happens.

Thread filtering simply allows you to exclude data according to the thread id.

Filtering by thread id is only going to be useful for multi-threaded applications.

Whether you realise it or not, many applications have threads that you may want to exclude from your data as being beyond your control.

 The thread filter dialog below also serves the dual purpose of manually naming threads. The names are used elsewhere in the display of data and selection of threads.

The thread filter dialog

 **Managers** menu › **Thread Filter...** › shows the Filter Memory by Thread Id dialog

The dialog is very simple - it just has a list of all the threads in the target program, with check boxes to enable and disable the display of data from each thread id.

The dialog has three sections. A thread selector at the top, a thread grid in the middle, and at the bottom a control that allows you to choose how the thread filter shows or hides threads.

Thread Selector

The thread selector lists are threads in the target application plus two special values: **All Threads** and **Specific Threads**.

When a thread is selected that thread is the thread that the thread filter works on.

When **All Threads** is selected, the thread filter works on all threads.

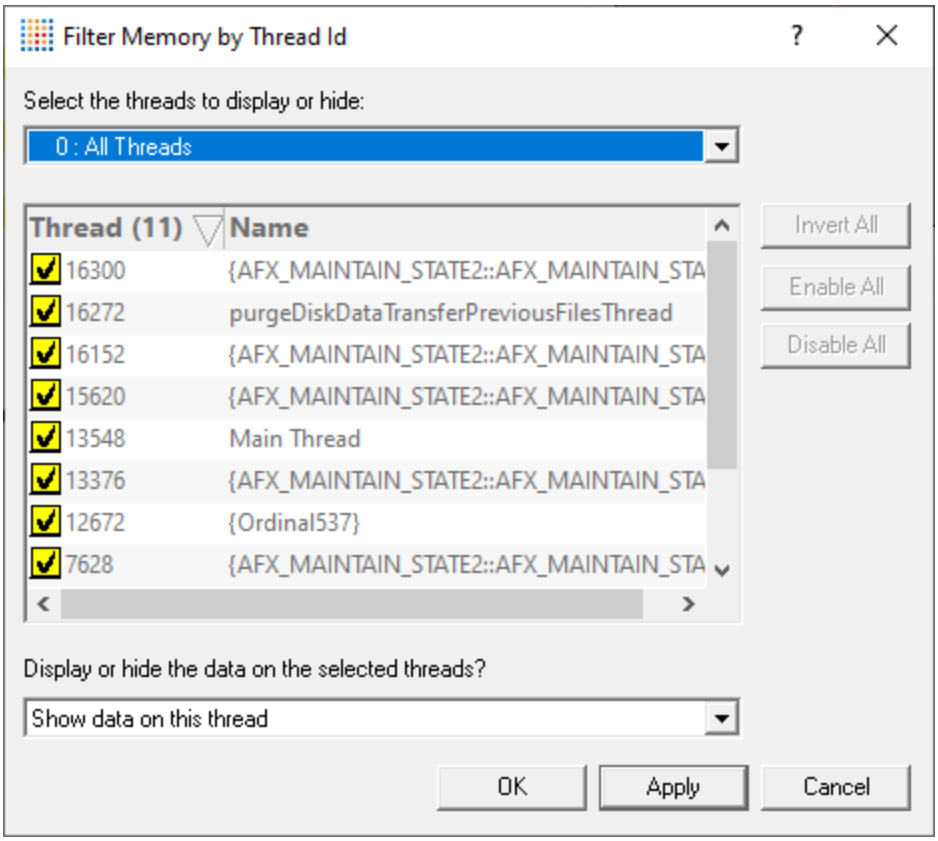
When **Specific Threads** is selected, the thread filter works on the enabled threads in the thread grid.

Thread grid

If the thread selector is set to **Specific Threads** the grid and the controls next to it are enabled.

The Specific Threads option allows you to choose one or more threads to filter, giving you more flexibility that the other options on the thread selector.

The thread grid is very simple - it just has a list of all the threads in the target program, with check boxes to enable and disable the display of data from each thread id.



- **Invert All** › toggle the enabled/disabled state for each thread
- **Enable All** › checks all threads, meaning that no thread filtering will occur
- **Disable All** › unchecks all threads, making it easier to enable just one or two of many threads

Sorting

To sort the list on id or name, click one of the headers.

Showing (or hiding) filtered results

The filters can be set to hide data that matches any of the filters, or to show data that matches any of the filters. The default is to hide data that matches any of the filters.

- **Hide data on this thread** › if selected, sets the filter to *hide* matching threads from display, rather than *displaying* it
- **Show data on this thread** › if selected, sets the filter to *only display* matching threads to display, rather than *hiding* it

Filters that show data are perfect for showing very focused and targeted results.

Filters that show data can be more computationally expensive, so use them with caution!

Thread names

If a thread has been named using the Win32 `RaiseException` method, the Win32 `SetThreadDescription()`, or using `mvSetThreadName()` its name is shown in the list. See the link below for more details.

For threads not explicitly named by the above methods, Memory Validator provides automatically generated names based on the name of the function passed to `CreateThread()`, `_beginthread()`, or `_beginthreadex()`. If you want to give a thread a name here by double clicking on the name column and entering a name for the thread. Click outside the box or press return to complete the entry.

Names of threads will be used where relevant in other parts of the application. Eg thread selection in the Types and Sizes tabs.

➜ How can I give a name to a thread from my code?

3.13.2.2 Global Filters and per-Session Filters.

Global and session filters

Global filters affect all the relevant tab views for all sessions.

Session filters also affect all the tab views, but only for the that session.

Both global and session filters are set up in the Filter Manager dialog.

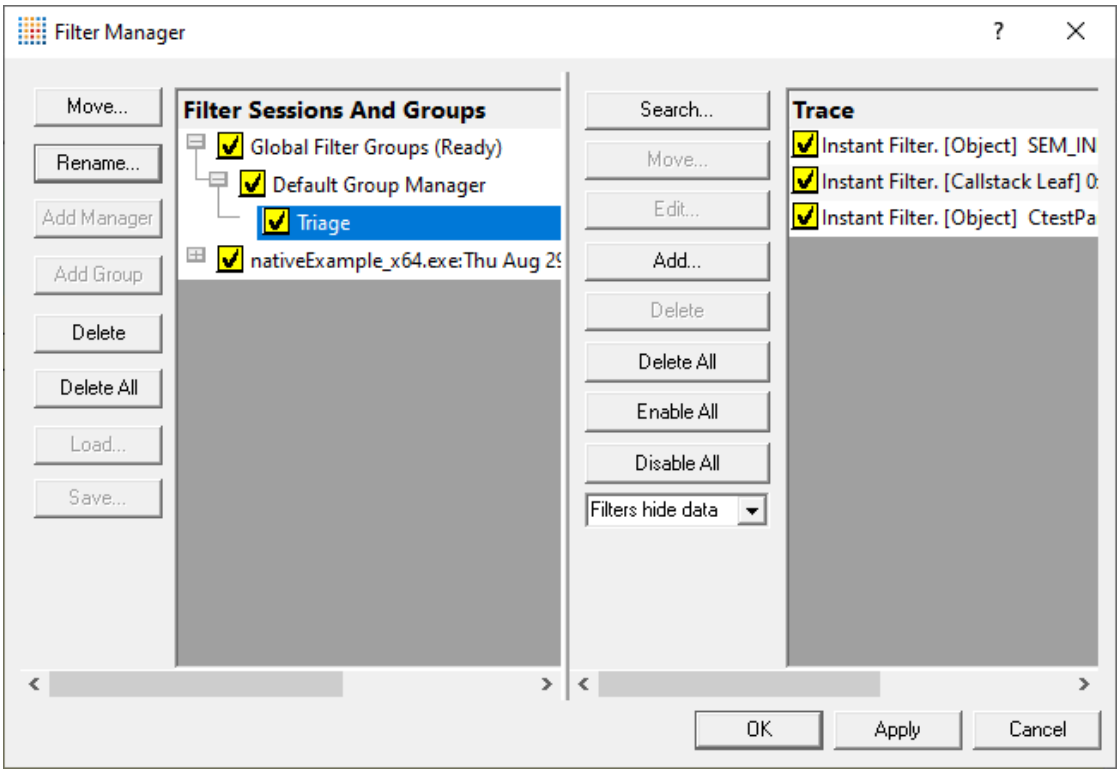
Opening the filter manager dialog

Managers menu › **Global Filter Manager...** › shows the Filter Manager dialog

Or use this option on the Session Toolbar:

The filter manager dialog

The filter manger has two main components: on the left you can manage the groups of filters, and on the right, the filters themselves.



Filters are arranged in the following hierarchy:

- Global and session filter groups
  - Filter group manager
    - Filter groups
      - Filters

Filters are displayed on the right when a filter group is selected, as shown in the image above.

Like the global filters, each session has a default filter group manager under which filters are placed, unless you create your own group structure.

Building the filter group hierarchy

Building a complex or hierarchy is not essential - you can just have one filter group and manager and work perfectly well.

However, if you make regular use of this feature, you may wish to fine-tune how you use different groups of filters.

Having created the hierarchy and added some filters, you can move filters and groups of filters around and reuse them.

To build the hierarchy, different actions are available depending on each selected item:

Select a global or session filter groups:

- **Add Manager** › creates a new filter group manager
- **Save...** › save all the filters and filter groups for the selected global or session filter group to a .mvf file
- **Load...** › load filters and groups from a previously saved .mvf file, replacing any existing content

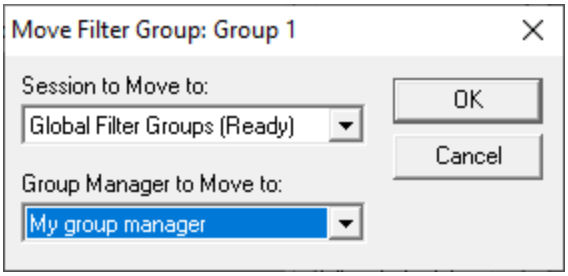
Select a filter group manager:

- **Add Group** › creates a new filter group
- **Rename...** › enter a new name for this filter group manager
- **Delete** › remove this filter group manager from its global/session filter group

Select a filter group:

- **Rename...** › enter a new name for this filter group
- **Delete** › remove this group from its filter group manager
- **Move...** › opens the Move Filter Group dialog below so you can move the group to a new owner

  First choose a global or session filter group, and then which of its managers you want to move the filter group to.



- **Delete All** › remove the entire hierarchy and all its filters

  There's a confirmation dialog in case you click this accidentally!

Adding filters to a group

In the right hand panel, is a list of filters, belonging to the filter group you selected on the left. Initially empty, this is where you add your filters.

The functionality here is almost identical to the Local Filters dialog, with filters being added and modified via the the Filter Definition dialog.

Turning a filter group on / off

Filter Group Managers and Filter Groups can be enabled or disabled by clicking the check box in the tree.

A disabled filter group manager will not take part in filtering any data.

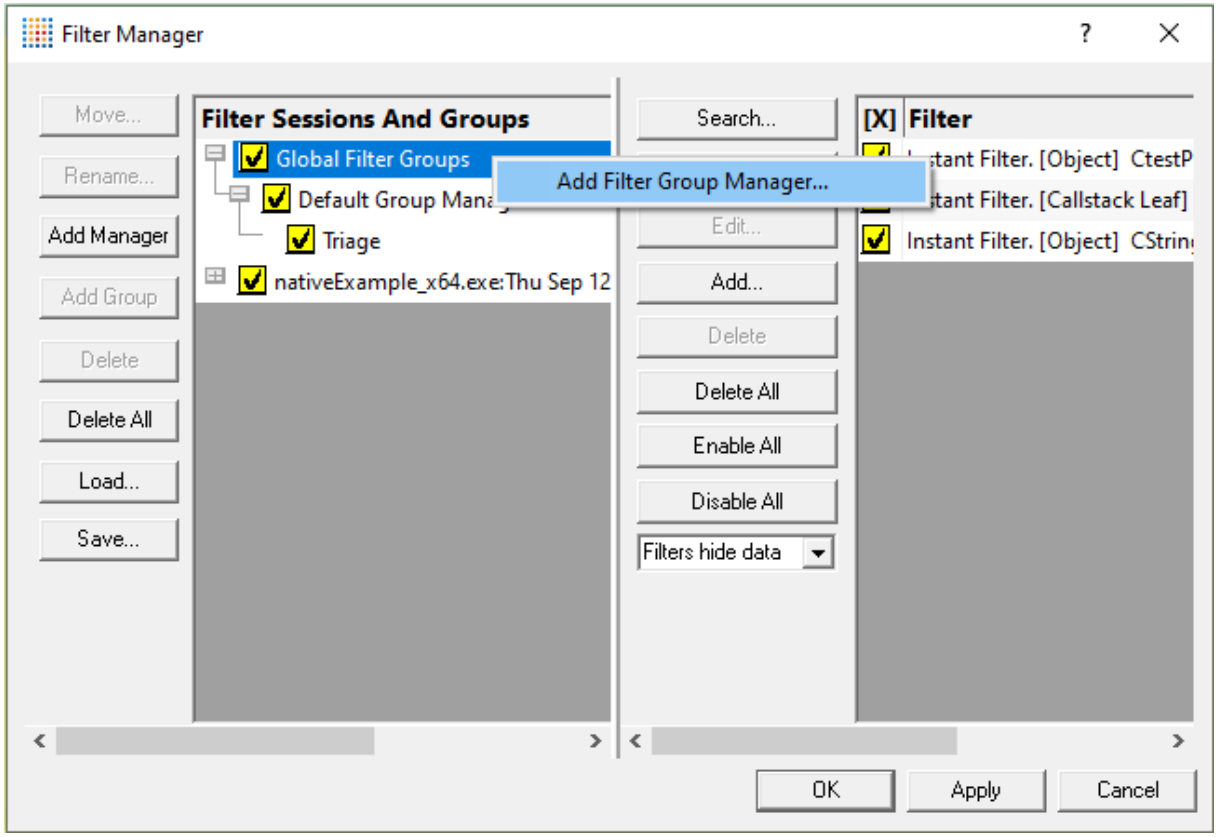
A disabled filter group will not take part in filtering any data.

To see the effects of changing the enabled status you need to click the **Apply** button or the **OK** button.

Context menus

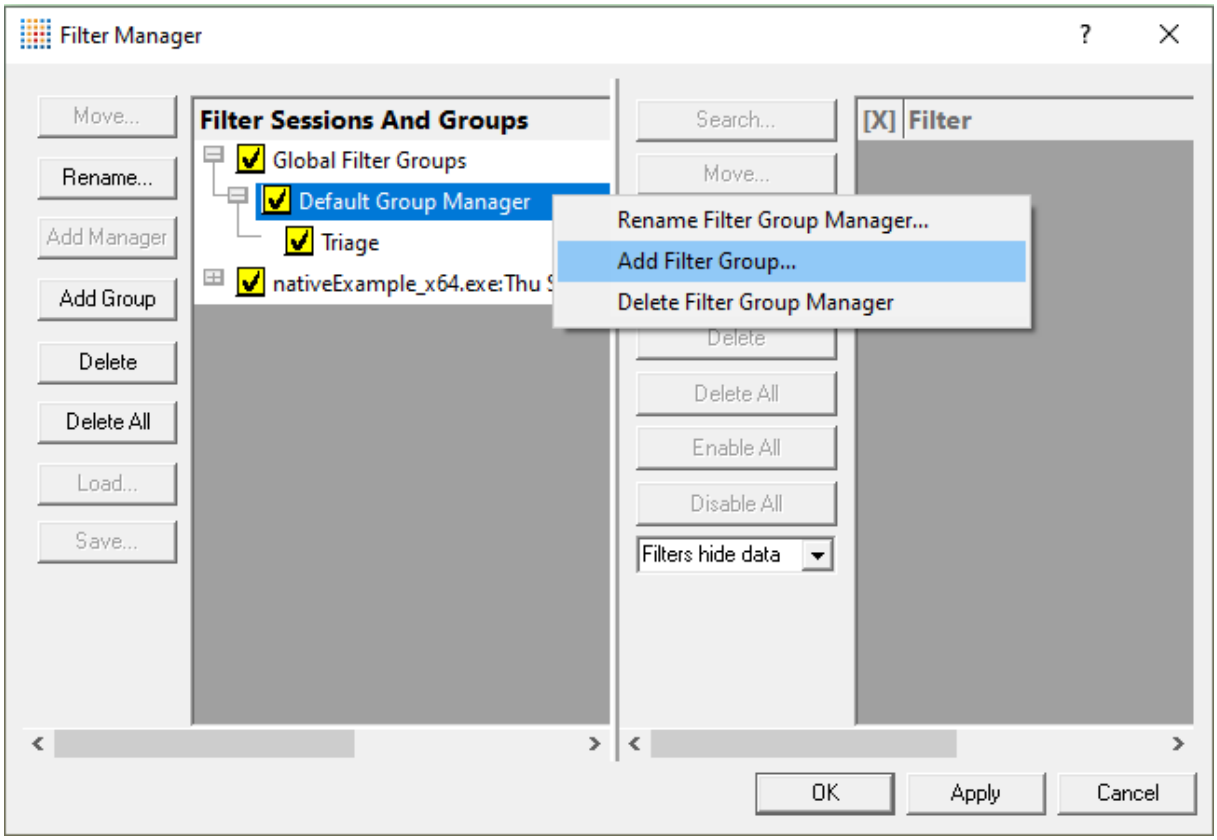
Items in the left hand part of the display have context menus that provide access to functions relevant to the item selected.

Global Filters and Session Filters



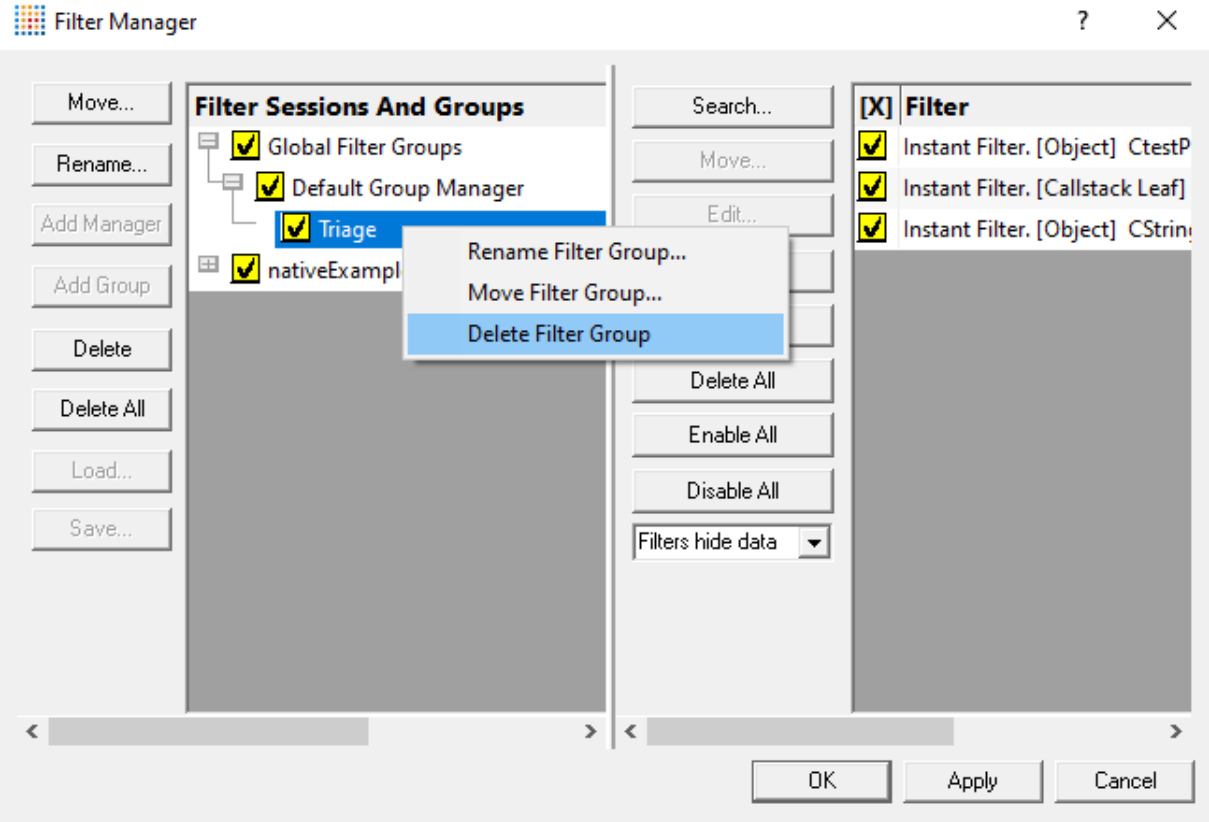
- **Add Filter Group Manager...** › creates a new filter group manager

Filter Group Managers



- **Rename Filter Group Manager...** › enter a new name for this filter group manager
- **Add Filter Group...** › creates a new filter group
- **Delete Filter Group Manager** › remove this filter group manager from its global / session filter group

Filter Groups



- **Rename Filter Group...** › enter a new name for this filter group
- **Move Filter Group...** › move this filter group to a new filter group manager
- **Delete Filter Group** › remove this filter group from its filter group manager

3.13.2.3 Local Filters

Local Filters

Local filters let you remove unwanted noise from the data views, or when *inverted*, can target very specific data you want to display.

Being local means they apply only to the tab view where you create them, without affecting other data elsewhere in Memory Validator

The following tabs have local filters:

- Memory tab
- Types tab
- Coverage tab
- Analysis tab
- Pages tab

While most of the local filters use the more detailed FIlter Definition dialog to fine-tune the filters, the Types and Coverage tab use dialogs with much simpler filtering rules.

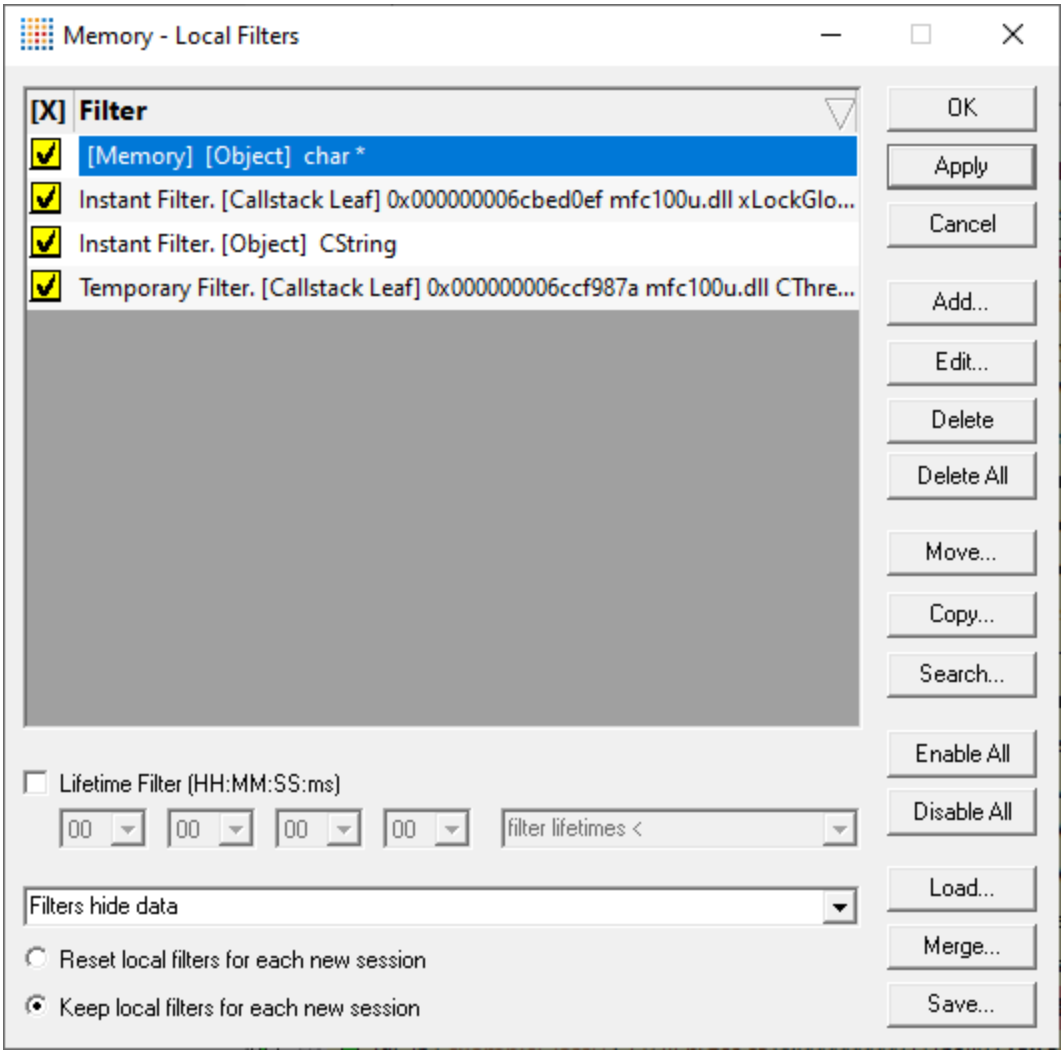
The Coverage tab filter dialog can also be accessed on the memory coverage page of the global settings dialog, where it is described in detail.

In the Types tab, local filters are applied to each of the five local object tabs (Thread, DLL, etc). They do not affect the *values* in each column, only whether the filtered object type is actually shown in the view.

The local filters dialog

This dialog lets you manage a group of local filters by adding new filters, modifying existing ones and enabling or removing them as required.

This example shows instant, temporary and custom filters:



The local filters dialog for the Types and Coverage do not have the Move or Search options.

Local filter management options

- **OK** › updates the data view with any newly changed local filters and closes the dialog
- **Apply** › updates the data view with any newly changed local filters and without closing the dialog
- **Cancel** › discards and changes to filters and closes the dialog

- **Add...** › shows a dialog such as the Filter Definition dialog to define a new filter to add to the list
- **Edit...** › reopens the definition dialog to view or modify the selected filter

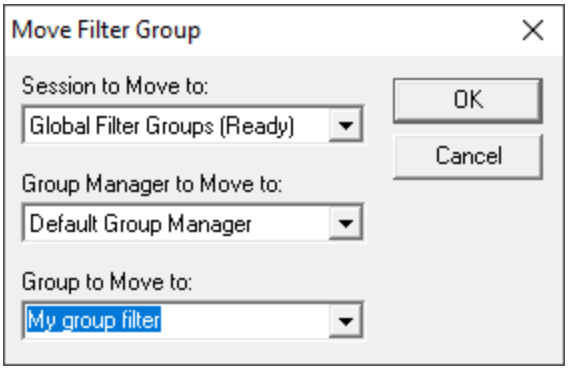
  Double clicking a filter in the list also works.
- **Delete** › removes the selected filters from the list
- **Delete All** › clears the list completely

The next three options are only found on the Memory, Analysis and Pages local filter dialogs:

- **Move...** › opens the Move Filter Group dialog below so you can move the filter to a group elsewhere, thus not making it local any more.

  First choose a global or session filter group, and then which of its managers and groups you want to move the filter to.

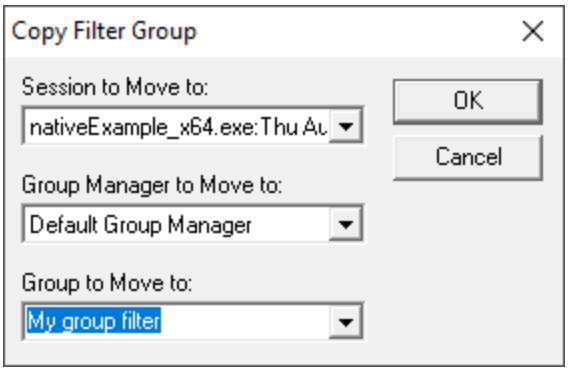
  If you don't have a target group yet, you can type a new group name or select NewGroup from the list.



- **Copy...** › opens the Copy Filter Group dialog below so you can copy the filter to a group elsewhere, thus not making it local any more.

  First choose a global or session filter group, and then which of its managers and groups you want to move the filter to.

  If you don't have a target group yet, you can type a new group name or select NewGroup from the list.



- **Search...** > shows the Find Filter dialog to look for a filter in your list which matches an object type, size or file location

  Searching is only going to be useful when you have many filters in your group.

- **Enable/Disable All** > switches all the filters on or off

- **Load...** > load the filters from a local filters file, replacing the existing filters

- **Merge...** > load the filters from a local filters file, adding to the existing filters without causing duplicate filters.

- **Save...** > save the filters to a local filters file

Sorting Filters

Filters can be sorted by:

- **Enabled state** > filters are sorted by their enabled state. For any two filters that have the same enabled state they are sorted by the filter comment field

- **Comment** > filters are sorted by their comment field. For any two filters that have the same comment they are sorted by the enabled field

To sort filters click the appropriate column header. To change sort direction click the same column header again.

Lifetime Filter

The lifetime filter allows you to show or hide data that has existed for less than a specific time, or for longer than a specific time.

The time is specified in hours, minutes, seconds and milliseconds.

- **Lifetime Filter** › turn the lifetime filter on / off
- **HH:MM:SS:mm** › specify the time using the four combo boxes
- **Filter comparison** › specify if events are filtered because they have lifetimes shorter than the filter, or longer than the filter

Showing (or hiding) filtered results

The filters can be set to hide data that matches any of the filters, or to show data that matches any of the filters. The default is to hide data that matches any of the filters.

- **Filters hide data** › if selected, sets the filter to *hide* matching data from display, rather than *displaying* it
- **Filters show data** › if selected, sets the filter to *only display* matching data to display, rather than *hiding* it

  Filters that show data are perfect for showing very focused and targeted results.

  Filters that show data can be more computationally expensive, so use them with caution!

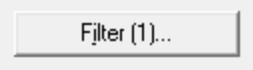
Remembering local filters between sessions

The local filters for each view can be reset or kept when a new session starts.

The default behaviour is to *keep* them between sessions, whether or not the executable changes

- **Reset local filters for each new session** › removes the local filters at the start of the next session
- **Keep local filters for each new session** › keeps the filters (the default)

When you start a session the number of local filters displayed on the Filter button on those tabs that have it.

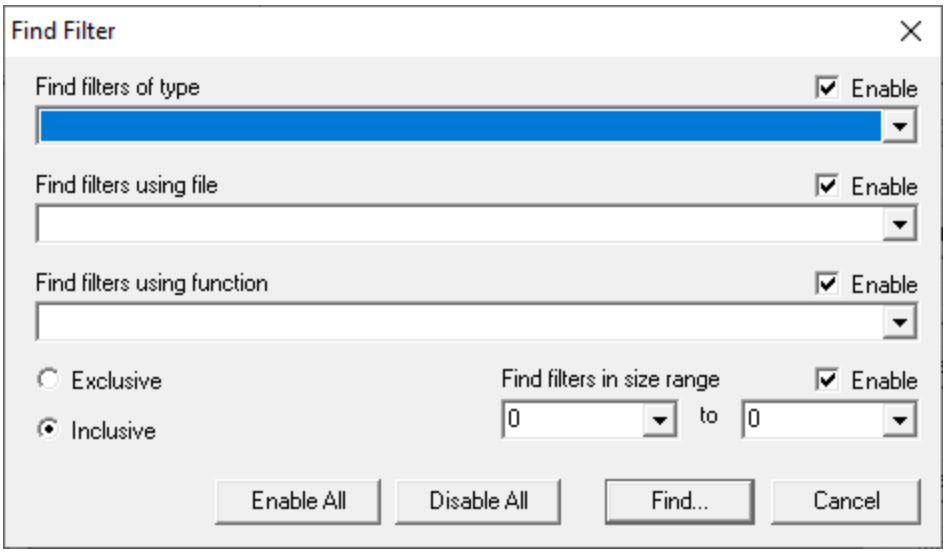


3.13.2.4 Find Filter

Sometimes when you have a long list of filters in a group, it can be awkward to find specific filters.

Whether you're using local filters or global or session filters, the Find Filter dialog (via the **Search...** button) can help with this.

The find filter dialog



There are four characteristics of the filters that you can search for, any or all of which can be enabled:

- **Find filters of type** › choose from a list of known datatypes for the target program in the current session
- **Find filters using file** › specify the source code file your target filter is using
- **Find filters using function** › target filters using a selected function name
- **Find filters in size range** › select a lower and upper object size limit by which to search filters

Each of these criteria will only show options relevant to the target program for the current session.

Exclusive or inclusive searches

The find filter dialog allows two types of search:

- **Exclusive** › *all* criteria that are enabled must match the filter for it to be highlighted

  Returns a tightly focused set of results.

- **Inclusive** › *at least one* of the enabled criteria must match

  Gives a broader set of results.

Performing a search

- Enable and set the criteria › choose **inclusive** or **exclusive** search › click **Find...** › matching filters are highlighted in the list

Filters are highlighted in the *selected object* colour, which is set via the Global settings › Data Display › Colours.

Delays with this dialog appearing? If this dialog takes a few seconds to appear, it's probably because you have a very large dataset, with many object types, files and functions.

3.13.2.5 Filter Definition

Filter definitions

Filter definitions consist of a set of criteria or rules by which you can exclude data from the various main tab views.

You can filter the data by many characteristics, including:

- Full or partial callstack matching
- Callstack symbol location details such as name, class, file, directory and DLL
- Memory and handle allocation criteria
- Object attributes such as size, type and address

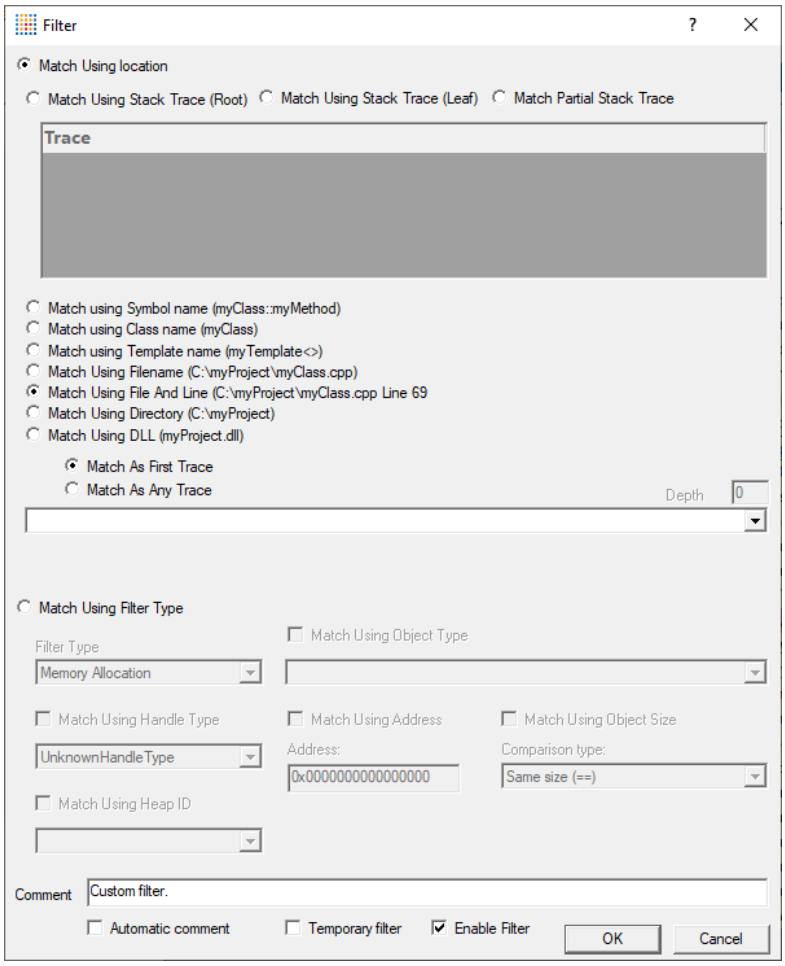
Global filters, session filters and local filters all use the filter definition dialog to create or modify a filter.

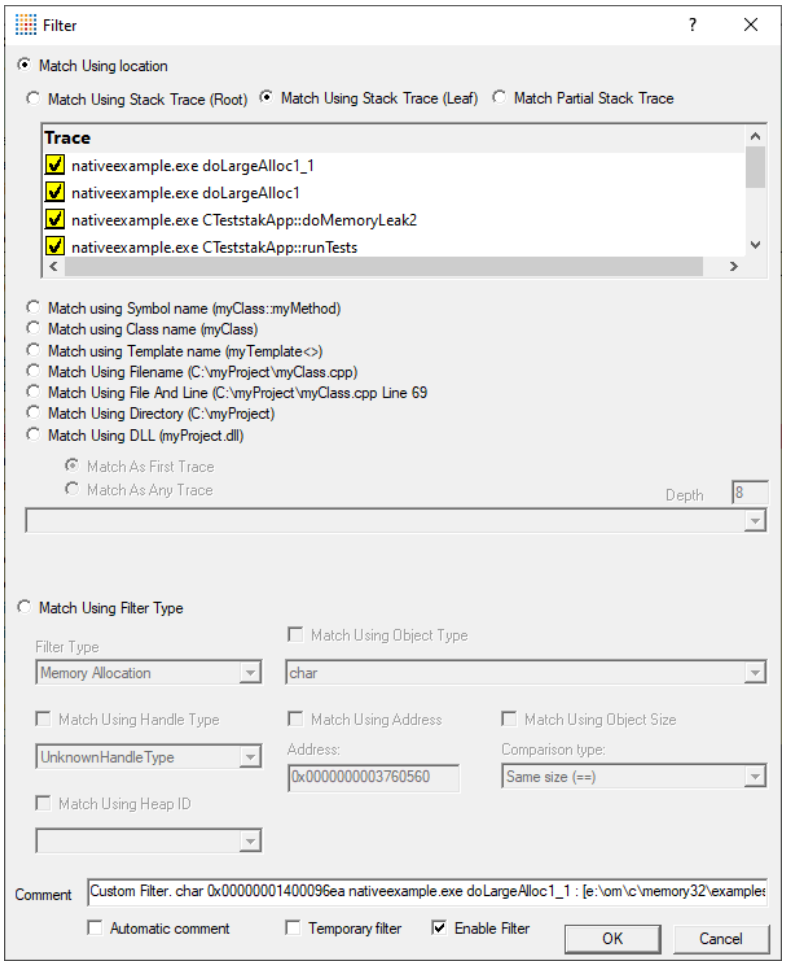
The filter definition dialog

The filter definition dialog appears as shown below left, when you first create a new local, session or global filter.

However, it's much easier to create and edit a filter directly from an existing data item displayed in the Memory or Analysis tab.

The example on the right below has pre-populated filter options for callstack, object type, size, address and other useful data to filter on.





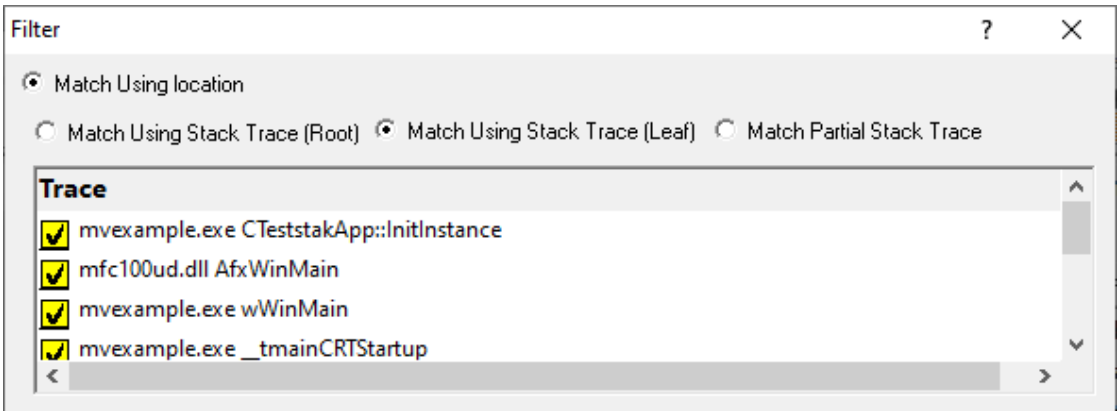
The options above may look complex, but you don't need to set all the options here - just focus on which of the criteria you want to filter by.

Filter by location or type

The dialog is split vertically into two halves: location-based options at the top and type-based options at the bottom

- **Match Using Location** › filter using callstack location or source file location
- **Match Using Filter Type** › filter using memory or handle object datatype attributes

Match using location (callstacks)

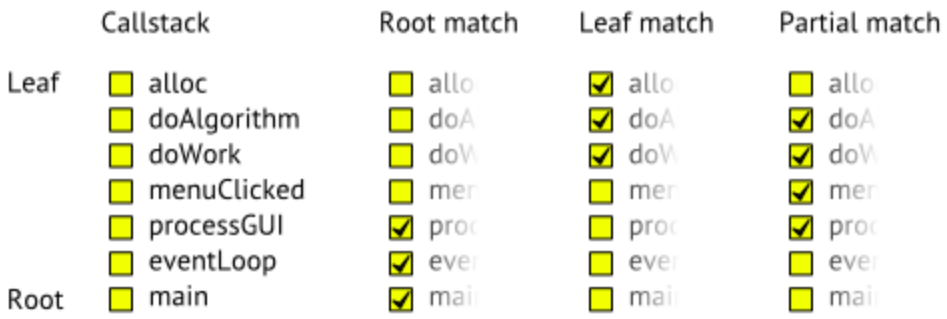


When filtering by all or part of a callstack, that callstack has to come from an existing data item.

Popup menu options on the Memory tab and Analysis tab can create filters using the callstack of the data item.

Having first created a local, session or global filter this way, you can then edit the filter to change exactly how much of the callstack the filter needs to match.

There are three ways to match by callstack: root, leaf and partial, each of which is illustrated below.



- **Match Using Stack Trace (Root)** › filter callstacks containing at least the *root* through all the selected functions

  Ticking a box automatically ticks all items between it and the root.

  Unticking a box unticks all items between it and the leaf

- **Match Using Stack Trace (Leaf)** › filter callstacks containing at least from the *leaf* through all the selected functions

  Ticking a box automatically ticks all items between it and the leaf.

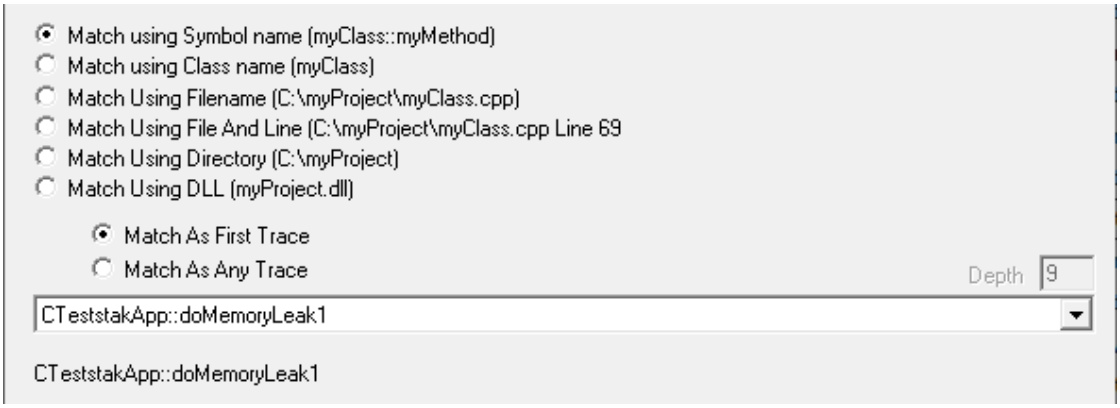
  Unticking a box unticks all items between it and the root.

- **Match Partial Stack Trace** › match any callstack containing at least the selected functions

You can tick any boxes in any order, but they do have to be contiguous.

Filtered items can have the selected items anywhere in their callstack.

Match using location (source)



Match using...

- **Symbol name** › match a selected function name or class::method name from the dropdown list
- **Class name** › match a selected class name from the dropdown list
- **Filename** › match a source file from the list
- **File and line** › match a source file *and* line number

  The list contains combinations of all known allocation points in the source files for the target program
- **Directory** › match a directory file from the list
- **DLL** › match a DLL from the list

  The list contains DLLs loaded by the target program as well as the target executable itself

You can choose how much in the callstack is compared to find the selected item. The measure is from the *leaf* callstack position (see diagram above)

- **Match As First Trace** › looks for the selected item only at the leaf position
- **Match As Any Trace** › looks at within N levels from the leaf position

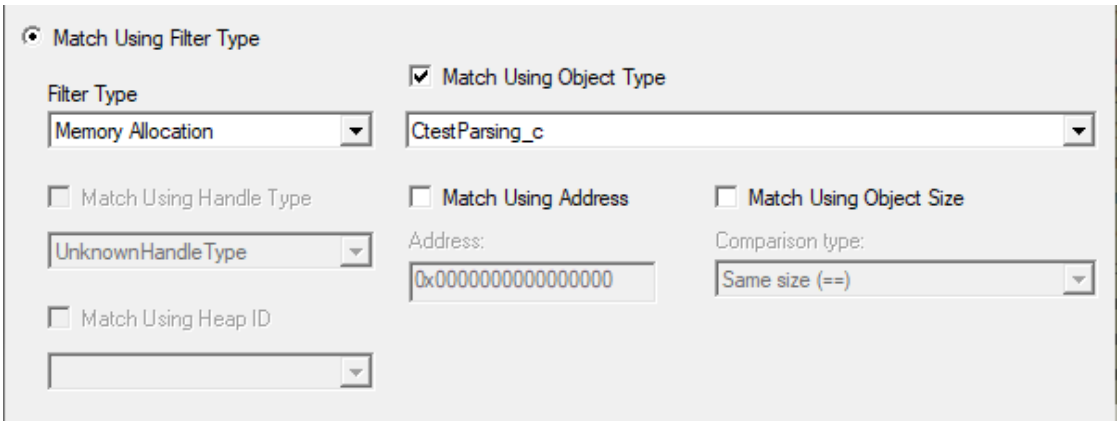
  Enter the **Depth** (on the right) within which to match the selected item

  A depth of 0 (zero) means match the entire callstack.

Match using filter type

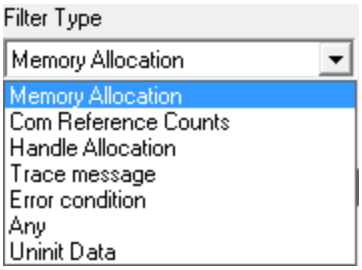
Instead of filtering by callstack location, you can filter according to

- attributes of the object being allocated
- the type of item in the display



First you need to decide what type of filter you want to create.

- **Filter Type** › choose a type of filter from the list below



Changing the filter type enables other parts of the dialog so you can set specific filter values.

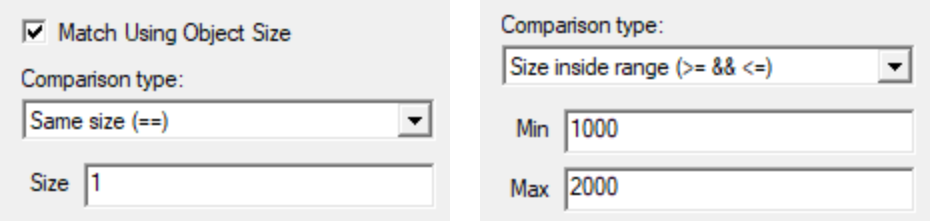
When the filter type is set to Memory Allocation, you can target the filter with object type, size and address.

Some of the filter types simply allow you to include or exclude these types of items in the display.

Match using object type

- **Match Using Object Type** › choose a type from the **Object Type** dropdown list

Match using object size



- **Match Using Object Size** › choose from the **Comparison type** dropdown list and a size, or size range

  Changing the comparison type will cause the display to update to show the appropriate controls.

  You can match exact sizes, not equal to a size, greater than or less than a size, or inside or outside a range.

Remember filters *exclude* data, so take care with your logic on this (and other) options!

Match using address

Matching address is useful if want to select a particular COM Object for example, so that all `AddRef()`, `Release()` and `QueryInterface()` calls can be shown.

- **Match Using Address** › enter a specific allocation address to match against

Combined with the Invert Match option below, this option is great for displaying data about just one particular object.

Match using handle or heap

When the filter type is set to Handle Allocation, you can filter by handle type and/or heap id.

- **Match using Handle Type** › select the type of handle to filter

  Common resource handles might be Bitmap, Brush, Font etc, but there are a host of other handle types in the list.

- **Match using Heap ID** › select the id of the heap to filter, if any have been created

  If a heap has been named, the heap name will be shown in the list instead of the heap ID

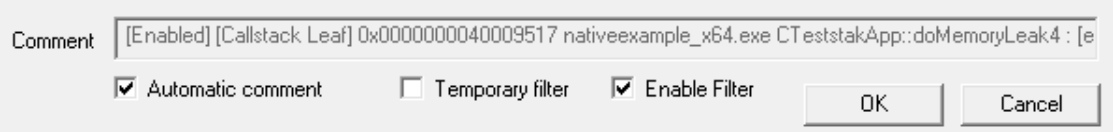
Other Match types

Other filter types include trace messages, error conditions, uninitialised data and COM reference counts.

These types have no further options necessary to filter the relevant data out of a display.

Filter Details

Finally, the details at the bottom let you add a comment to the filter and control how the filter gets applied, if at all.



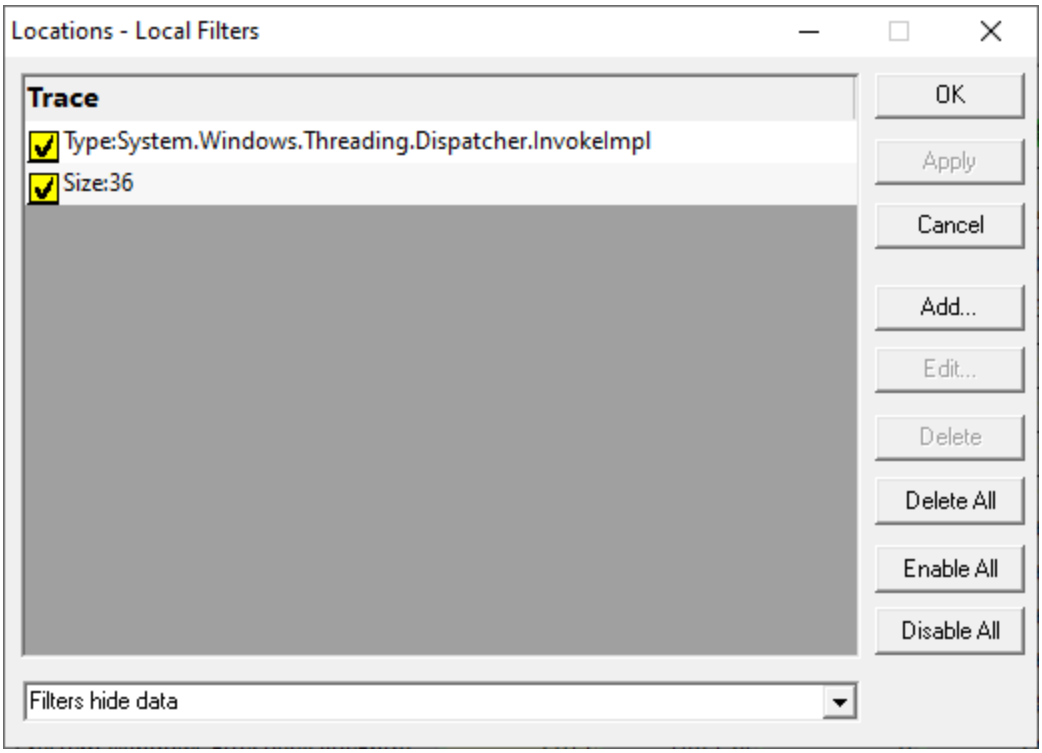
- **Comment** › enter a description of the filter to show in group filter lists

  Usually the comment is auto-generated from the selected filter options, but you can add your own touch here!

- **Automatic comment** › set this if you want an automatically generated comment. Unset this if you want to write your own comment.
- **Temporary Filter** › set the filter to be temporary - meaning it won't be saved
- **Enable Filter** › enables and disables the filter

3.13.2.6 Location Filters

The Location Filters dialog allows you to edit filters for the Locations view.



Local filter management options

- **Add...** › shows a dialog such as the Location Filter Definition to define a new filter to add to the list
- **Edit...** › reopens the definition dialog to view or modify the selected filter

  Double clicking a filter in the list also works.
- **Delete** › removes the selected filters from the list
- **Delete All** › clears the list completely
- **Enable/Disable All** › switches all the filters on or off
- **Apply** › updates the data view with any newly changed local filters and without closing the dialog

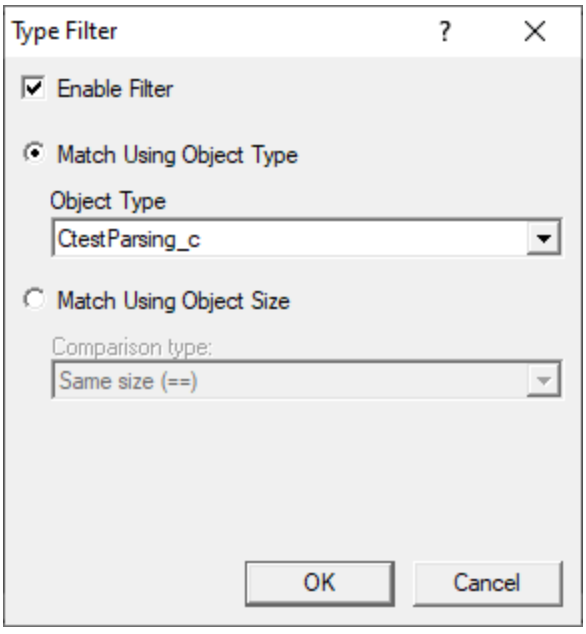
Data display

Whether filters remove data from the display or show data on the display is determined by the combo box at the bottom of the screen.

- **Filters hide data** › data matching the filter is not shown. All other data is shown.
- **Filters show data** › data matching the filter is shown. All other data is not shown.

3.13.2.7 Location Filter Definition

The Location Filter dialog allows you to edit filters on the Locations view.



Enable Filter

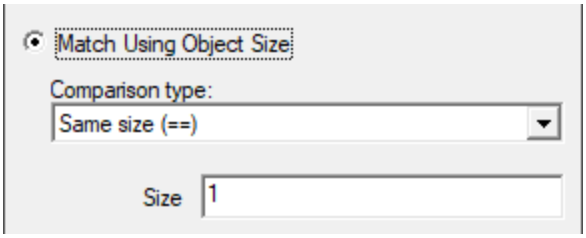
Filters can be turned on and off so that you can easily change what is displayed without having to create/delete filters.

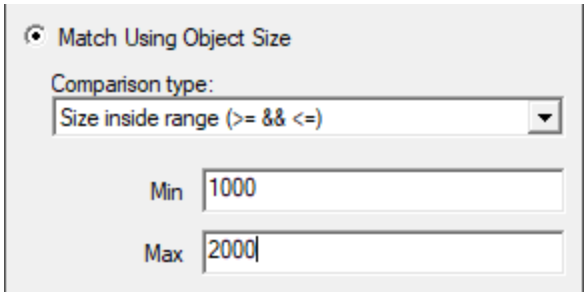
- **Enable Filter** › allow the filter to affect the filtering of data on the Locations view

Match using function name

- **Match Using Function Name** › choose a type from the **Function** dropdown list

Match using object size





- **Match Using Object Size** › choose from the **Comparison type** dropdown list and a size, or size range

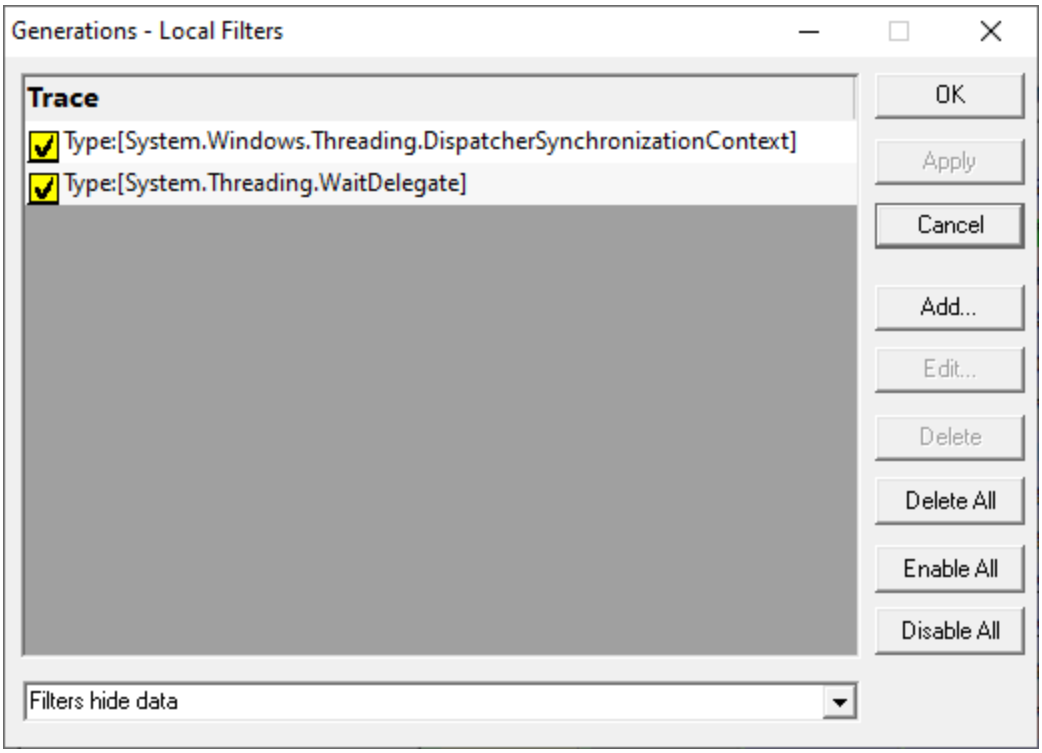
  Changing the comparison type will cause the display to update to show the appropriate controls.

  You can match exact sizes, greater than or less than a size, or inside or outside a range.

Remember filters *exclude or include* data depending on the setting in the Location Filters dialog, so take care with your logic on this (and other) options!

3.13.2.8 Generation Filters

The Generation Filters dialog allows you to edit filters for the Generations view.



Local filter management options

- **Add...** › shows a dialog such as the Generation Filter Definition to define a new filter to add to the list
- **Edit...** › reopens the definition dialog to view or modify the selected filter

  Double clicking a filter in the list also works.
- **Delete** › removes the selected filters from the list
- **Delete All** › clears the list completely
- **Enable/Disable All** › switches all the filters on or off
- **Apply** › updates the data view with any newly changed local filters and without closing the dialog

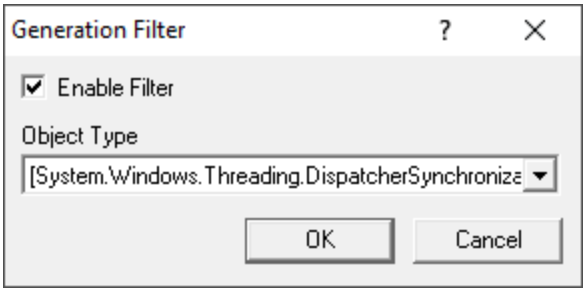
Data display

Whether filters remove data from the display or show data on the display is determined by the combo box at the bottom of the screen.

- **Filters hide data** › data matching the filter is not shown. All other data is shown.
- **Filters show data** › data matching the filter is shown. All other data is not shown.

3.13.2.9 Generation Filter Definition

The Generation Filter dialog allows you to edit filters on the Generations view.



Enable Filter

Filters can be turned on and off so that you can easily change what is displayed without having to create/delete filters.

- **Enable Filter** › allow the filter to affect the filtering of data on the Locations view

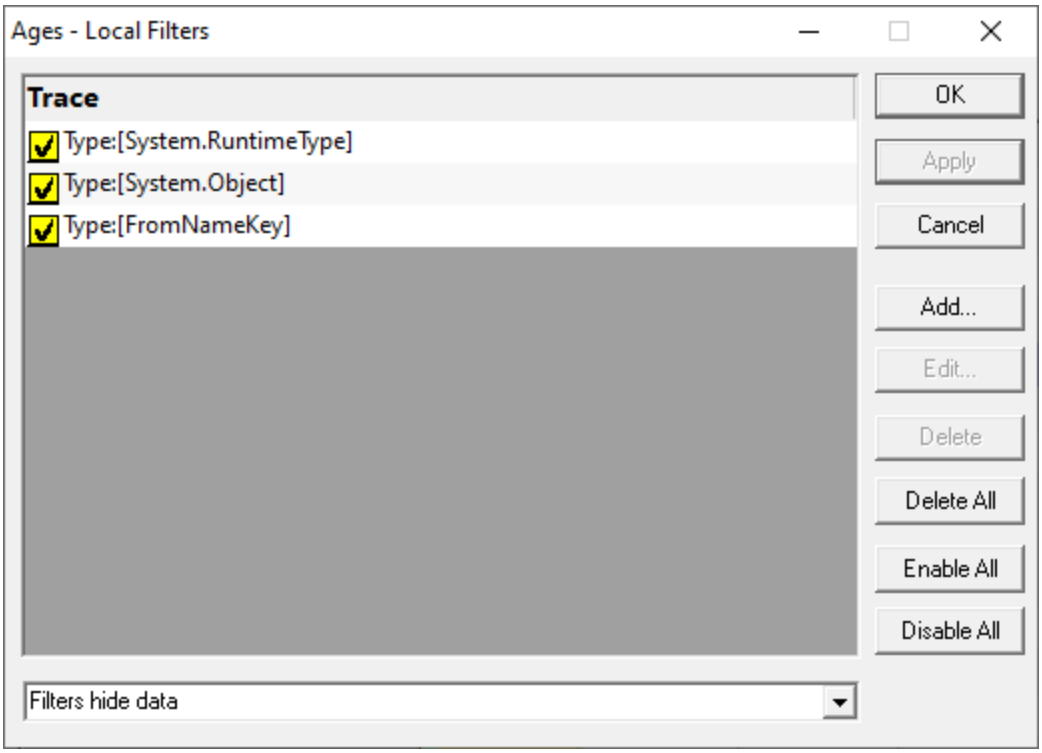
Match using object type

- **Object Type** › choose a type from the **Object Type** dropdown list

Remember filters *exclude or include* data depending on the setting in the Generation Filters dialog, so take care with your logic on this (and other) options!

3.13.2.10 Ages Filters

The Ages Filters dialog allows you to edit filters for the Ages view.



Local filter management options

- **Add...** › shows a dialog such as the Age Filter Definition to define a new filter to add to the list
- **Edit...** › reopens the definition dialog to view or modify the selected filter

  Double clicking a filter in the list also works.
- **Delete** › removes the selected filters from the list
- **Delete All** › clears the list completely
- **Enable/Disable All** › switches all the filters on or off
- **Apply** › updates the data view with any newly changed local filters and without closing the dialog

Data display

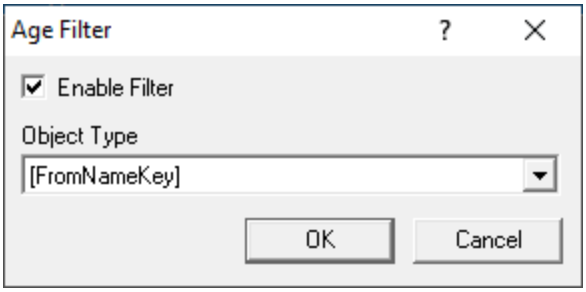
Whether filters remove data from the display or show data on the display is determined by the combo box at the bottom of the screen.

- **Filters hide data** › data matching the filter is not shown. All other data is shown.

- **Filters show data** › data matching the filter is shown. All other data is not shown.

3.13.2.11 Age Filter Defintion

The Generation Filter dialog allows you to edit filters on the Ages view.



Enable Filter

Filters can be turned on and off so that you can easily change what is displayed without having to create/delete filters.

- **Enable Filter** › allow the filter to affect the filtering of data on the Locations view

Match using object type

- **Object Type** › choose a type from the **Object Type** dropdown list

Remember filters *exclude or include* data depending on the setting in the Ages Filters dialog, so take care with your logic on this (and other) options!

3.13.3 Named Heaps

Heap names

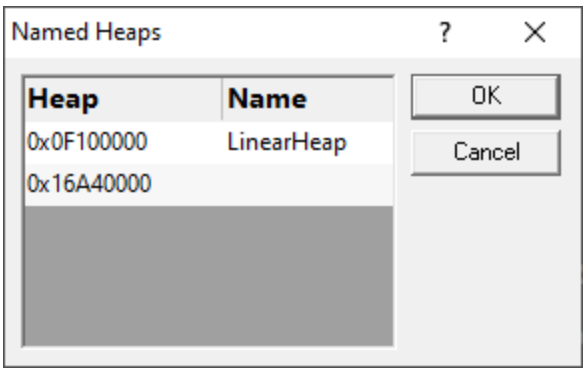
Heap names are used in the following locations in Memory Validator for convenient identification of heaps:

- Filter definition dialog when filtering by handle allocation using a Heap ID
- Memory tab when showing the allocation location against a data item

Naming heaps

The named heap manager lets you view and edit the heap names that are shown when inspecting allocations in heaps that Memory Validator is aware of..

Managers menu › **Named Heaps...** › shows the Named Heaps dialog below



The list shows any known heaps with ID and name.

Heap names will be shown if they were already assigned using the Memory Validator API function `mvSetHeapName()`.

Debug and Release C Runtime Heaps will be automatically named.

Changing heap names

You can name a heap directly:

Double click a box in the name column › enter the name › click outside the box to finish

3.13.4 Watermarks

Watermarks

In Memory Validator, watermarks are event *markers* in the allocation history at which you can say other events occurred either before or after the watermark.

Watermarks are used in the following areas as a kind of filter to constrain displayed data to that which happened between two watermarks.

- The Memory tab
- The watermark view of the Types and Sizes tabs
- The Hotspots tab
- The Analysis tab

Adding watermarks

You can add new watermarks directly from allocation items displayed in the Memory and Analysis tabs, using the popup menu options.

Watermarks added this way are initially named using the .exe or DLL and the function name as in the picture below, but you can change this if you want.

Alternatively, you can add a watermark at the most recent recorded allocation event:

Managers menu › choose **Add watermark at most recent trace** › **enter** a name › click **OK**

Or use the option on the Session Toolbar:

The most recent trace event may not actually be visible in any of the displays as it could be filtered or hidden for other reasons.

First and last watermarks

There are two permanent special watermarks, not directly associated with particular events:

- The **first watermark** is the point *before* every other event
- The **last watermark** is the point *after* every other event

These two watermarks are the default settings, meaning that no data is filtered due to watermarks.

You cannot remove or rename the first and last watermarks.

The watermarks dialog

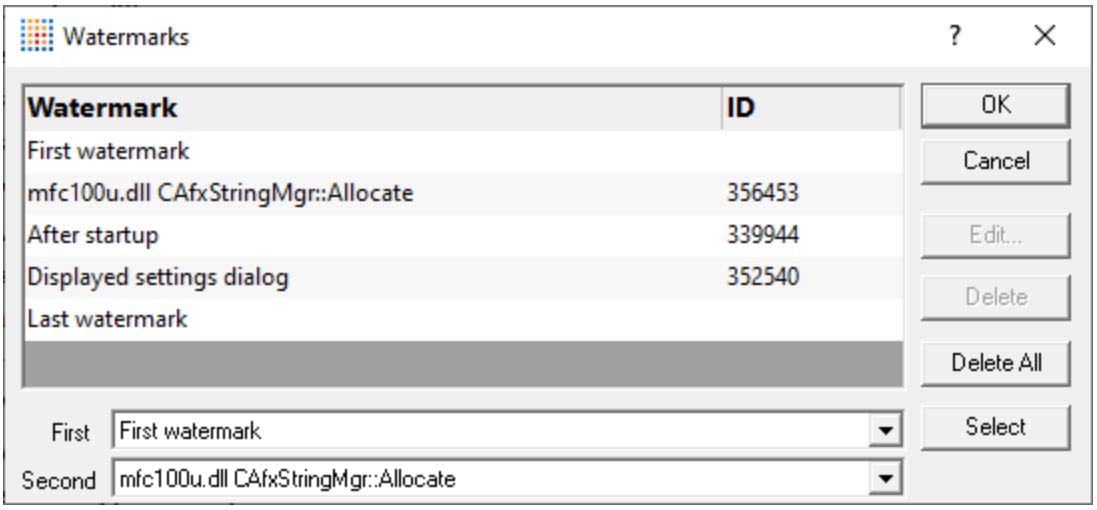
When a session is active, you can show the watermark manager to see a list of watermarks, change their names, or apply selected watermarks to the data views:

Managers menu › **Watermark Manager...** › shows the Watermarks dialog

Or use the Session Toolbar option:

Watermarks are shown in order of their associated event allocation history.

If you haven't added any watermarks yet, this will just be the special first and last watermarks.



The watermarks dialog can only be shown when the Memory, Hotspots or Analysis tab is open.

Managing the watermarks

There's a few options for renaming and removing watermarks:

- **Edit...** › rename the selected watermark

  Double clicking on the watermark also works.

  You can't edit the first and last watermarks.

- **Delete** › delete the selected watermark

  You can't remove the first and last watermarks.

- **Delete All** › delete all the watermarks except the first and last

You can't change watermark locations. If you want to do that, delete the watermarks you don't want and add new ones.

Applying watermarks to the data displays

You can override the local watermark settings in the Memory, Hotspots and Analysis tabs.

At the bottom of the watermarks dialog choose the watermarks:

- **First** › set the earlier of the watermark range
- **Second** › set the later of the watermark range

  You can't choose a second watermark which is earlier than (or the same as) the first one.

The watermarks views in the Types and Sizes tabs are not affected.

3.13.5 Bookmarks

Bookmarks

Bookmarks are event *markers* in the allocation history. You can use the Bookmarks dialog to jump back to a bookmarked location any time.

Bookmarks are only used in the Memory and Analysis tabs.

Adding bookmarks

Adding bookmarks is very similar to adding watermarks.

You can add new bookmarks directly from allocation items displayed in the Memory and Analysis tabs, using the popup menu options.

Bookmarks added this way are initially named using the .exe or DLL and the function name as in the picture below, but you can change this if you want.

Alternatively, you can add a bookmark at the most recent recorded allocation event:

Managers menu › choose **Add bookmark at most recent trace** › **enter** a name › click **OK**

Or use the option on the Session Toolbar:

The most recent trace event may not actually be visible in any of the displays as it could be filtered or hidden for other reasons.

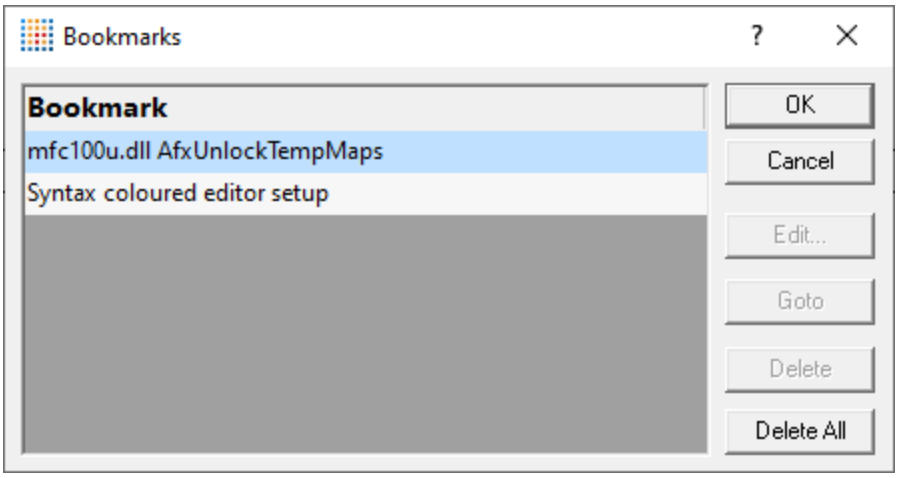
The bookmarks dialog

When a session is active, you can show the bookmark manager to see a list of bookmarks, change their names, or jump to a bookmark location:

Managers menu › **Bookmark Manager...** › shows the Bookmarks dialog

Or use the Session Toolbar option:

Unlike watermarks, bookmarks are shown in the order you add them.



The bookmarks dialog can only be shown when the Memory or Analysis tab is open.

Jumping to bookmark locations

The most useful option in this dialog is the Goto:

- **Goto** › scrolls the open tab to the selected bookmark and selects it

  Double clicking on the bookmark also jumps to its location.

Managing bookmarks

There's also a few options for renaming and removing bookmarks:

- **Edit...** › rename the selected bookmark
- **Delete** › delete the selected bookmark
- **Delete All** › delete all the bookmarks in the list

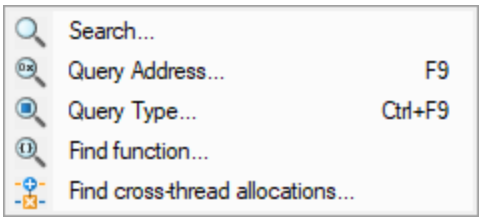
You can't change bookmark locations. If you want to do that, delete the bookmarks you don't want and add new ones.

3.14 Query and Search

Tools to search for allocations

The following tools help you find memory and handle allocations using different criteria and are all found on the Query Menu.

Click on an item in the picture or in the list below to find out more in the following topics.



- **Search** › use the Find Memory dialog to search the data in some of the tab views for different types of memory allocations
- **Query Address** › use the Find Address dialog to search for allocation events at or near an address
- **Query Object** › use the Find Objects dialog to search for allocations of objects by their datatype
- **Find Function** › use the Find Functions dialog to search for allocations occurring in certain functions
- **Find cross-thread allocations** › use the Cross-thread Allocations dialog to find memory allocated in one thread and deallocated in another

3.14.1 Finding memory

Searching for memory

Using the Find Memory dialog below, you can search the data in some of the tab views for different types of memory allocations.

Searches can be:

- based on allocation location such as in a function, file or module
- based on the allocated object such as size, type or address
- based on other identifiers like tag id, sequence id, or allocation type

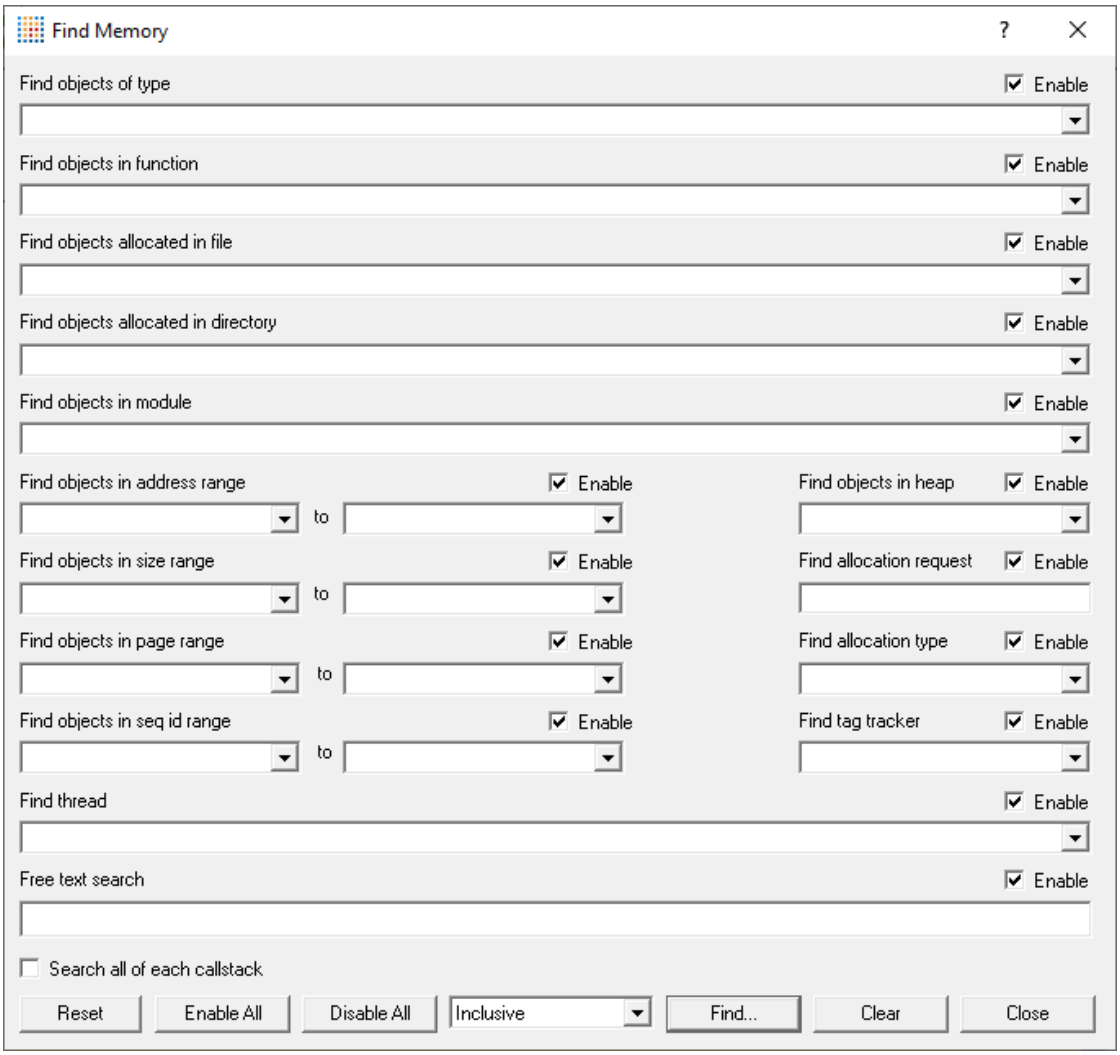
The find memory dialog

To show the Find Memory dialog, choose the menu option below:

Query menu › choose **Search...** › displays the Find Memory dialog

Or use the search icon on the Query Toolbar.

The Find Memory dialog has many search options, each of which can be switched on and off.



There is a different instance of the dialog for each of the following tabs:

- The Memory tab
- The Hotspots tab
- The Analysis tab
- The Pages tab

Note that the Find Memory dialog can remain open while you inspect the results.

Search Criteria

Each of the desired search options needs to be enabled and criteria specified:

To find objects...	enable the option and choose...
• **of type**	a data type from the drop down list
• **in function**	a function from the list

- **allocated in file** a file from the list
- **allocated in directory** a directory from the list
- **in module** an .exe, DLL or other module
- **in heap** a heap ID
- **with allocation id** the allocation request id
- **of allocation type** opne of alloc, realloc or free from the dropdown list
- **with tag tracker** a tag tracker id
- **by thread id** a thread id

Objects between...

- **addresses** type in the values or choose the current minimum or maximum from the drop down list
- **sizes**
- **pages**
- **sequence ids**

Free text search

- **event description** a string that is a subset of an event description. * wildcard is supported.

For options where there is a set choice of values, the lists show all the options that Memory Validator knows about at the time the dialog is shown.

Where a range is specified, you can enter the same value for the beginning and end of the range.

 If you want to search by object type, note that Memory Validator only fetches type information for each memory location when it is requested. This improves performance, but can mean the drop down list doesn't have all the datatypes in yet. If you don't see a datatype you want, type it in instead, and the list will be updated when possible.

Resetting fields

To start over and create a new set of search criteria, there's a handy reset option:

- **Reset** › clears all the fields and enables all options as seen in the picture above

Searching callstacks

By default, callstacks are not searched, but if you want to search all stack frames you can.

- **Search all of each callstack** › if selected all frames of each callstack are searched as well as the top frame

Exclusive or inclusive searches

There are two ways of searching:

- **Exclusive** › search results satisfy ***all*** the enabled search criteria
- **Inclusive** › search results satisfy ***at least one*** of the enabled search criteria

As a general rule, exclusive searches return a tightly focused set of results, while inclusive searches return a broader set of results.

For those who prefer logic notation, exclusive searches use AND logic, while inclusive use OR logic.

Performing the search

Once search criteria are enabled, and search mode selected, results can be found

- **Find...** › matching results in the corresponding tab are highlighted

  The colour of highlighted objects is set on the colours tab of the Global Settings dialog.

  The first time a search with a new search criteria is performed the first item in the search results is navigated to and selected.

  Subsequent searches without changing the search criteria will navigate to the next search result, or if shift is held, the previous search result.

  Ctrl F is the equivalent of clicking the **Find...** button.

- **Clear** › all previous search results in the corresponding tab are cleared

Example

In the picture above, the dialog shows a search setup to find:

- objects of size 4
- objects of type CtestParsing_c.

The results of that search are shown in the Memory tab below, first as an exclusive search and then an inclusive one.

Example results: Exclusive results, showing only the 4 byte CtestParsing_c object allocation:

Example results: Inclusive results, now showing the 4 byte CString objects also selected.

3.14.2 Finding addresses

Searching for allocation addresses

Using the Address Query dialog below, you can search the data for allocation events at or near an address.

Searches can include:

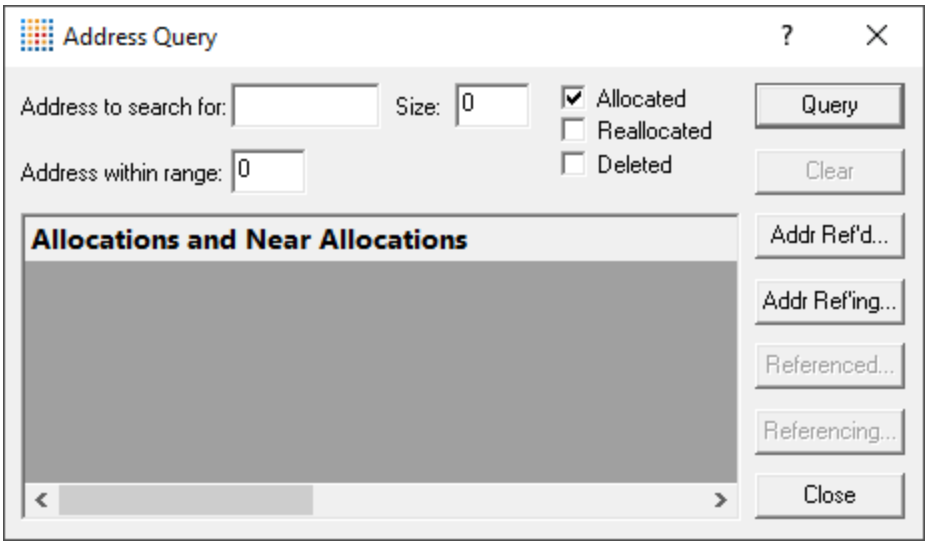
- allocations in a memory block of a certain size, starting at an address of your choice
- allocations within a specified range of a chosen address (+/-)

The address query dialog

To show the Address Query dialog, choose the menu option below:

Query menu › choose **Query Address...** › displays the Address Query dialog

Or use the following icon on the Query Toolbar.



Search criteria

The search address is essential, but size and range are optional for adding some tolerance.

The format of these values can be decimal or prefixed with 0x for a hexadecimal address.

- **Address to search for** › enter the memory address of interest
- **Size** › choose a size of memory block within which to search for allocations
- **Address within range** › enter a tolerance range for finding allocations near to the address

  The range can help find potential candidates for memory overrun, underrun and corruption problems.

Allocation types

As well as allocations you can search for reallocations and/or free events:

- **Allocated** › search for allocations
- **Reallocated** › search for reallocations
- **Deleted** › search for free events

Query results

- **Query** › performs the search

  The search results are added to the list in the dialog.

- **Clear** › clears all the results from the list

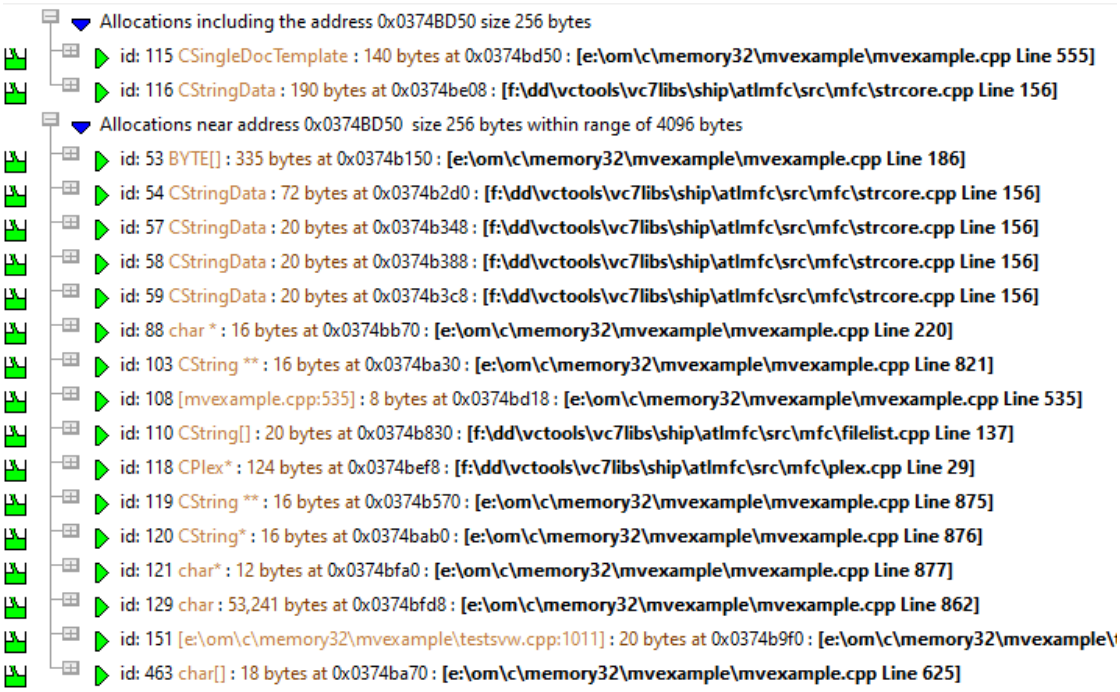
  The list is not automatically cleared with each search, so you can compare results of different searches

The search results may be shown in the following sections if you entered any of the optional criteria:

- results including the address
- results within the size, if specified
- results within range, if specified

You can specify decimal values and hexadecimal values for your search query.

For example, when searching for address 0x037e6120 with a size of 30 and a range of 0x1000:



Results appear in the earliest of the above sections. They are not repeated in the case where the size and the range might capture overlapping results.

You can expand the search results, and double click the data items to edit source code in your preferred editor.

Finding referenced objects

As well as finding an object at an address, you can find which objects it may reference and which objects may reference it:

- **Addr Ref'd...** › find other objects which are referenced by the object at the search address
- **Addr Ref'ing...** › find other objects that reference the object at the search address

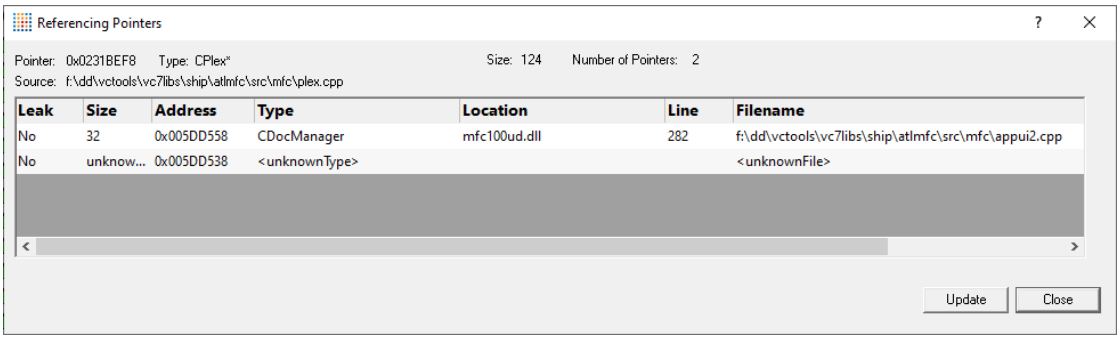
Similarly, within the main search results, you can select an item and find potentially referenced and referencing objects:

- **Referenced...** › find other objects which are referenced by the selected object
- **Referencing...** › find other objects that reference the selected object

The target program must still be executing for references to be found. You wouldn't be able to find references in a session that has been loaded from a file.

The references dialog

Referenced and referencing pointers are listed in a new dialog.



- **Update** › refresh the results

You can keep several of these dialogs open if you want to compare results.

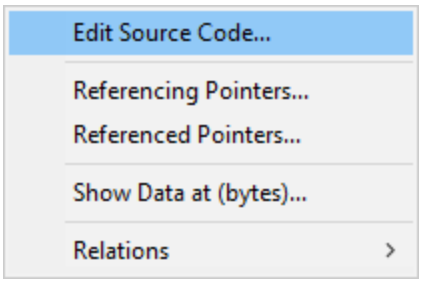
You can double click a row in the Referencing Pointers dialog to edit source code if its available.

It's normal for there to sometimes be a pause before referenced objects are displayed. This is while the data is being found.

References dialog popup menu

The following popup menu is available over the data area

Click on any part of the menu to jump straight to the topic below:



- **Edit Source Code...** › view the source code in the source code editor
- **Referencing Pointers...** › view pointers referencing the selected address
- **Referenced Pointers...** › view pointers referenced by the selected address
- **Show Data at (bytes)...** › view the contents of memory at the selected address
- **Relations** › a submenu of additional queries related to the current allocation

Menu option: relations

The relations menu has a large sub-menu with many different options for choosing a set of related data to display in the upper analysis window.

Think of this as a sub-query on the working data - like searching for friends of friends on a social network!

Available relations are as follows, with *allocations* generally meaning any allocation, reallocation or deallocation

• **Same address** • **Same size** • **Smaller** • **Larger**	Finds any other allocations on the same memory address (allocations, reallocs, or frees) Allocations on any memory objects of identical size or on smaller or larger objects
• **Same object type**	Finds any other allocations of the same type
• **Same source file** • **Same DLL**	All allocations from the same file... or the same DLL
• **Allocations within**	For memory allocations, finds all other allocations within a range of 32 bytes up to 4Kb of this one

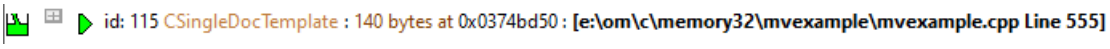
➔ Also see the examples below.

Examples of searches and finding referenced objects

Getting an address to search for

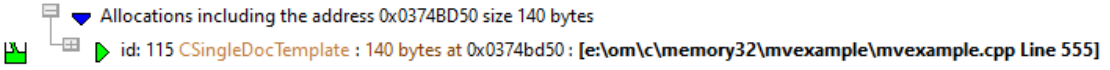
Memory Validator has an example program with which to safely explore all the features available.

While the example program is running, refreshing the Memory Tab view should show an allocated object of type `CSingleDocTemplate`:



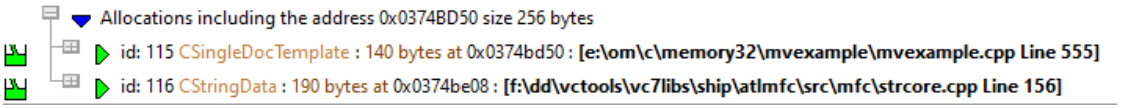
Address to search for

Using the Query Address dialog to search for that object's address and size (address 0x0060A10, size 140 in this case) should show the expected result:



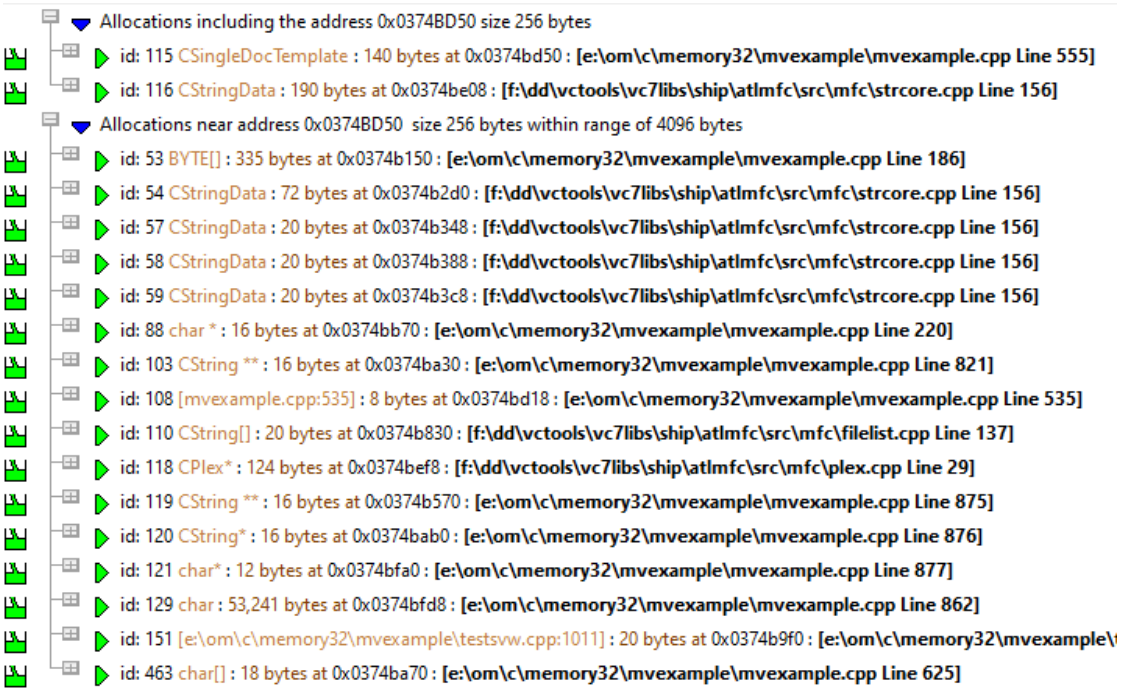
Size

Expanding the size to 0x100 may find more objects. For example:



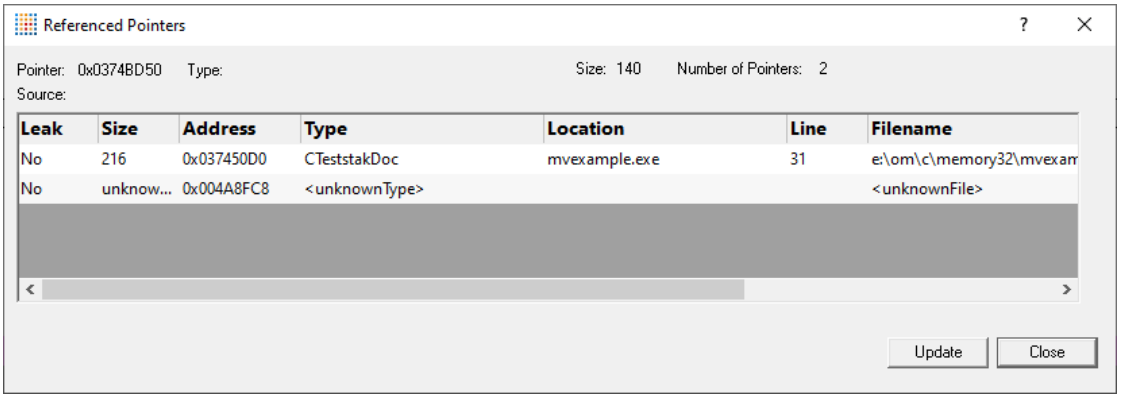
Address within range

Additionally, setting the range to 0x1000 might give the result:



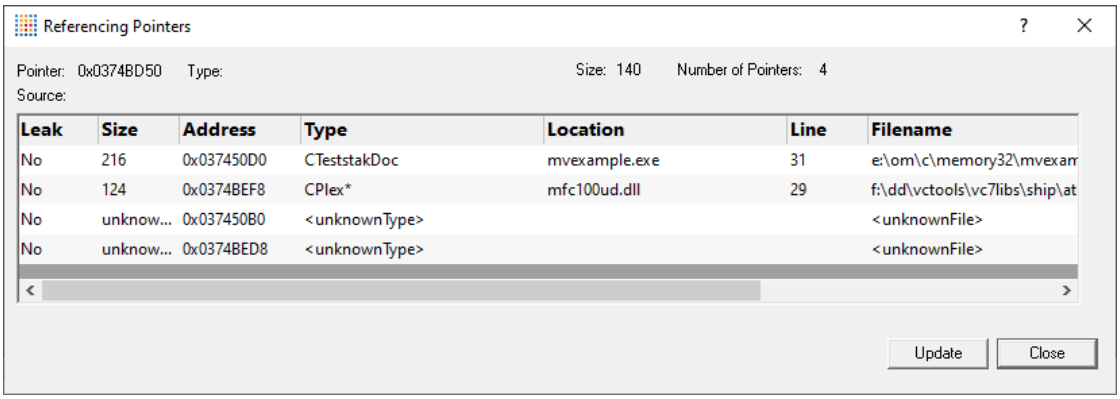
Addr Ref'd

Clicking Addr Ref'd to find referenced objects should include `CTeststakDoc` being referenced by the `CSingleDocTemplate` :



Addr Ref'ing

Clicking Addr Ref'ing to find objects referencing the object at the address 0x0231BD50 might show something like:



3.14.3 Finding objects

Searching for object allocations

Using the Object Query dialog below, you can search for allocations of objects by their datatype.

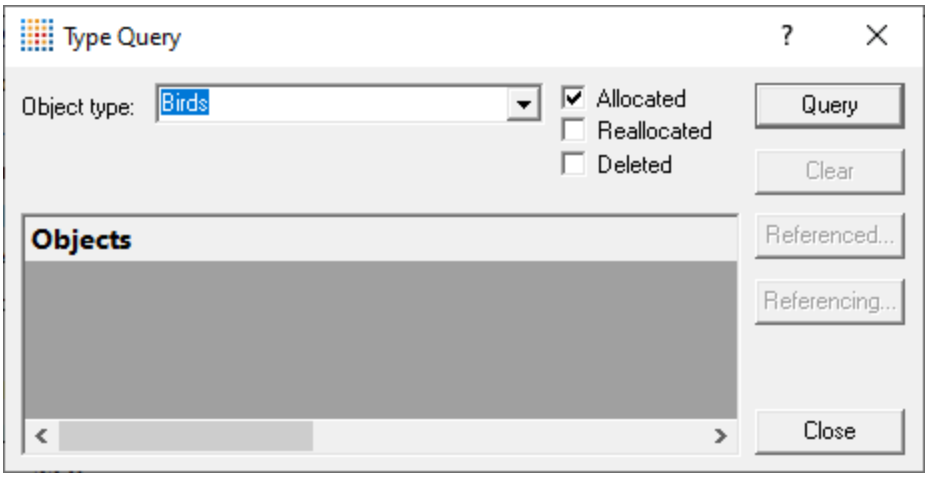
The object query dialog is very similar in behaviour to finding addresses.

The object query dialog

To show the Object Query dialog, choose the menu option below:

Query menu › choose **Query Object...** › displays the Object Query dialog

Or use the following icon on the Query Toolbar.



Search criteria

Just choose the object type for which you want to find allocations.

- **Object type** › choose the datatype of the objects you want to find

Allocation types

As well as allocations you can search for reallocations and/or free events:

- **Allocated** › search for allocations
- **Reallocated** › search for reallocations
- **Deleted** › search for free events

Query results

- **Query** › performs the search

  The search results are added to the list in the dialog.

- **Clear** › clears all the results from the list

  The list is not automatically cleared with each search, so you can compare results of different searches

You can expand the search results, and double click the data items to edit source code in your preferred editor.

Finding referenced objects

Within the search results, you can select an item and find potentially referenced and referencing objects:

- **Referenced...** › find other objects which are referenced by the selected object
- **Referencing...** › find other objects that reference the selected object

Referenced pointers are listed in the same references dialog used when finding addresses.

➡ Also see the examples below.

Examples of searches and finding referenced objects

Memory Validator has an example program with which to safely explore all the features available.

Among the datatypes it uses is `CString` and `CTeststakDoc`, used in the examples below.

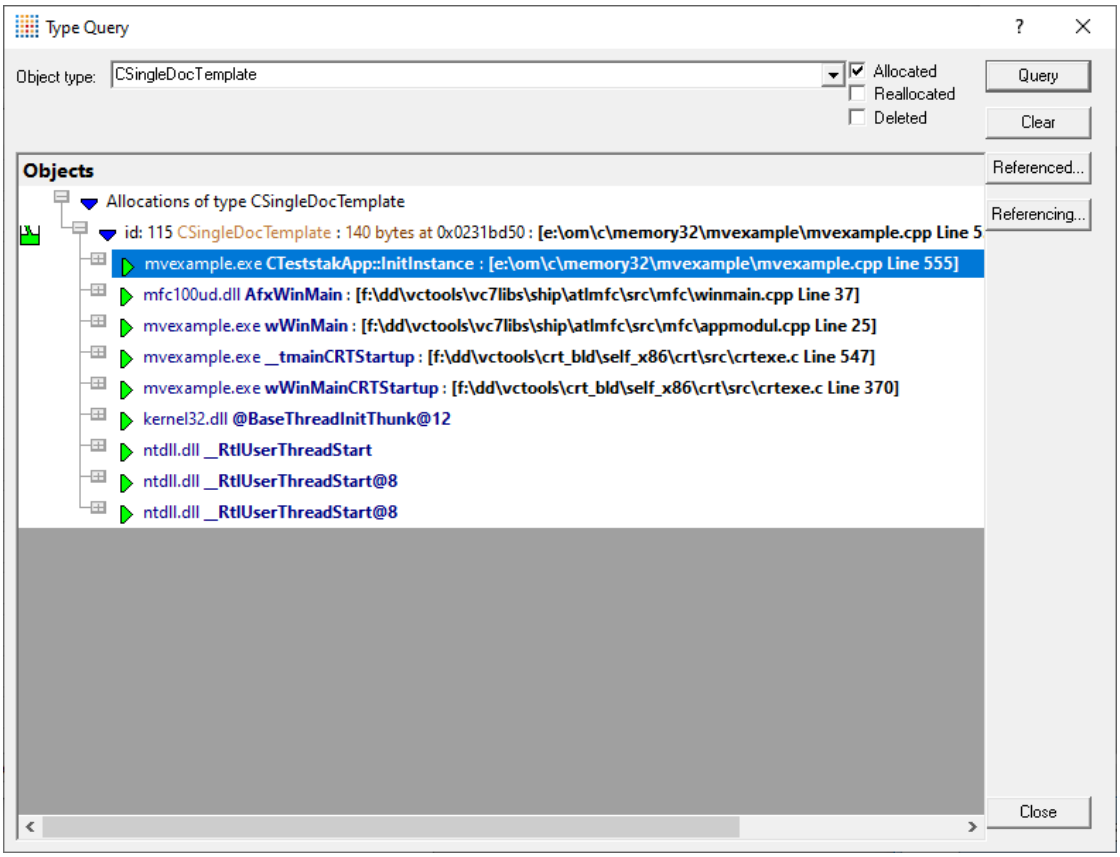
Searching for object types

If you searched for a datatype with many allocation instances, they would all be listed.

For example `CString`

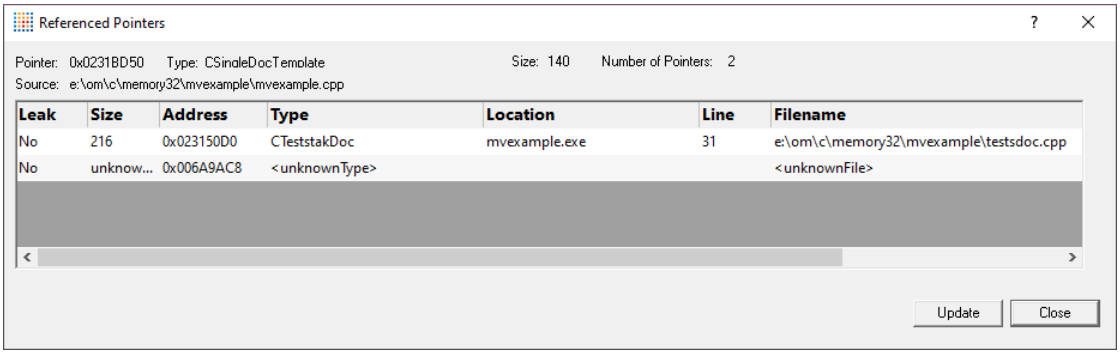
Below are the results of searching for `CSingleDocTemplate` datatype allocations.

The results are expanded to show the source code surrounding the allocation point.



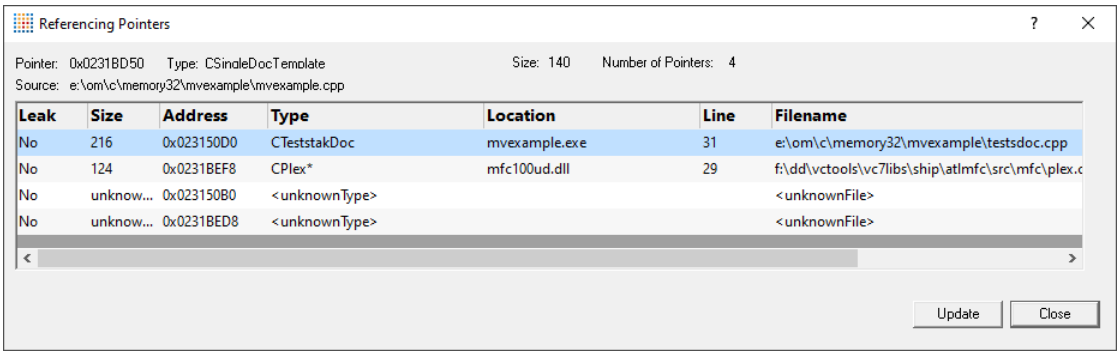
Referenced

Selecting the object in the list and clicking Referenced finds referenced objects and should include `CTeststakDoc` being referenced by the `CSingleDocTemplate` object:



Referencing

Clicking Referencing to find objects referencing the `CSingleDocTemplate` object might show something like:



3.14.4 Finding functions

Searching for object allocations in functions

Using the Find Function dialog below, you can search for allocations occurring in certain functions.

The functions can be the allocating function or anywhere in the allocation callstack

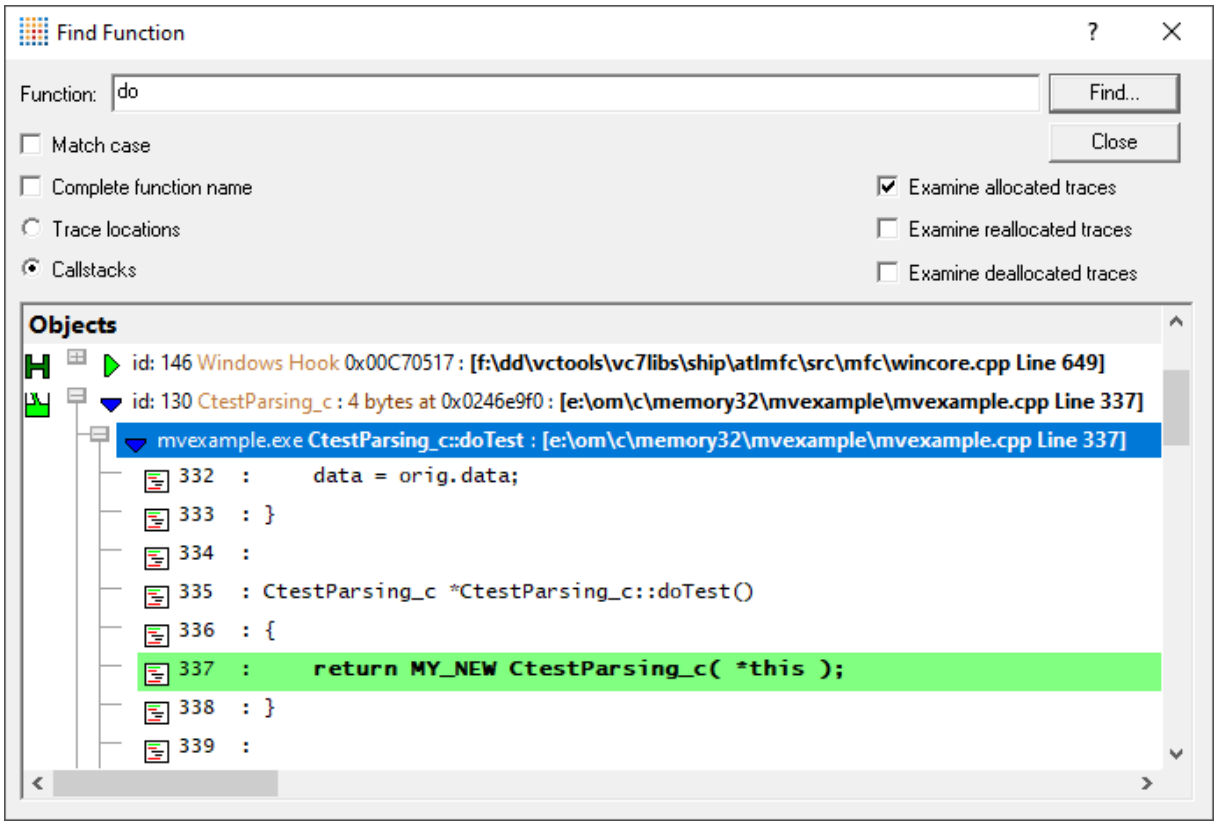
This feature is very similar in behaviour to finding objects.

The find function dialog

To show the Find Function dialog, choose the menu option below:

Query menu › choose **Find Function...** › displays the Find Function dialog

Or use the following icon on the Query Toolbar.



Search criteria

Enter a function name, and optionally any other search characteristics to find objects allocated in matching functions

- **Function** › enter full or partial function name
- **Match case** › tick to do a case-sensitive match
- **Complete function name** › tick to only match the whole name

  For C++ methods, complete names must be of the form classname::methodname.
- **Trace locations** › matches only the function containing the allocation
- **Callstacks** › matches any function in the allocation callstack

Allocation types

As well as allocations you can search for reallocation and/or deallocation events:

- **Allocated** › search for allocations
- **Reallocated** › search for reallocations
- **Deallocated** › search for deallocation events

Finding results

- **Find** › performs the search displaying results in the list

  Results *replace* any previous search, unlike querying addresses or objects where the results are *added* to the list.

You can expand the search results, and double click the data items to edit source code in your preferred editor.

Examples of finding allocations in functions

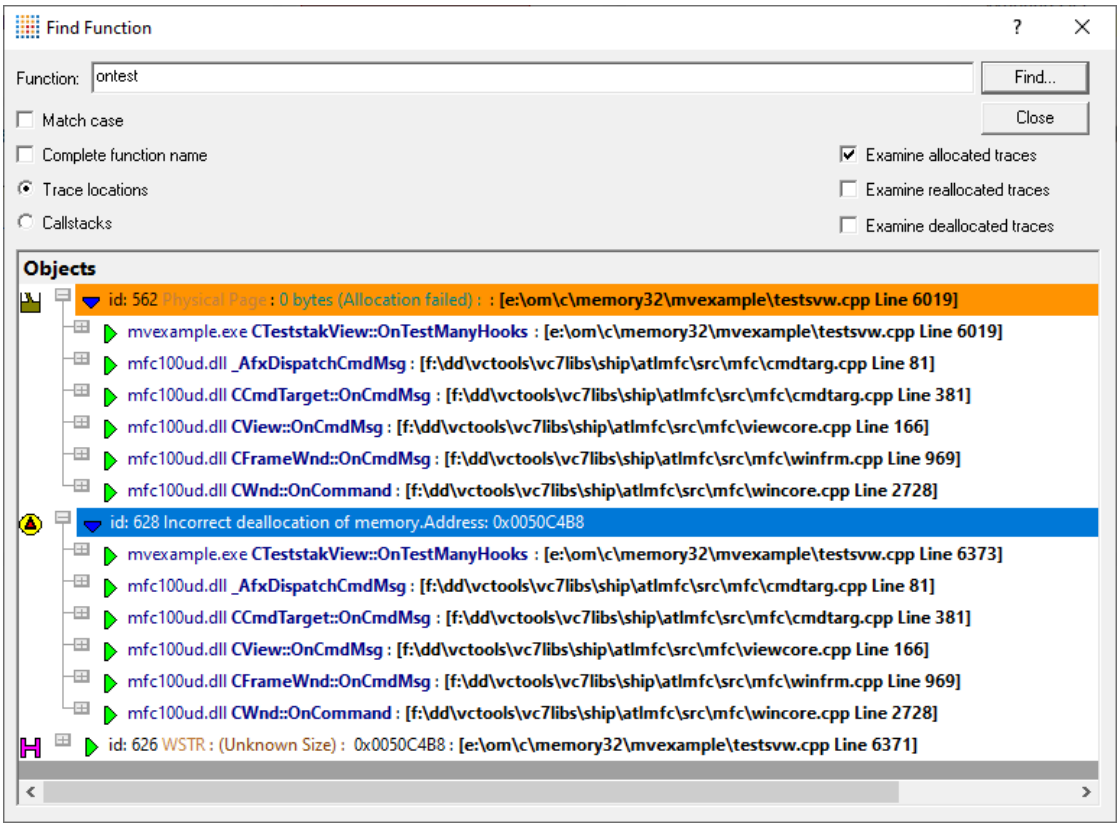
Memory Validator has an example program you can use to safely explore features.

In the example program, the **Allocation** menu has an option **Test Many Hooks at once**.

After doing that, the example searches can be made below.

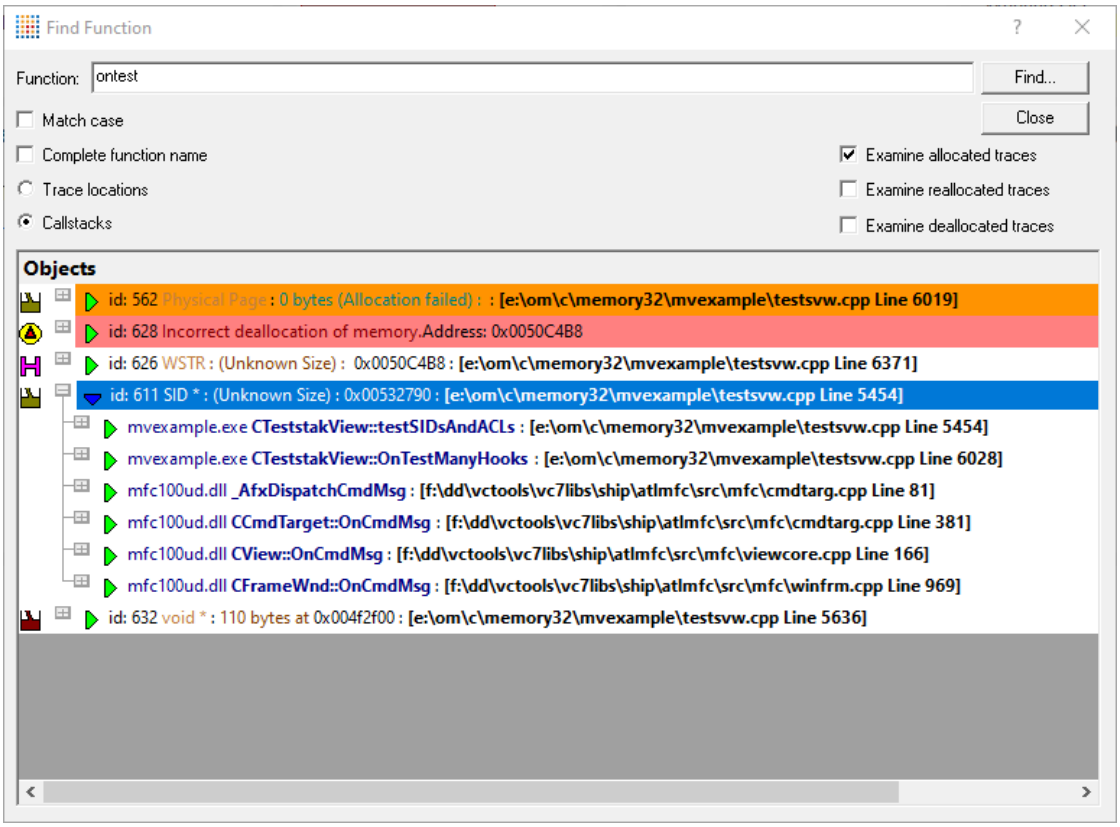
Searching only within matching functions

Searching for allocations only in functions that have `ontest` as part of their name finds these results in `CTeststakView::OnTestManyHooks`



Searching anywhere in the callstack

Keeping the same function name but now searching in *any* part of the callstack additionally finds two more allocations in functions underneath `CTeststakView::OnTestManyHooks`



3.14.5 Finding memory allocations deallocated in different threads

Cross-threaded allocations

Cross-thread allocations are when memory is allocated in one thread, and is deallocated or reallocated in another thread.

Memory Validator connects these events, as well as resource handle allocations and releases.

Cross-thread allocations are perfectly valid, for example, a main thread may place data objects onto a queue in a worker thread that deletes them after processing.

However, the practice can also result in unintentional behaviour as well. For example, after a process that started off single threaded has been made multi-threaded.

Often these errors occur when memory is *unexpectedly* deallocated in a diffferent thread.

Using the cross-thread allocations dialog below, you can find these cross-thread allocations and deallocations.

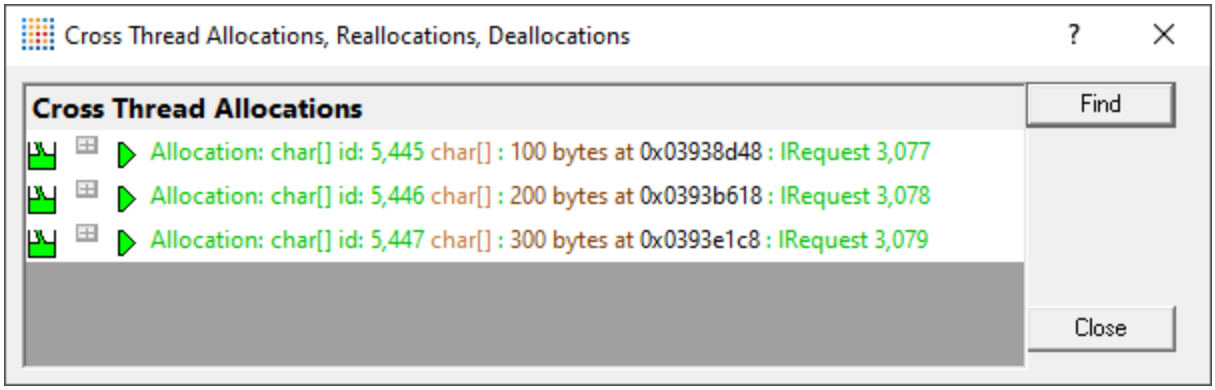
By inspecting them you can verify expected behaviour, and look out for unexpected behaviour.

The cross thread allocations dialog

To show the cross-thread allocations dialog, choose the menu option below:

Query menu › choose **Find cross-thread allocations...** › displays the dialog

Or use the following icon on the Query Toolbar.



There is just the one button - to start the search

- **Find** › start the search and display results in the list

  Results include cross-thread reallocations and deallocations.

  For larger datasets, the search can take some time.

Note that results displayed in green indicate that some (possibly all) allocations made on this callstack have been successfully deallocated.

Example of finding cross-thread allocations

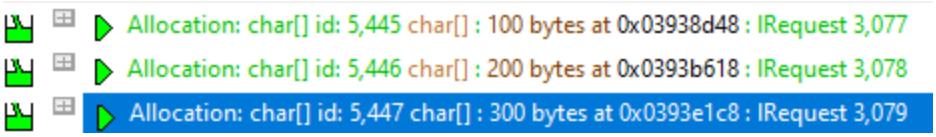
Memory Validator has an example program with which to safely explore all the features available.

The following sequence will allocate memory in one thread and free it in another.

mvExample.exe › **Handles** menu › **Start thread** › **Exit** program

Searching for cross-thread allocations

The following three `char[]` allocations should be found, with sequential allocation request ids.

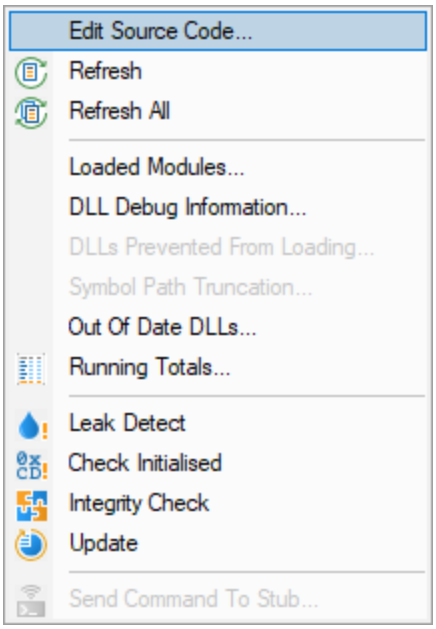


Expanding one of these entries shows the allocation location *and* the deallocation location below.

The various details include thread id (different for allocation and deallocation), timestamp, request id and callstack.

3.15 Tools

Click on an item in the picture of the Tools Menu below to jump to the relevant topic:



3.15.1 Colour coded source code editor

Source code editing

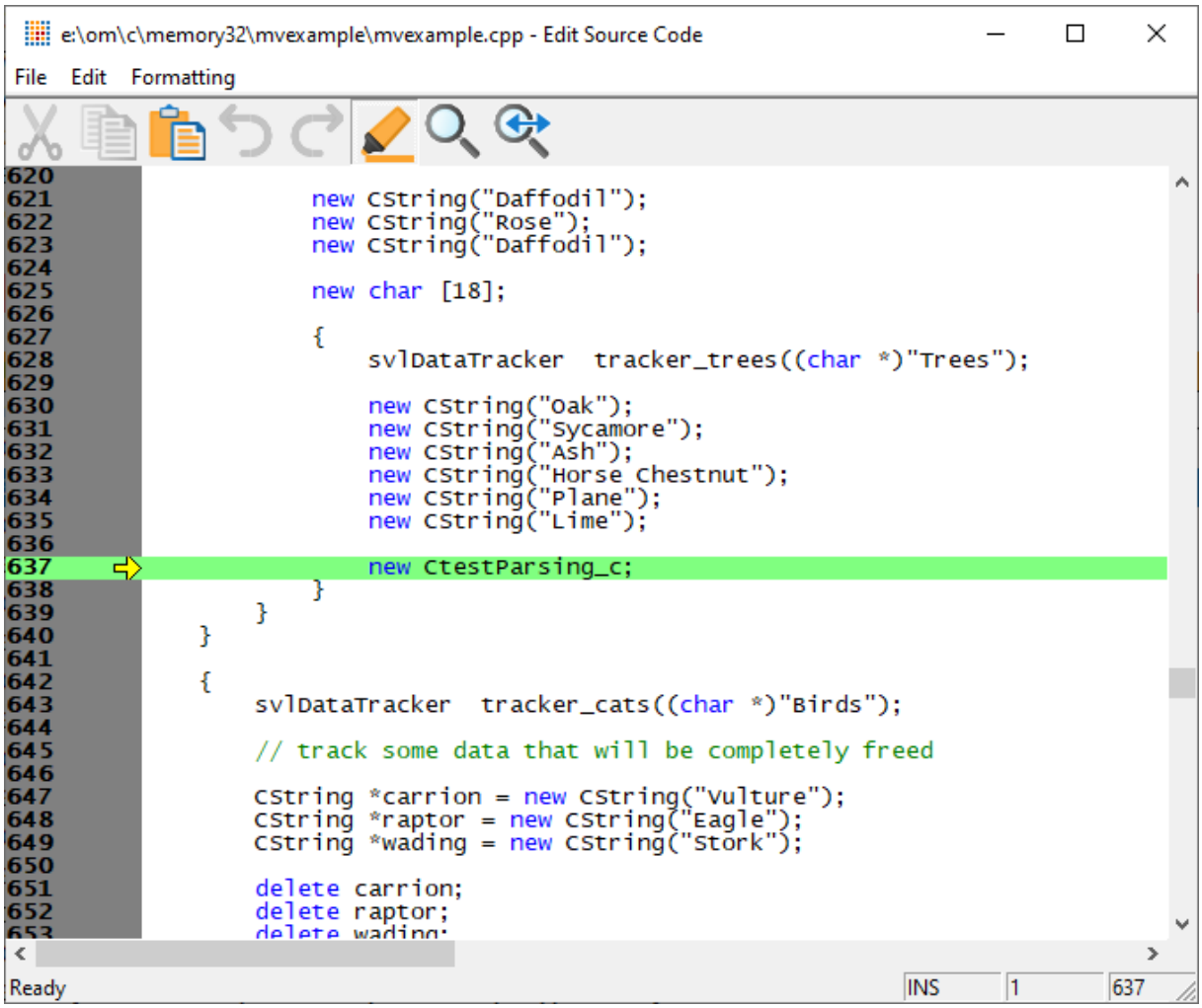
The editing settings let you set an editor of your choice to view or edit source code. Memory Validator's built-in editor is one of those options.

The built-in editor can be started in several ways:

- double click on a source code fragment in one of the views
- **popup** menu > **Edit Source Code...**
- **Tools** menu > **Edit Source Code...**

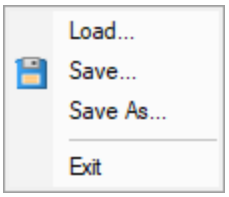
Using the built-in editor

The built-in editor supports the basic operations expected for editing source code:



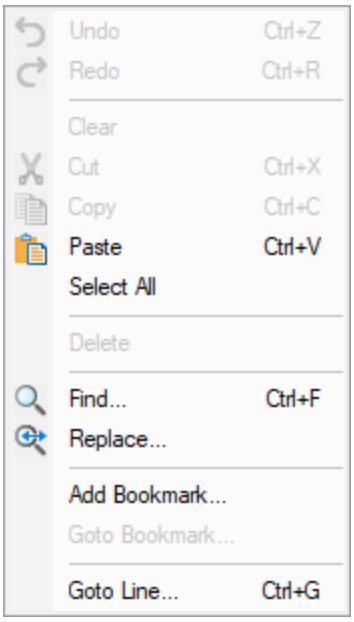
File menu

The file options need no explanation:



Edit menu

All the following edit options should also be familiar:

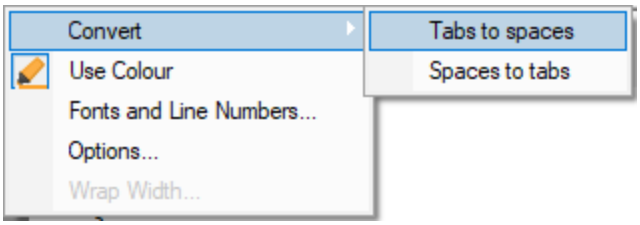


Undo/Redo is unlimited by default, but this can be changed in the options below.

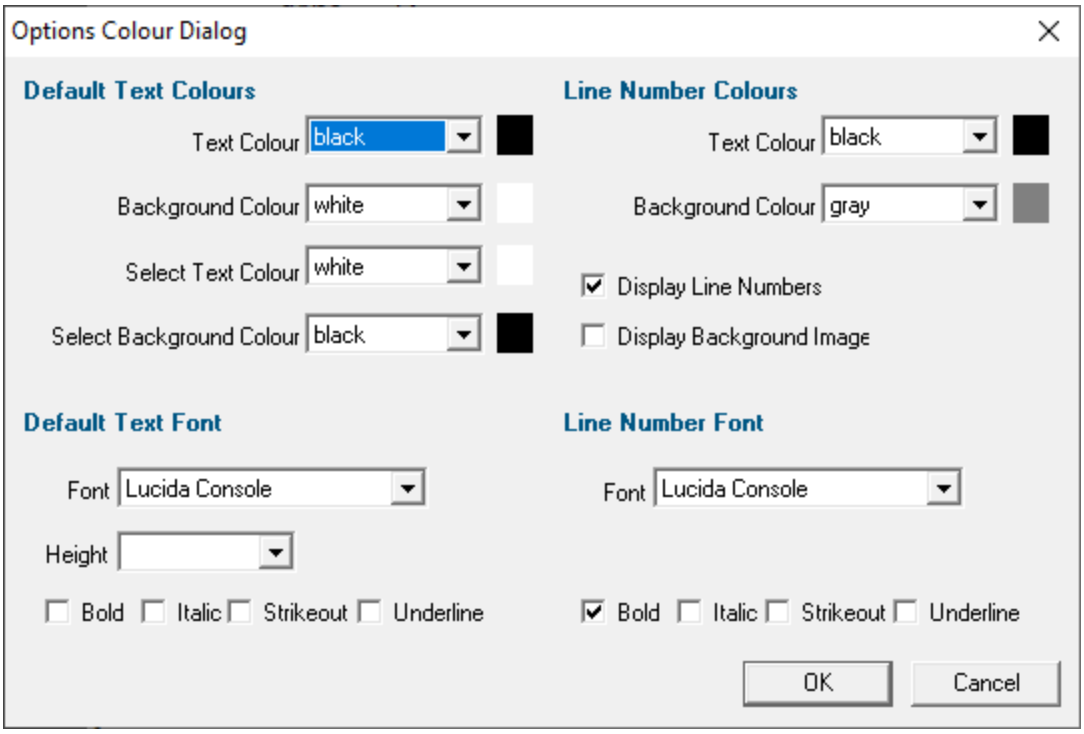
Editing bookmarks has nothing to do with Memory Validator's own bookmarks.

Formatting menu

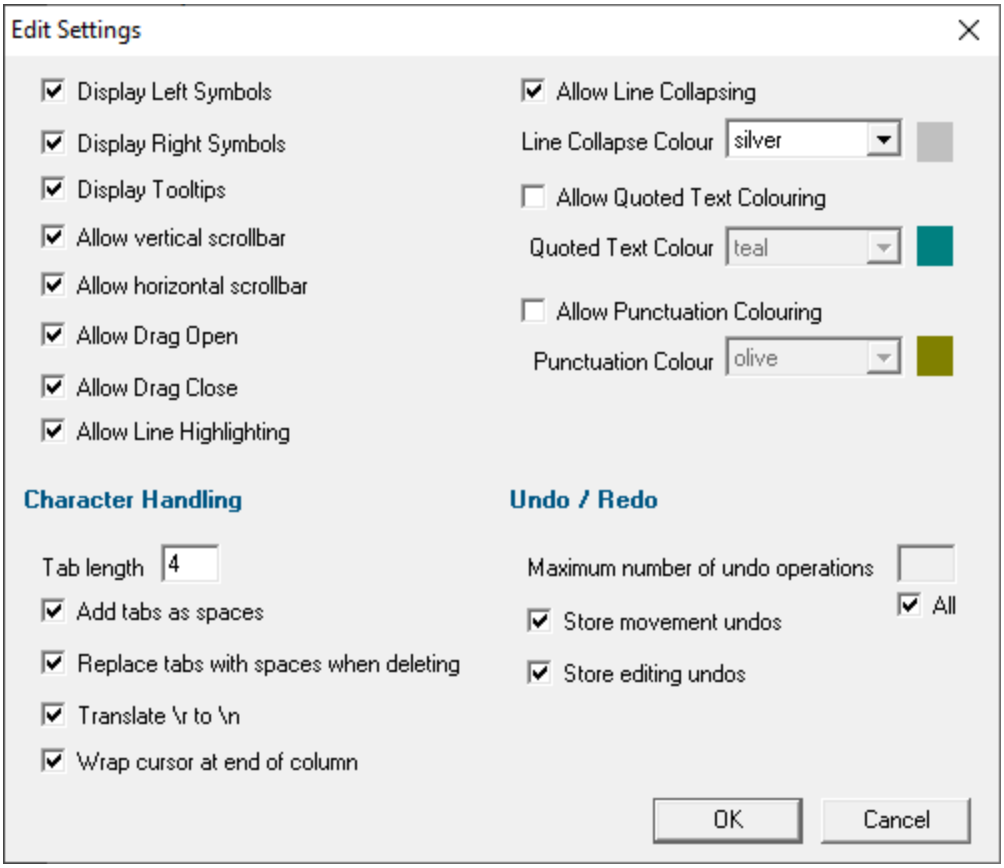
The formatting menu has general display and editing options



- **Convert** › **Tabs to spaces** › turns all tabs into spaces
- **Convert** › **Spaces to tabs** › turns all spaces into spaces
- **Use Colour** › toggles the colour coded display
- **Fonts and Line Numbers...** › change text colours, fonts and line numbers



- **Options...** › set tab length and other options



- **Wrap Width...** › changes the column width at which lines will wrap in the display

Status bar

The status bar shows help text at the bottom as you hover over menu and toolbar options.

To the right of the status bar are insert mode, column number and line number.

Line collapsing

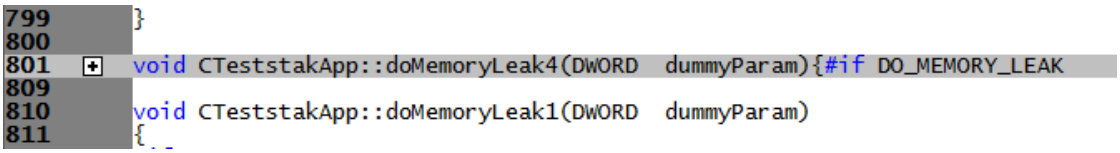
You can temporarily collapse sections of code as follows:

- **Left click** in the margin to start the section › **Drag** to define the length › **Release** to set the end of the section

  Click anywhere on the resulting indicator to collapse, and on the + to expand a section.

Expanded:

Collapsed:



Line collapsing is *temporary* and not remembered between edit sessions.

3.15.2 Refresh and Refresh All

Refreshing data

You have the option in some views to automatically update in the view at an interval of your choice.

Sometimes you need to refresh the data when *you* want to, especially while inspecting the data.

Most views have a local refresh button, which updates the data.

The same function is found in the Tools menu, as well as an option to update all views at once.

Tools menu › **Refresh** › refresh the data displayed only on the current tabbed view

Tools menu › **Refresh All** › refresh the data on *all* the tabbed views

or use the **Refresh** and **Refresh All** icons on the Tools toolbar.

3.15.3 Loaded Modules

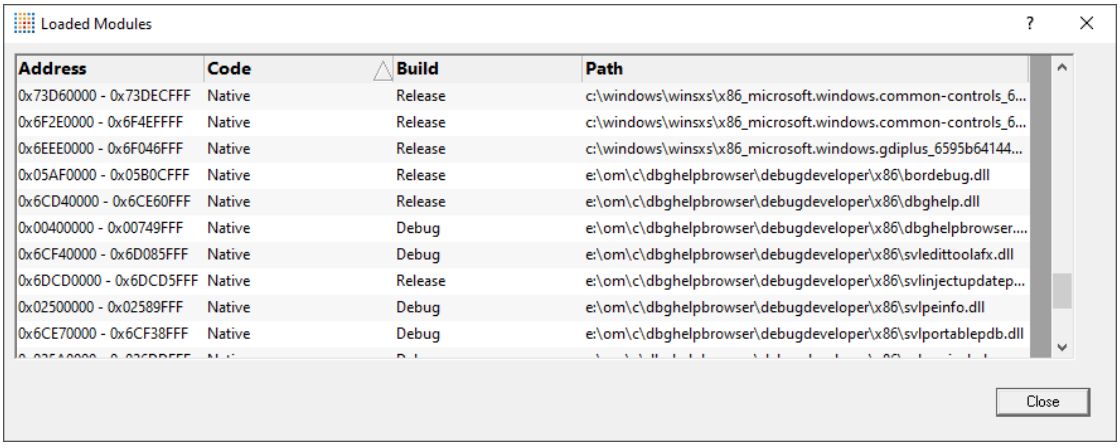
Viewing the loaded modules

You can view a list of the modules which are loaded by your target application.

Tools menu › **Loaded Modules...** › shows the Loaded Modules dialog

The dialog shows:

- the **Address** space occupied by the module (DLL or EXE)
- the type of **Code** in the module (native, managed, mixed mode or resources only
- the type of **Build** - is the code debug or release?
- the **Path** the module was loaded from



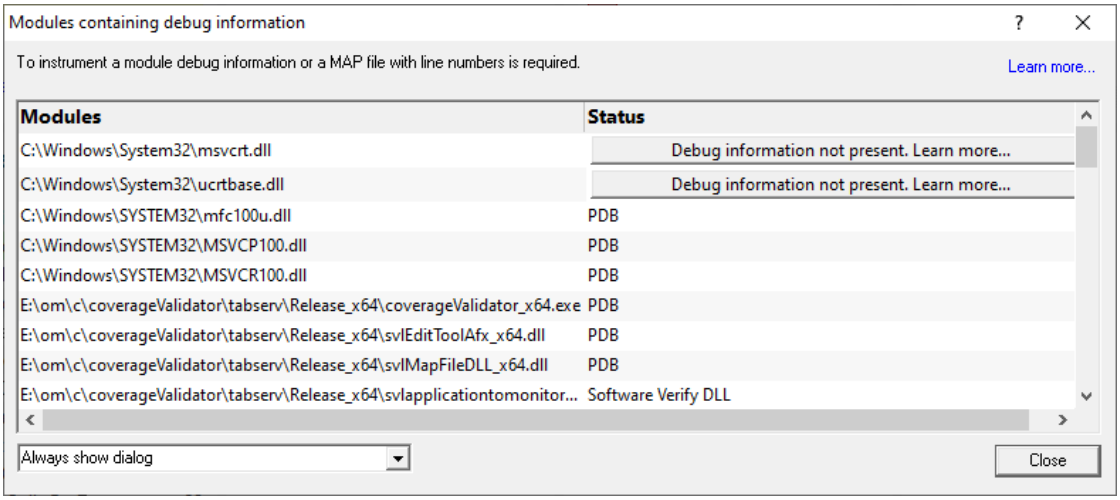
3.15.4 DLL Debug Information

Viewing the DLL debug information

If you are having problems collecting thread data for a particular EXE/DLL the problem may be that the debug information that is required to perform the instrumentation of the software cannot be found.

You can view a list of the debug information status of modules loaded by your target application.

Tools menu > **DLL Debug Information...** > shows the DLL Debug Information dialog below



The dialog shows:

- the path from which **Modules** (DLL or EXE) were loaded
- the debug **Status** (below)
- if any symbol server is not reachable (offline or doesn't exist) a message will be shown in red at the bottom of the dialog. You can edit the symbol server definitions here.

Debug status

There are various reasons why a module may not have its debug information read.

The dialog shows a comment or reason in the status column. Examples might be:

- PDB or MAP if the debug information *was* found and used
- Debug information not present

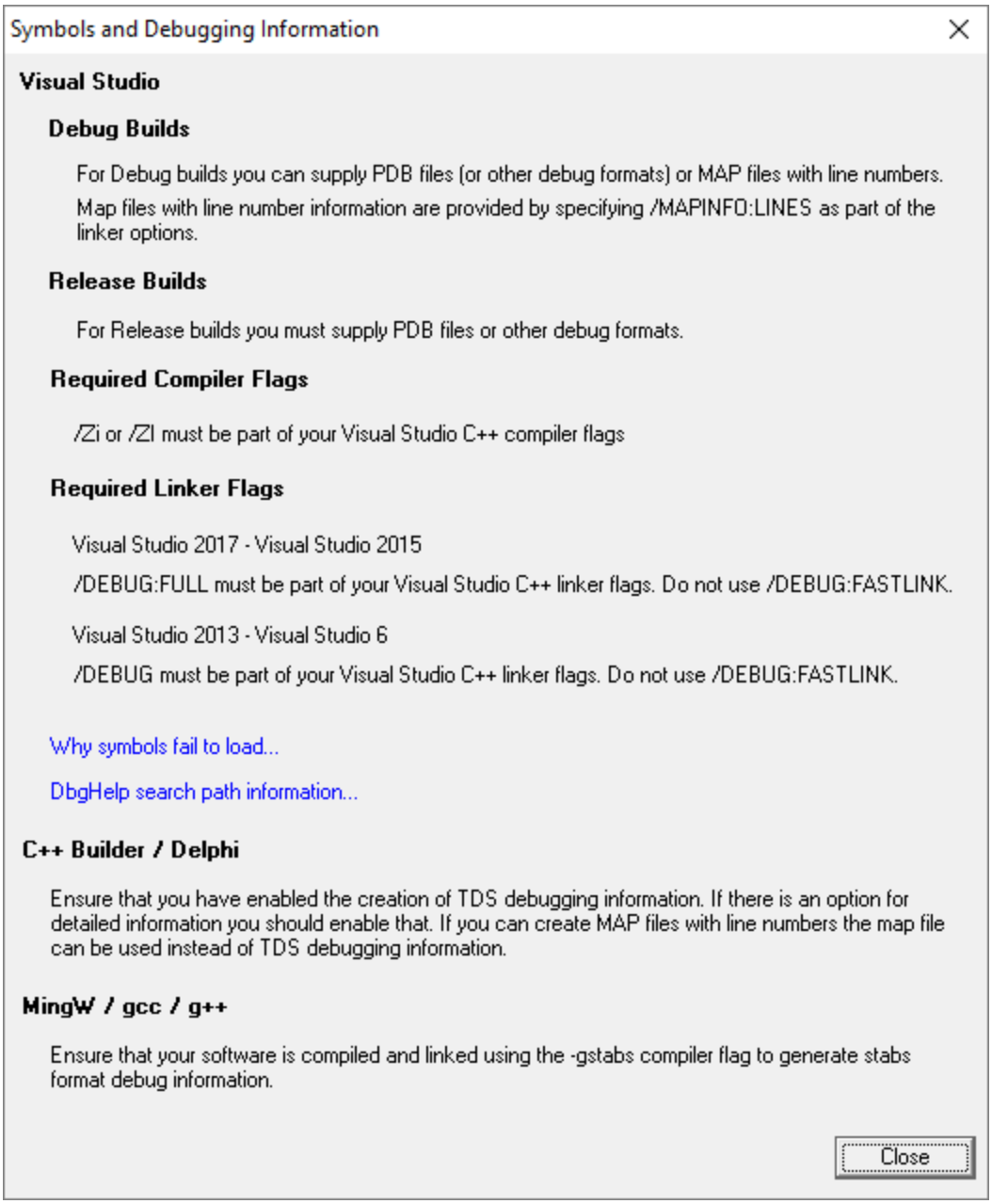
The status column may contain a button to enable you to find out more via the Debug Information Diagnosis dialog

- A reason for being ignored
- Module is a part of the C Run-time Library (CRT) or Standard Template Library (STL)
- Location is a system directory
- Ignored due to Hooked DLLs advanced settings
- File is a Software Verify own module
- Module has been specified as a 3rd party
- No executable code is contained
- The module only has GUI resources

More information about PDB and MAP files

Clicking on the **Learn more...** link at the top right of the dialog shows the Symbols and Debugging Information dialog (below) with more details with additional links to topics in this help.


Click the links below to read more in our frequently asked questions.

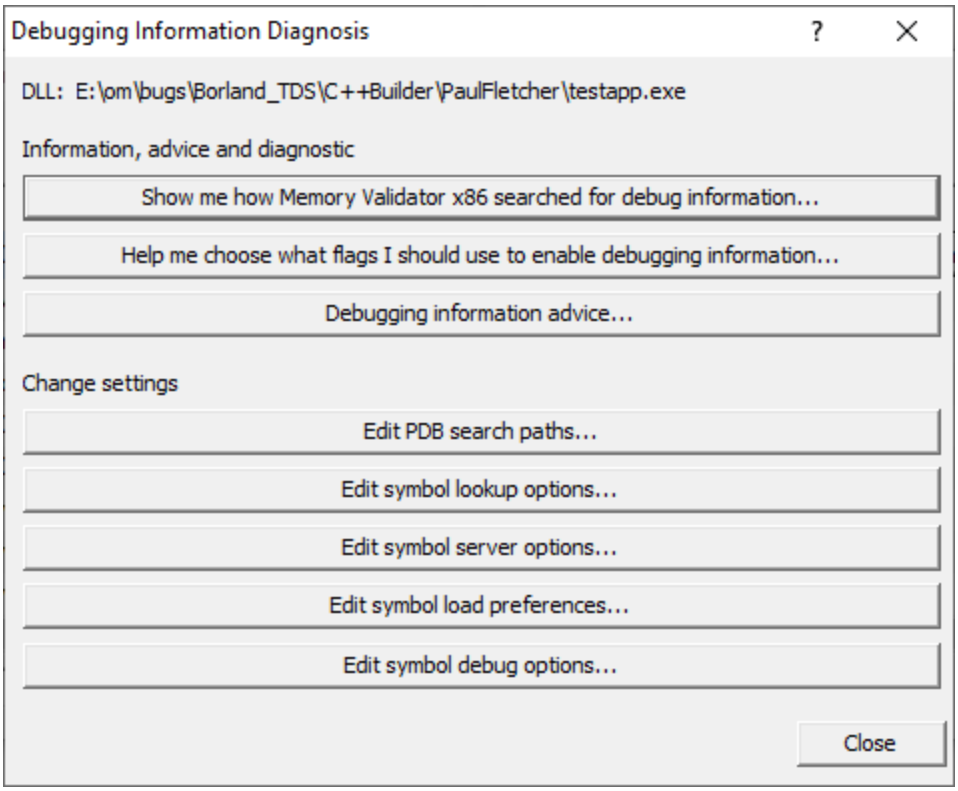


Finding out more using the Debugging Information Diagnosis Dialog

When debug information is not present for a given module the DLL Debug Information dialog (above) may display a button in the Status column to show the Debugging Information Diagnosis dialog.

The dialog shows:

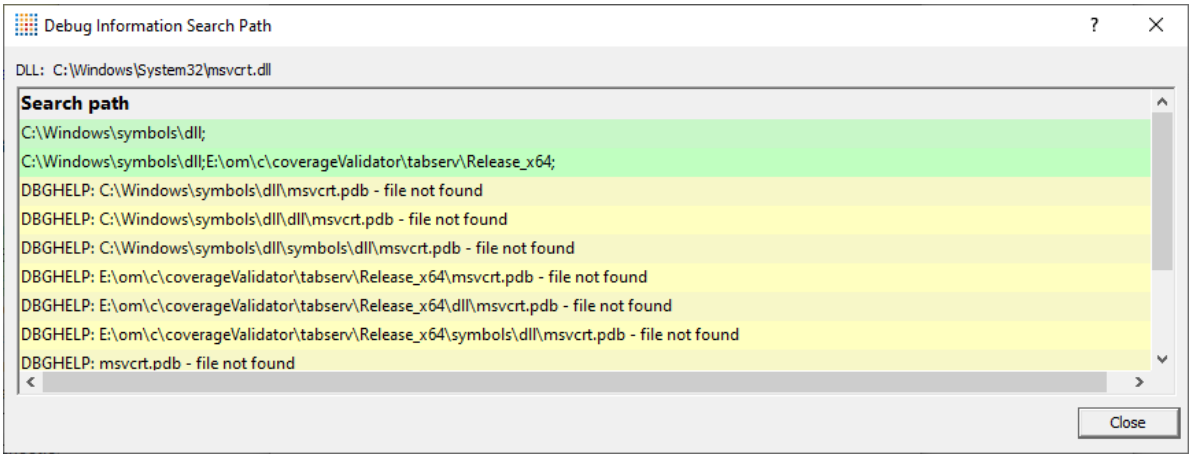
- Information, advice, and diagnostic help
- Quick links to change settings



The information options include:

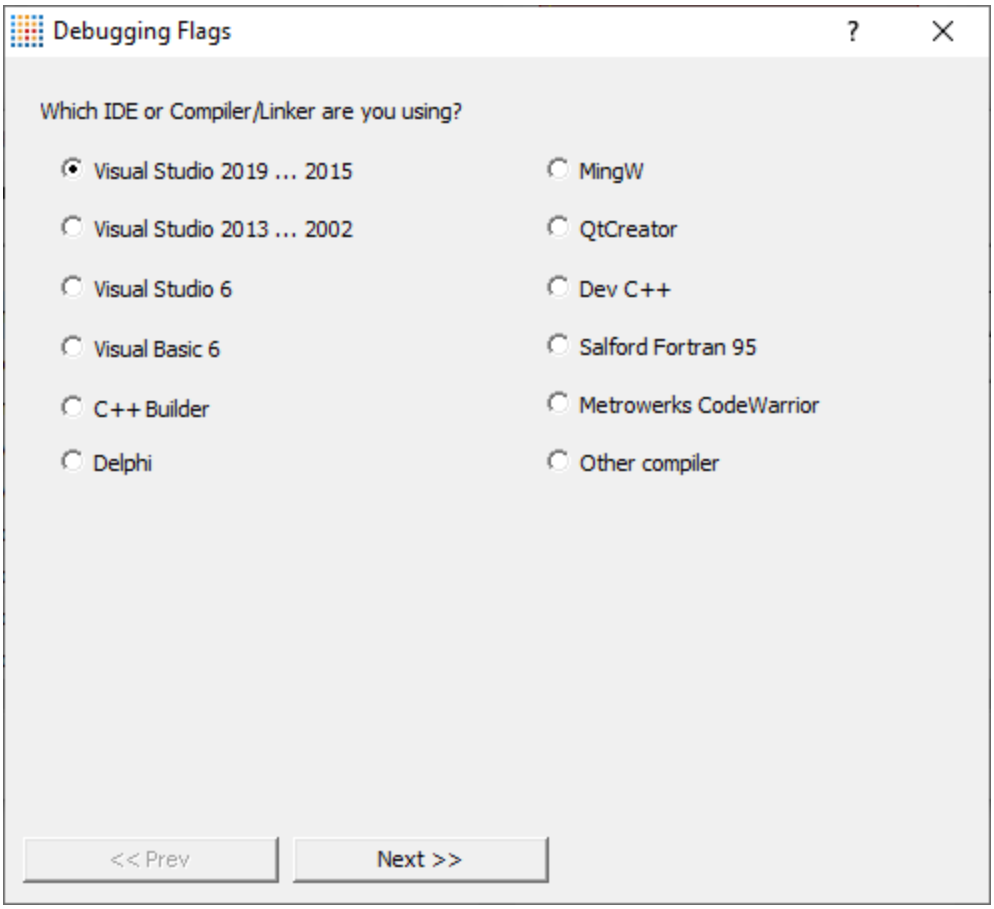
- **Show me how debug information was searched for...** › shows the Debug Information Search Path dialog

This information is extracted from the Diagnostic tab and shows only the relevant information for the module selected in the DLL Debug Information dialog.

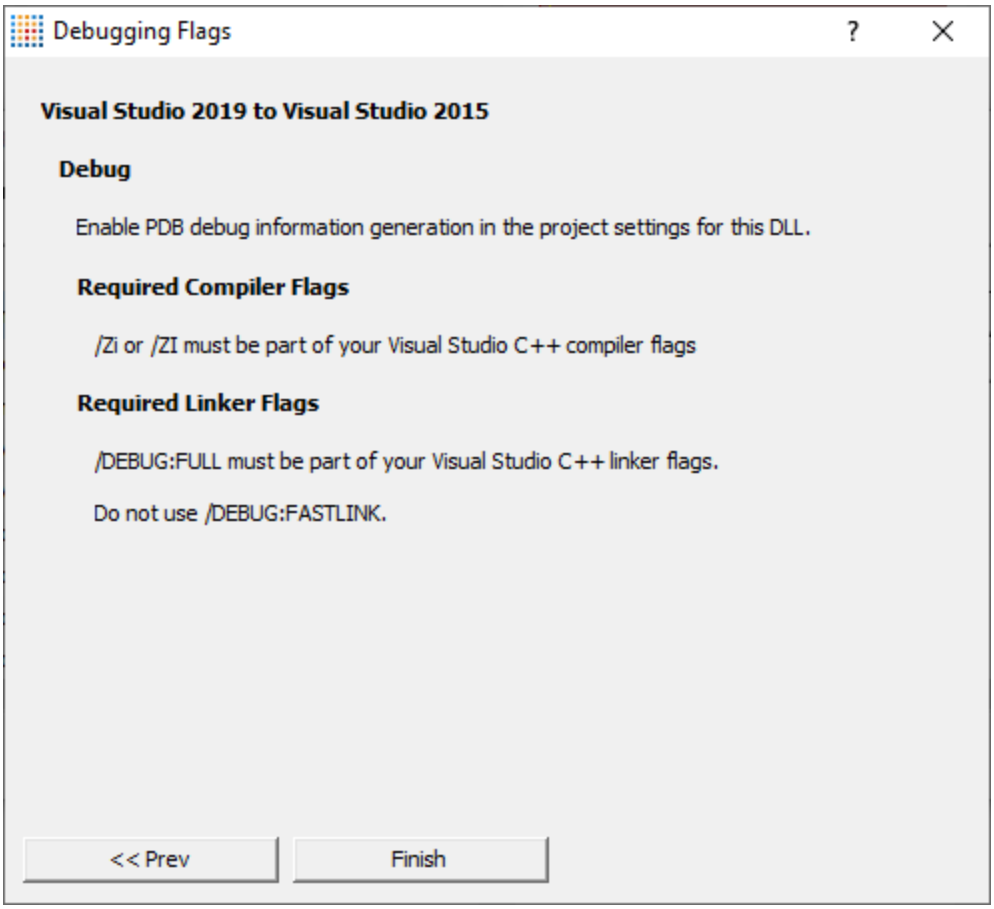


- **Help me choose what flags...** › shows the Debugging Flags wizard

Use the wizard to first select the compiler or linker you're using



Next >> › Provides the relevant debug compiler and linker flags. An example for Visual Studio 2017 to 2015 is below:



- **Debugging information advice...** › shows the Symbols and Debugging Information dialog above.

The options for changing settings include quick links to the following pages from the Global Settings Dialog

- **Edit PDB search paths...** › shows the File Location settings page for PDB files.
- **Edit symbol lookup options...** › shows the Symbol Lookup settings page
- **Edit symbol server options...** › shows the Symbol Servers settings page
- **Edit symbol load preferences...** › shows the Symbol Load Preferences settings page
- **Edit symbol debug options...** › shows the Symbol Misc settings page

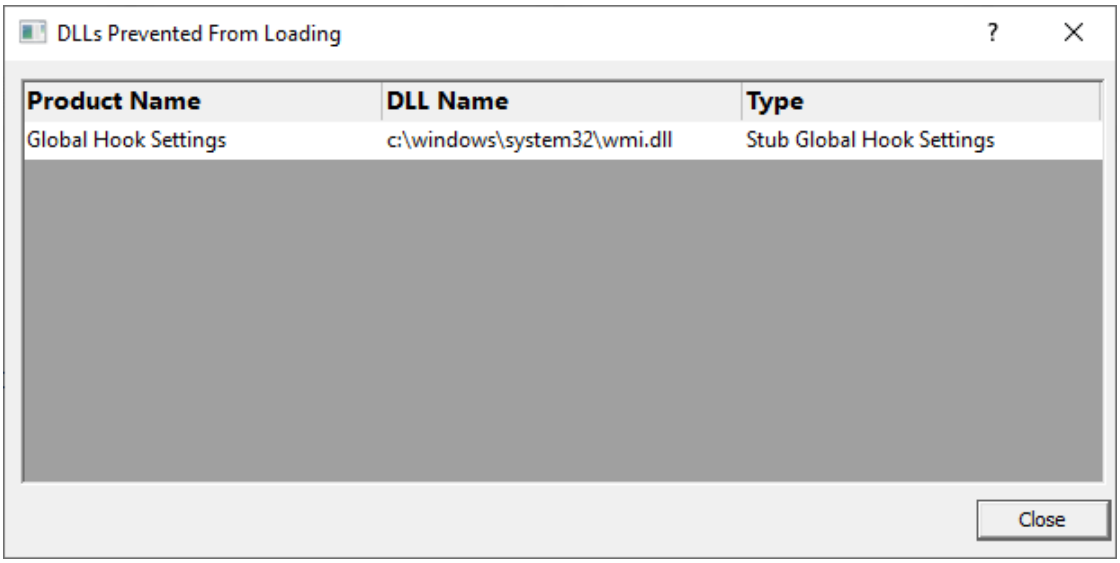
3.15.5 DLLs Prevented from Loading

Viewing the DLL prevented from loading information

If some DLLs are not loading via LoadLibrary() this dialog will show you any DLLs that Memory Validator has deliberately prevented from loading.

Memory Validator will prevent DLLs from loading that are known to be problematic. These are typically badly written shell extensions, or any DLL that uses Visual Leak Detector, or any DLL that is listed in the Stub Global Hook DLLs settings.

Tools menu > **DLLs Prevented From Loading...** > shows the DLL Debug Information dialog below



For each DLL the dialog shows:

- the Product Name associated with the DLL
- the DLL name
- the type (known bad DLL, Visual Leak Detector, Stub Global Hook DLLs)

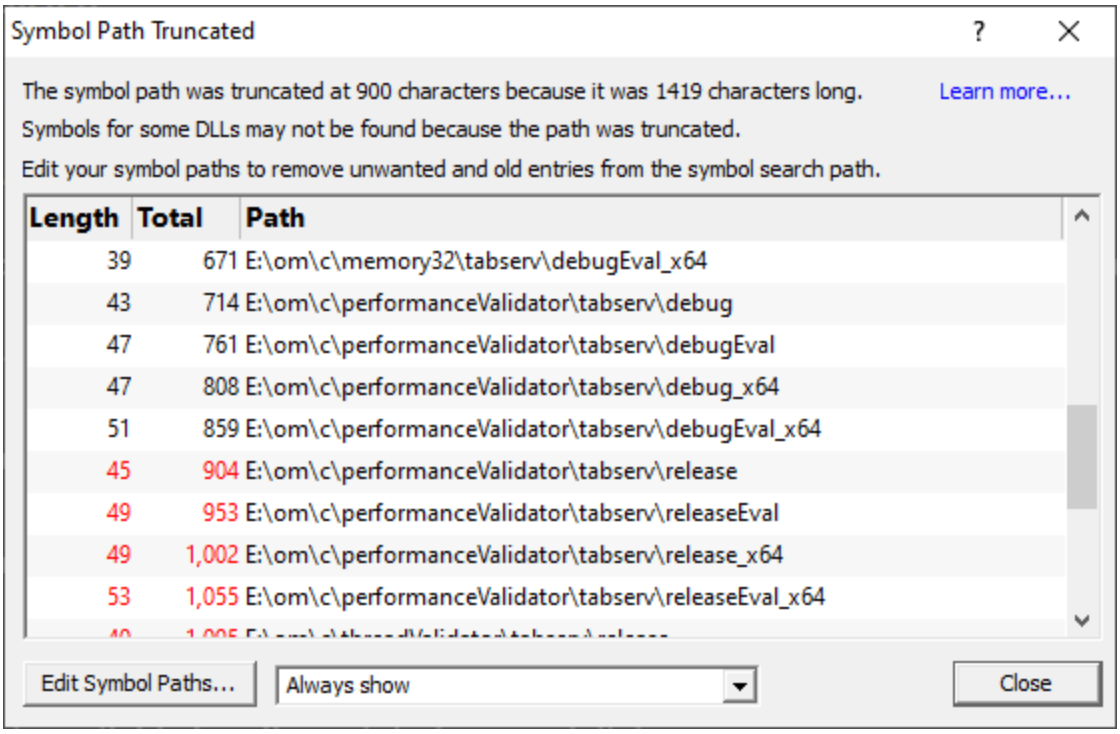
In normal use, most of the time you won't be able to access this dialog because there will be no data to display.

If you're seeing a notification on the Summary page, or the Tools menu it is enabled it's because we've identified a DLL trying to load into your process that is known to cause problems (a shell extension, or any DLL linked with Visual Leak Detector) or is a DLL that is listed on the Stub Global Hook DLLs settings.

3.15.6 Symbol Path Truncation

The Symbol Path Truncated warning dialog

The symbol path truncated warning dialog is displayed to warn you when the symbol path is too long.



- **Edit Symbol Paths...** › shows the file locations dialog so that you can edit the paths used for subsequent runs of the program.

You can choose when this dialog is displayed.

- **Always show** › The dialog is always shown when the symbol path is too long.
- **Show when path changes** › The dialog is shown when the symbol path is too long, but only if the symbol path is different than last time this warning was shown.
- **Never show** › The dialog is never shown.

Whether this dialog is displayed or not there is always a warning message written to the diagnostic window when the symbol path is truncated.

The display lists each path with it's length (including the unshown ';' path separator) and the total length so far so that you can see which paths exceed the truncation point (length and total displayed in red).

Any paths that don't exist on this computer are displayed in red.

Why is this dialog displayed?

You may see a Symbol Path Truncated warning dialog in some rare circumstances.

This dialog is displayed when the symbol path that has been calculated to pass to DbgHelp.dll to load Microsoft debugging symbols (found in .PDB files) is too long.

If the symbol path has been truncated because it is too long it is possible this may mean that some symbol searches will fail, resulting in failure to load some symbols. We display this dialog so that you

are aware that the symbol path is too long and would benefit from editing to make the symbol path shorter.

Passing a symbol path that is too long to DbgHelp.dll will cause the program being tested to end with an EXCEPTION_INVALID_CRUNTIME_PARAMETER C runtime error. This happens because internally DbgHelp.dll is using a fixed length array to format a string. To prevent this fatal termination of the test program we limit the length of the path passed to DbgHelp.dll.

Typically if a path that is long enough to cause this problem is passed to DbgHelp it's because the number of paths in the calculated path contain paths not relevant to finding symbols for the test program. We use the Symbol Path Truncated warning dialog to show you the calculated paths so that you can work out which paths to delete.

The calculated symbol paths come from several places:

- File locations PDB paths
- Symbol server symbol storage directories
- Symbol handling environment variables

Fixing the symbol path

For this example, we are testing the program **E:\om\c\3RD_SRC\cdplayer\Release\cdplayer.exe**

In the image shown above you can see that seven paths exceed the truncation limit, one of the 7 paths doesn't exist.

To work out what to do we need to do several actions:

1. Looking at the environment variable settings shows that none of the environment variables are being used. We do not need to consider the content of these environment variables.

2. Examining the symbol servers shows that **C:\Users\Admin** is a local symbol storage location. We should keep this path.

3. We should delete the path that doesn't exist: **E:\om\c\testApps\testStdinStdoutRedirectEx\Release**. We do this using the file locations dialog by clicking **Edit Symbol Paths...** then click **Delete invalid**.

4. Examining the paths in the file locations dialog we can identify any paths not relevant to the program we are testing. In this case the following paths are not relevant and can be deleted.

   E:\om\c\testApps\testStdinStdoutRedirect\Release
   E:\om\c\testApps\testAppTheReadsFromStdinAndWritesToStdout\Release
   E:\om\c\testApps\testSimpleMemoryLeak\Release
   e:\om\c\3rd_src\cppunit-1.12.1\examples\cppunittest\release
   e:\om\c\3rd_src\cppunit-1.12.1\examples\cppunittest\releasedll

3.15.7 Out Of Date DLLs

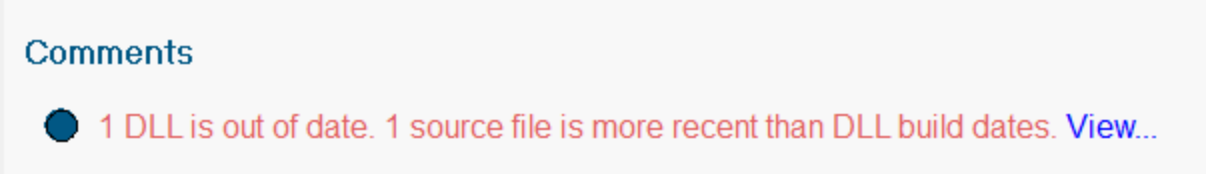
It may happen that if you forget to build a DLL, or if a build error occurs that you perform memory leak detection on DLLs that are not built with the most recent version of your source code.

We refer to these DLLs are out of date DLLs because they are out of date compared to the source code that is compiled to create the DLLs.

Memory Validator can detect this, and warn you about it.

Summary tab

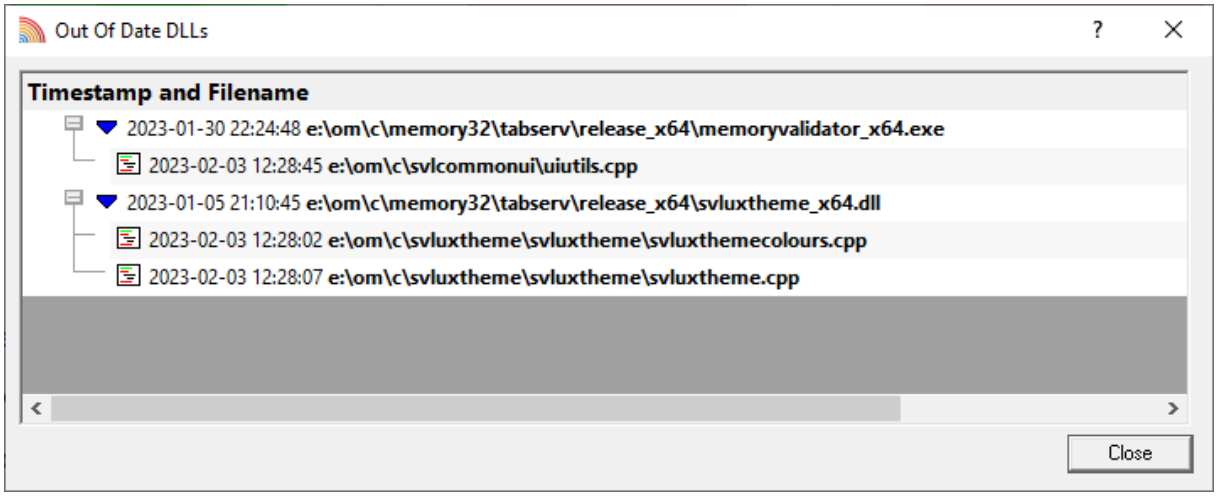
When out of date DLLs are found a warning is displayed on the summary tab, in the lower section of the display.



Out Of Date DLLs Dialog

The **View...** link will display the Out Of Date DLLs dialog.

There is also an option to display the Out Of Date DLLs dialog on the Tools menu.



The Out of date DLLs dialog shows the DLLs that are out of date, and the source files for each DLLs. The dates of both the DLLs and the source files are displayed.

The above image shows 1 DLL that is out of date, with 1 file being more recently edited than the build timestamp for the respective DLL.

3.15.8 Running totals

Viewing the running totals

In order to keep a collective watch on some general statistics, the Running Totals dialog keeps a count of the number of bytes and handles allocated.

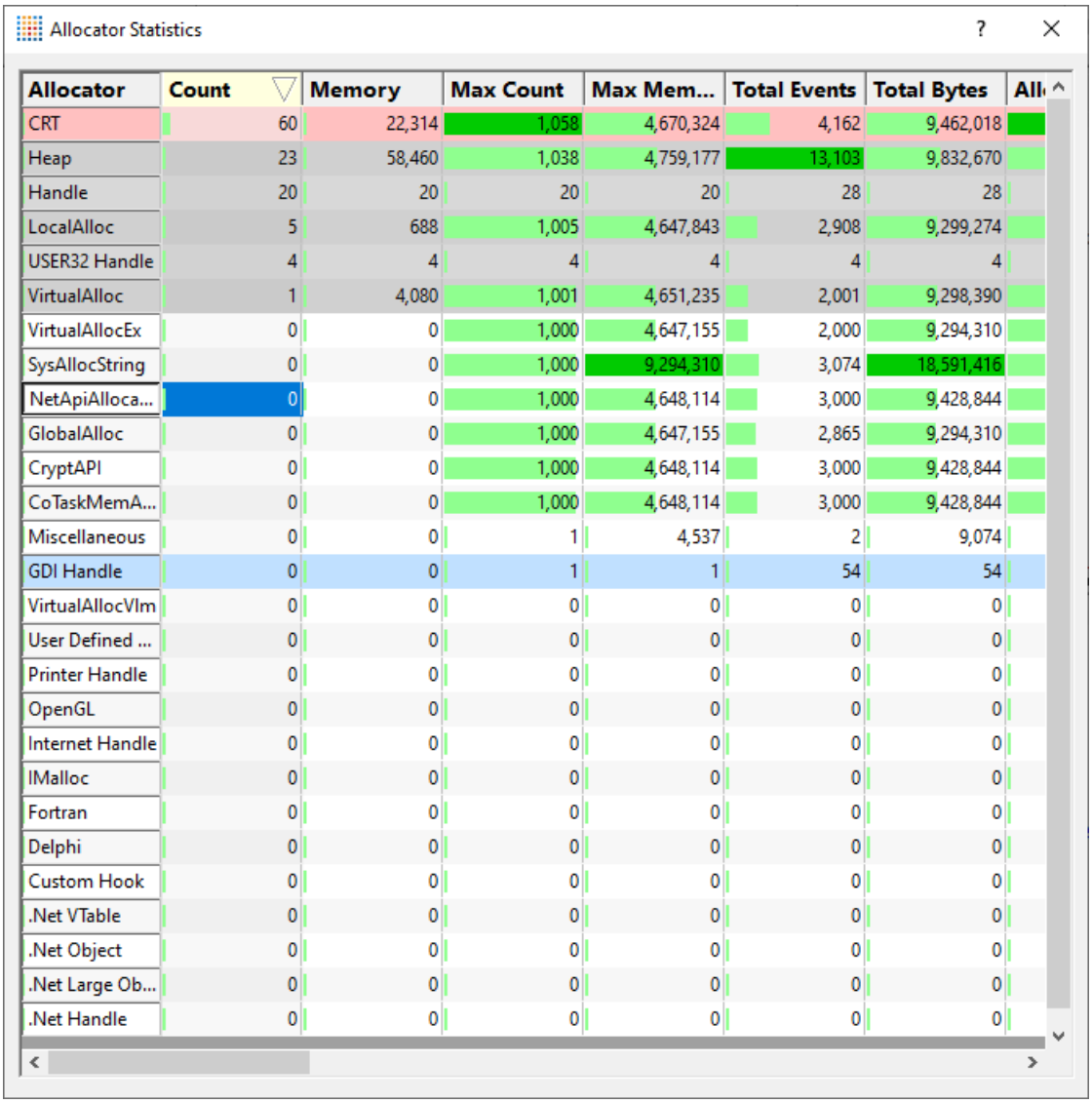
To show the Find Memory dialog, choose the menu option below:

Tools menu › choose **Running Totals...** › displays the Running Totals dialog

Or use the icon on the Tools toolbar.

The running totals dialog

The totals are grouped by related memory/handle allocator, described in the first column.



Allocator	Count	Memory	Max Count	Max Mem...	Total Events	Total Bytes	All
CRT	60	22,314	1,058	4,670,324	4,162	9,462,018	
Heap	23	58,460	1,038	4,759,177	13,103	9,832,670	
Handle	20	20	20	20	28	28	
LocalAlloc	5	688	1,005	4,647,843	2,908	9,299,274	
USER32 Handle	4	4	4	4	4	4	
VirtualAlloc	1	4,080	1,001	4,651,235	2,001	9,298,390	
VirtualAllocEx	0	0	1,000	4,647,155	2,000	9,294,310	
SysAllocString	0	0	1,000	9,294,310	3,074	18,591,416	
NetApiAlloca...	0	0	1,000	4,648,114	3,000	9,428,844	
GlobalAlloc	0	0	1,000	4,647,155	2,865	9,294,310	
CryptAPI	0	0	1,000	4,648,114	3,000	9,428,844	
CoTaskMemA...	0	0	1,000	4,648,114	3,000	9,428,844	
Miscellaneous	0	0	1	4,537	2	9,074	
GDI Handle	0	0	1	1	54	54	
VirtualAllocVlm	0	0	0	0	0	0	
User Defined ...	0	0	0	0	0	0	
Printer Handle	0	0	0	0	0	0	
OpenGL	0	0	0	0	0	0	
Internet Handle	0	0	0	0	0	0	
IMalloc	0	0	0	0	0	0	
Fortran	0	0	0	0	0	0	
Delphi	0	0	0	0	0	0	
Custom Hook	0	0	0	0	0	0	
.Net VTable	0	0	0	0	0	0	
.Net Object	0	0	0	0	0	0	
.Net Large Ob...	0	0	0	0	0	0	
.Net Handle	0	0	0	0	0	0	

Each group of statistics has the following values:

- **Count** › the number of memory allocations or handles *currently in use* at this time
- **Memory** › the amount of memory or handles *currently in use* at this time
- **Max Count** › the maximum number of memory allocations or handles *in use at any time*
- **Max Memory** › the maximum amount of memory or handles *in use at any time*
- **Total Events** › the total number of memory allocations, reallocations, deallocations or handle allocations, handle deallocations *until this point in time*
- **Total Bytes** › the total amount of memory allocations, reallocations, deallocations or handle allocations, handle deallocations *until this point in time*
- **Alloc Count** › the number of memory or handles that have been allocated

- **Alloc Size** › the size of memory or handles that have been allocated
- **Alloc Max Count** › the maximum number of memory or handles that have been allocated
- **Alloc Max Size** › the maximum amount of memory or handles that has been allocated
- **Alloc Cumulative Count** › the total number amount of memory or handles that have been allocated
- **Alloc Cumulative Size** › the total amount of memory or handles that have been allocated
- **Realloc Count** › the number of memory or handles that have been reallocated
- **Realloc Size** › the size of memory or handles that have been reallocated
- **Realloc Max Count** › the maximum number of memory or handles that have been reallocated
- **Realloc Max Size** › the maximum amount of memory or handles that has been reallocated
- **Realloc Cumulative Count** › the total number amount of memory or handles that have been reallocated
- **Realloc Cumulative Size** › the total amount of memory or handles that have been reallocated
- **Free Count** › the number of memory or handles that have been deallocated
- **Free Size** › the size of memory or handles that have been deallocated
- **Free Max Count** › the maximum number of memory or handles that have been deallocated
- **Free Max Size** › the maximum amount of memory or handles that has been deallocated
- **Free Cumulative Count** › the total number amount of memory or handles that have been deallocated
- **Free Cumulative Size** › the total amount of memory or handles that have been deallocated

If you see a **negative Handle Total**, then more handles have been detected as deallocated than allocated.

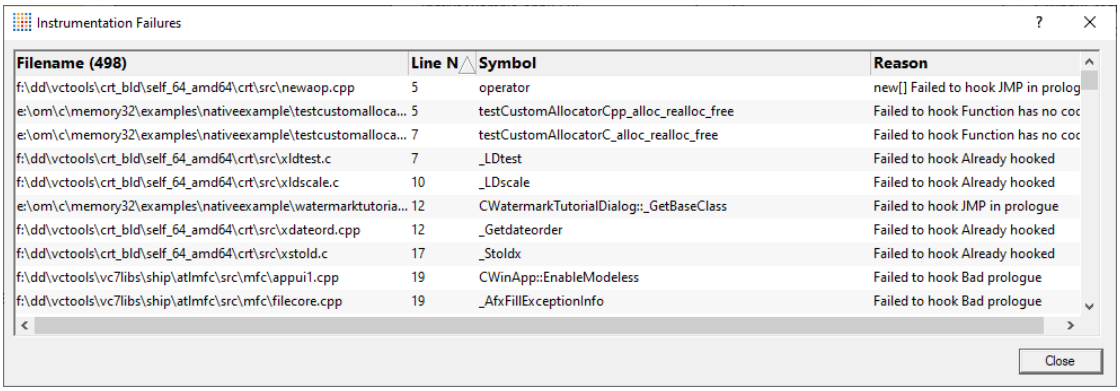
3.15.9 Instrumentation Failure Data

Instrumentation Failure Data

It can be very useful to know which functions in your code failed instrumentation.

You can view a list of the functions that have failed instrumentation via the Tools menu.

Tools menu › **Instrumentation Failure Data...** › shows the Instrumentation Failures dialog



The dialog shows:

- the file name of the item that hasn't been instrumented
- the line number of the item that hasn't been instrumented
- the symbol name of the item that hasn't been instrumented
- the reason why each item wasn't instrumented

Example reasons why an item might not be instrumented include the following:

- Disallow computed unconditional jmp
- Failed to disassemble
- Found privileged instruction

3.15.10 Memory leak and handle leak detection

Leak detection

If needed, you can perform *in-place* leak detection on memory and handles whilst your application is running.

The Leak Detect dialog lets you search for leaks between watermarks or at specific addresses to help narrow the search for large datasets.

In-place memory leak detection can be time consuming so it's usually more efficient to leave Memory Validator to analyze the collected data once your application has closed normally.

➡ See the recommended leak detection process at the bottom of this page.

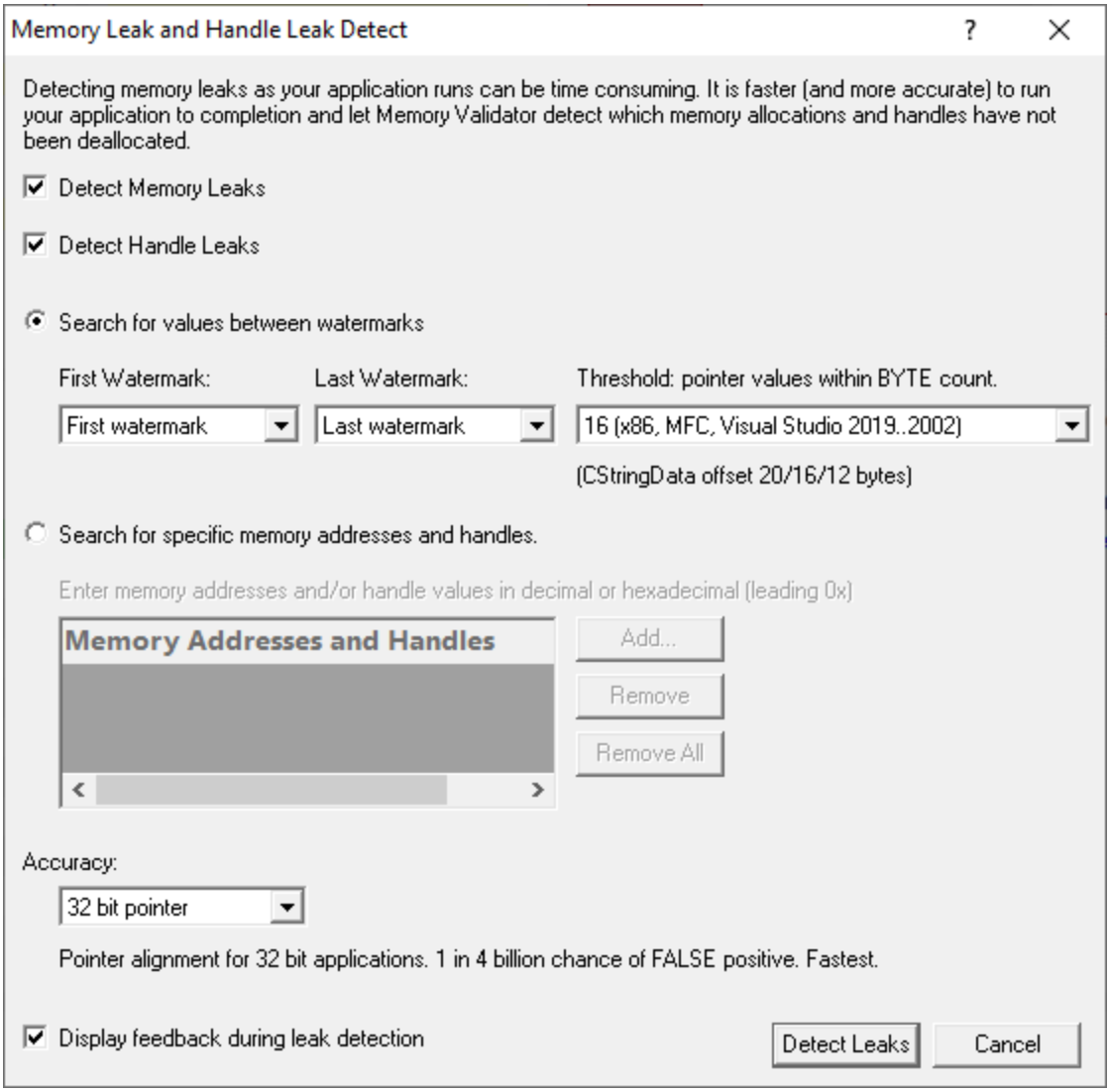
Detecting leaks

To show the Leak Detect dialog, choose the menu option below:

Tools menu › choose **Leak Detect...** › displays the Leak Detect dialog

Or use this icon on the Tools toolbar.

The leak detect dialog



What to search for

The default is to search for both memory leaks and handle leaks

- **Detect Memory Leaks** › detect memory leaks

- **Detect Handle Leaks** › detect handle leaks

You can then choose to search between watermarks or between addresses.

Searching between watermarks

Watermarks are points in the allocation event history that you can mark and use as endpoints for showing data or performing searches

- **Search for values between watermarks** › choose a first and last watermark to search between
- **First Watermark** › set watermark as start marker
- **Last Watermark** › set watermark as end marker

A typical process might be:

- set a watermark
- perform an action in your application that you know is self contained and should not leak
- set another watermark
- search for leaks between the two watermarks

Address threshold searching

When searching between watermarks, (i.e. not searching for a specific address) you can specify a threshold around each memory address.

This is useful because for some data structures, the stored pointer is not the *actual* allocation memory address.

One example of this is the MFC CString object which uses an internal object CStringData which is allocated with a N byte header which stores information about the CStringData.
The CStringData offset varies with Visual Studio version and processor bit depth:

Visual Studio	CStringData offset
x86 VS 6	12
x86 VS 2002..2019	16
x64 VS 2008..2019	20

The pointer that is stored as **m_pchData** in the CString object is a pointer to the Nth byte in the allocation.

For the reason above, the default is 16 for x86 and 20 for x64, but you can change this:

- **Threshold: pointer values within BYTE count** › set the address threshold between 0 and 63 bytes

    If a pointer is found within the threshold range, it's considered a match.

At least 16 bytes is recommended for MFC applications, but you can change this if not using MFC:

Searching at specific addresses

- **Search for specific memory addresses and handles** › use address or handle ids to search for leaks

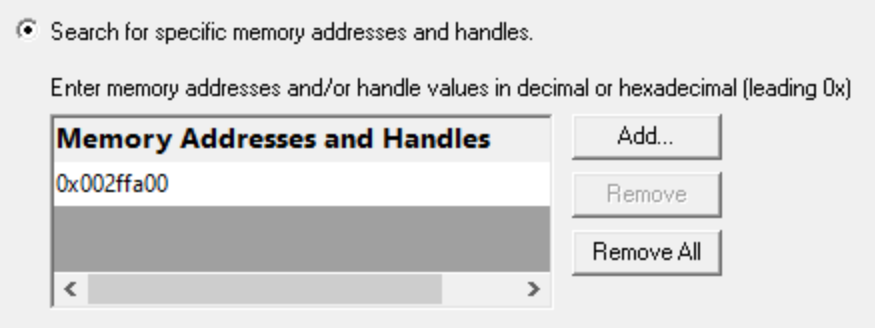
The list should be edited to add the memory addresses or handle ids you want to search for.

- **Add** › adds a new item to the list › **enter** the address or id in the new list item

  Addresses can be specified in decimal or hexadecimal with a leading 0x and must be positive.

- **Remove** › removes a selected item
- **Remove All** › removes all items

  The list is also cleared each time the dialog is shown



Search Accuracy

Most applications will have pointers and handles stored in memory aligned on 4 byte (DWORD) boundaries.

Some Windows datatypes are stored more compactly, and align data on 1 byte and 2 byte (WORD) boundaries.

Your application may have its own datatypes which align data on 1 and 2 byte boundaries.

Depending on the datatypes involved and how they are stored internally, you may wish to specify more accurate address address checking than the 4 byte DWORD alignment that is used by default.

- **BYTE boundaries** › checks addresses and handles on 1 byte boundaries
- **WORD boundaries** › checks 2 byte boundaries
- **32 bit pointer boundaries** › checks 4 byte boundaries

- **64 bit pointer boundaries** › checks 8 byte boundaries. *This option is only available with Memory Validator x64.*

Byte boundary checking will be slower and possibly find more false positive matches for leaked memory.

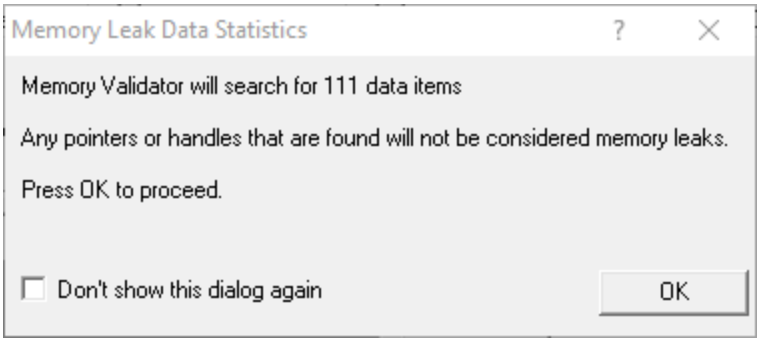
Handles are the same size as pointers on 32 bit Windows (4 bytes) and 64 bit Windows (8 bytes).

Starting the leak detection

- **Detect Leaks** › starts the detection process
- **Don't Detect Leaks** › abandon and close the dialog

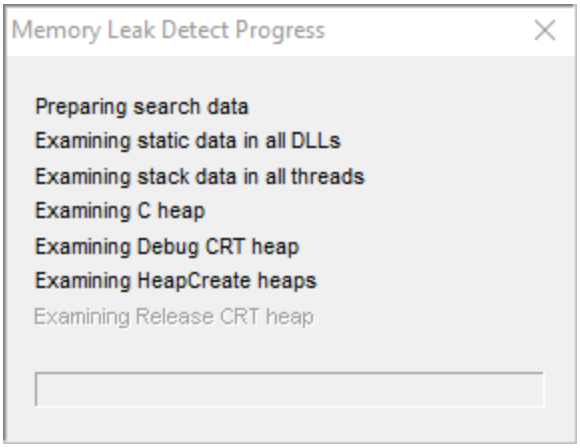
When searching between watermarks mode, a list of memory allocations and handle allocations (as appropriate) is prepared.

A message dialog shows the number of data items to be checked:



When searching for specific values, no such dialog is displayed as the number of data items is obvious.

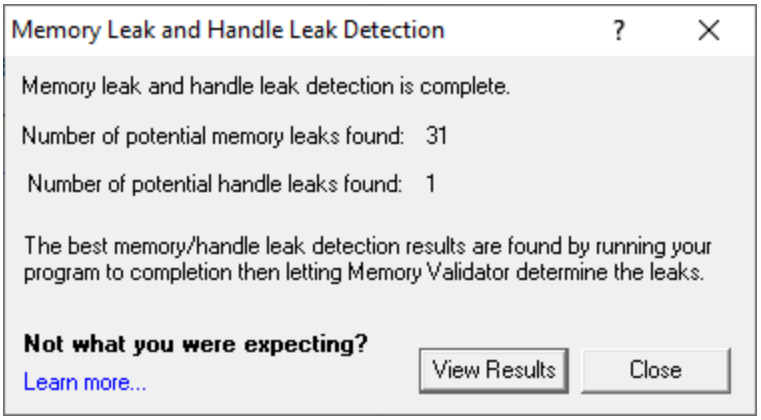
A progress dialog indicates what's being examined:



In place leak detection can take a long time.

Memory leak search results

When the search is complete a status dialog is displayed.



- **View Results** › displays the search results on the Memory and Handle tab.

If pointers to memory can be found in memory or handles can be found in memory they will not be regarded as leaked. Because of this fewer items may be reported as leaked than you may expect. For more information see Why doesn't in-place leak detection always find leaking objects?

Memory leak search results (searching between watermarks)

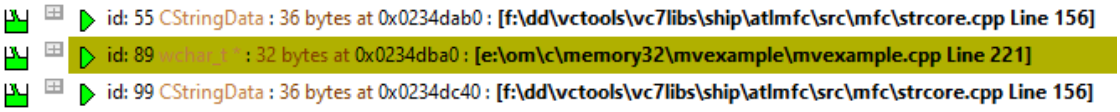
When searching between watermarks is complete, any objects that were determined to have leaked are display in the Memory view.

These objects are given the status 'potential memory leak', and displayed in the relevant colour.

The reason for being a 'potential' memory leak is because the address pointer may (depending on your application) have been

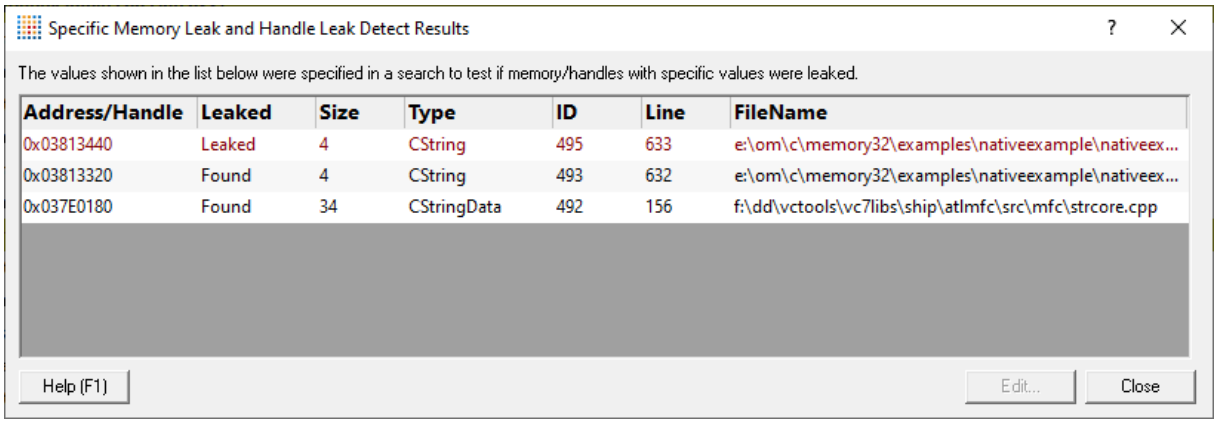
- stored in encrypted format
- offset (as described for CStringData above)
- altered in some other way prior to being stored.

An example portion of the display of potential memory leaks (after using the example application) is shown below:



Memory leak search results (searching for specific values)

When searching for specific values is over, the results are displayed in the Memory Leak Detect dialog:



The dialog displays the results in a list showing:

- the memory address or handle value
- if the specified value was found in the application, or was leaked
- the object identified as using the specified address or handle value

  This may be blank if no object relating to the value could be determined.

- **Edit...** (or double clicking any row) › displays the relevant source code in your preferred editor

  This is only enabled when you select a result in the list

Why doesn't in-place leak detection always find leaking objects?

When you run the in-place memory leak detection method the only way to detect that memory has leaked (pointer A, let's say) is to search all of your program's memory for pointer A.

If pointer A is not found we can regard that memory as leaked.

However, if pointer A *is* found anywhere (dynamically allocated memory, static memory, stack variables, function parameters) the pointer must be regarded as valid and cannot be marked as leaked.

The pointer may still be in memory but the program may not be aware of it (so in reality it is leaked) but because it is in memory our scan can't know that the program has forgotten about the pointer.

Consider the following example:

Show a C++ example (showing some bad programming):

```
class container
{
public:
 container();

 ~container();

 void flush();

 void addData(DWORD        value);

private:
 DWORD        *data;
 DWORD        len;
};

container::container(DWORD        value)
{
    data = NULL;
    len = 0;
}

container::~container()
{
    flush();
}

void container::flush()
{
    // whoops, forgot to free the data here!

    len = 0;
}

void container::addData(DWORD        value)
{
    len++;
    if (data == NULL)
         data = (DWORD *)malloc(sizeof(DWORD) * len);
    else
         data = (DWORD *)realloc(data, sizeof(DWORD) * len);
```

```
    if (data != NULL)
         data[len - 1] = value;
}

static container        testObj;

static void test()
{
    testObj.addData(1);
    testObj.addData(2);
    testObj.addData(3);

    testObj.flush();
}
```

The example uses a simple class `container` to store some data which has several methods.

The `flush` method

`container` uses a method `flush()` to clear out all data.

`flush()` can be called at arbitrary times to reset the data.

It is also called from the destructor to ensure that the class cleans up after itself.

However, the `flush()` method contains a bug: it should call `free(data)`, but doesn't - and so this results in a memory leak.

The `test` method

If we examine the function `test()` we can see it calls `addData()` three times which results in memory being allocated inside the `container` object.

Finally `flush()` is called to discard the data, causing a memory leak.

The memory pointer is not freed but is also not *reset* which means that the pointer data in the object `testObj` is still pointing to the memory that has been leaked.

This pointer will be found when the in-place memory leak detector scans memory and so *the memory will not be regarded as leaked*.

 This memory *will* be reported as leaked *when the program finishes executing* and C++ Memory Validator can reconcile all memory allocations with memory deallocations.

The recommended leak detection process

If in doubt about memory leak detection always use the following methodology:

1. Start your program with C++ Memory Validator
2. Run your program as normal, executing the parts of your program you want to test for memory leaks.
3. Close your program
4. Wait for C++ Memory Validator to analyze all memory allocations and memory deallocations
5. Examine the memory leak report on the Memory tab.

3.15.11 Uninitialised memory detector

Uninitialised memory

Memory Validator can detect potentially uninitialized data by looking for memory allocations that have the Microsoft® uninitialized data signature in them.

The debug C runtime heap initialises all allocated memory with a signature byte of 0xCD.

Bit patterns in memory are sought out that match the default values:

- `0xCD` for BYTEs
- `0xCDCD` for WORDSs
- `0xCDCDCDCD` for 32 bit pointers (and DWORDs)
- `0xCDCDCDCDCDCDCDCD` for 64 bit pointers (and QWORDs)

Any objects found with the signature are displayed in the Memory tab which is updated to display the allocations with uninitialised memory.

 Uninitialized data detection is only available when your program is compiled in _DEBUG mode.

➜ See the global settings for checking uninitialised data.

False positives

It may not be that uncommon to find a byte that has actually been deliberately initialised to `0xCD` by the program - a 1 in 256 chance for random data for example.

It's less likely to find words similarly initialised to `0xCDCD`, very unlikely to find such DWORDS (1 in 4 billion for random data), and exceptionally unlikely to find such QWORDS (1 in 16 quintillion for random data).

Because of this, you should use your own judgement as to whether reported allocations are false positives, based on what you know about the behaviour of your program.

Invoking a memory initialisation check

Use one of the following methods during a session to start uninitialized memory detection:

Tools menu > **Check Initialised** > performs the check

or click the **Check Initialised** icon on the session toolbar.

3.15.12 Integrity checker

Integrity checks

An integrity check is an examination of the C runtime heap that looks for blocks that are damaged or have errors.

Any errors found in the heap are communicated to the user.

Internally the status for any memory blocks having errors is updated so that the user interface displays such blocks as damaged.

Invoking an integrity check

Use one of the following methods during a session to invoke the CRT heap integrity check:

Tools menu > **Integrity Check** > performs the check

or use the **Integrity Check** icon on the Tools toolbar:

3.15.13 Update information

Why is Update needed?

When Memory Validator attaches to a target program, some memory allocations may have been made already.

Although those allocations are typically in the very first startup code executed as the C runtime initialises, Memory Validator won't know their location.

This is true even when Memory Validator launches, waits for, or is linked to the target program.

Where the program has been running for a while and Memory Validator is *injected* into the program, then clearly *a lot* of allocations could have occurred already!

In most cases it's not important to know about the C runtime initialisation memory, as the C runtime will deallocate the memory when the program shuts down.

However, if you need Memory Validator to know about this memory you can use this update function to cause the stub to send information about all C runtime memory allocations to the Memory Validator user interface.

These allocations won't have a stack trace to identify where the memory was allocated, but for the _DEBUG versions of the C runtime, there may be useful filename and line information.

Invoking a CRT memory update

Use one of the following methods to invoke an update:

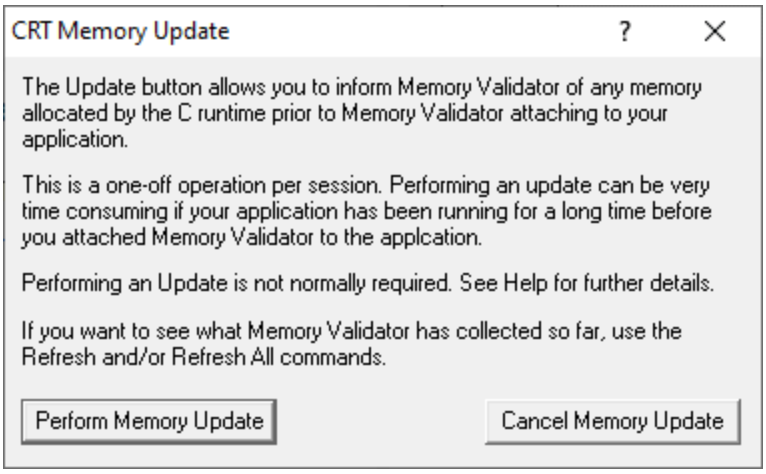
Tools menu > **Update** > shows the CRT Memory Update dialog

or use the **Update** icon on the Tools toolbar:

Update is a one-off operation in each session. The option is disabled after the first time.

The CRT Memory Update dialog

A confirmation dialog appears with a notice before the update proceeds.



3.15.14 Send command to stub extension DLL

This utility is only relevant if you have first built a Memory Validator stub extension and then specified a stub extension DLL in the settings.

It allows you to send a command to one or all user supplied stub extension DLLs, for whatever custom purpose you may need.

➜ See the example stub extension DLL provided with Memory Validator.

Sending a command to a stub extension DLL

Use one of the following methods to access the dialog:

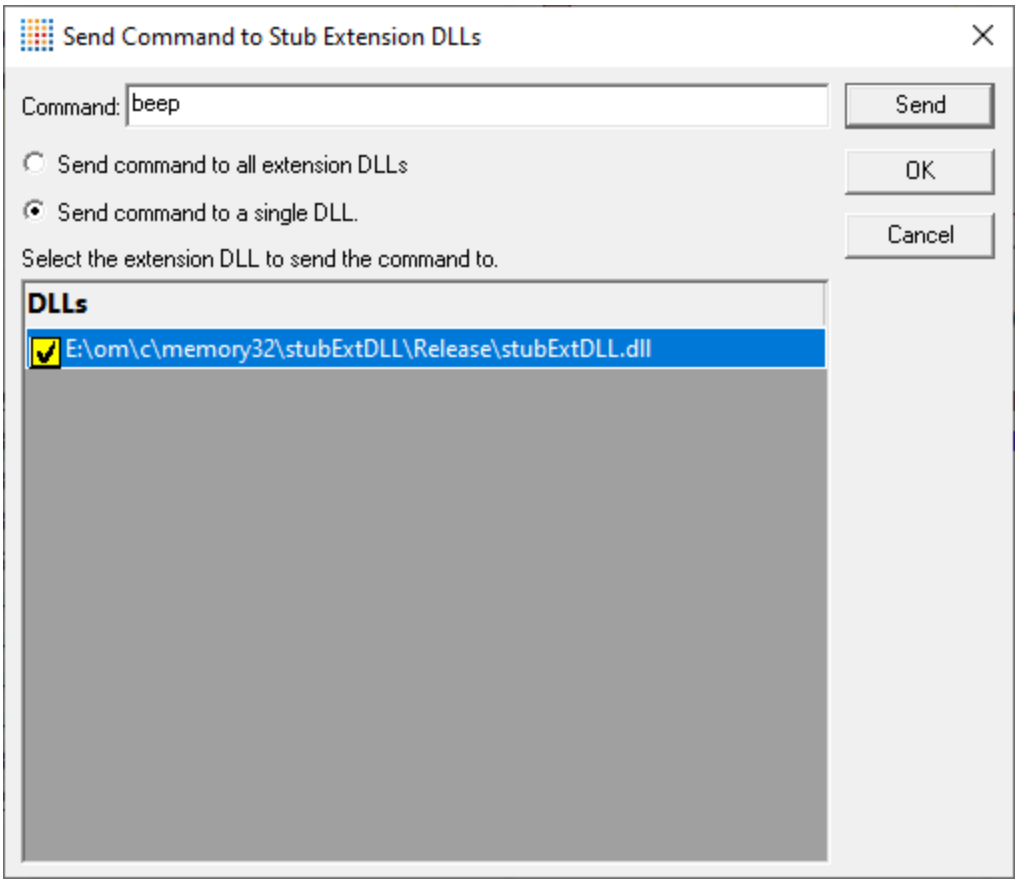
Tools menu › **Send Command To Stub...** › shows the Send Command to Stub Extension DLLs dialog

or use the **Send Command To Stub** icon on the Tools toolbar.

These options will be disabled if there are no stub extensions specified in the settings.

The Send Command to Stub Extension DLLs dialog

The dialog appears similar to the one below, letting you type in a message to be sent to one or all stub extension DLLs.



- **Command** › type the command to send to the extension DLL(s)
- **Send command to all extension DLLs** › will send the command to all the known stub extension DLLs
- **Send command to a single DLLs** › only sends the command to a stub extension DLL selected from the list

  The list is only enabled once you select this option.
- **Send** › immediately sends the message you entered

If you're using the example stub extension DLL provided with Memory Validator, the command `beep` would emit the system beep sound.

3.16 .Net Tools

Click on an item in the picture of the .Net Tools Menu below to jump to the relevant topic:

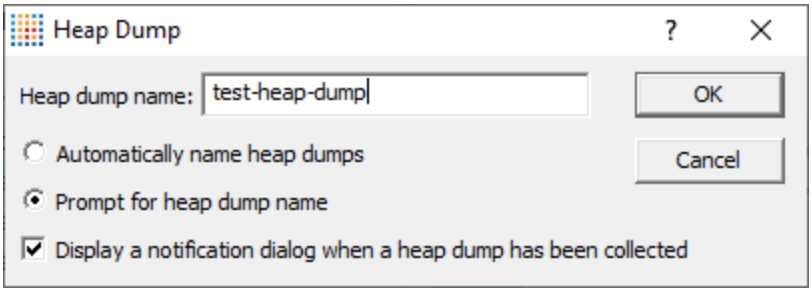
3.16.1 Heap Dump

To cause a .Net heap dump, choose the menu option below:

.Net Tools menu › choose **Heap Dump...** › displays the Heap Dump dialog

Or use the heap dump icon on the .Net Tools Toolbar.

The heap dump dialog is displayed.

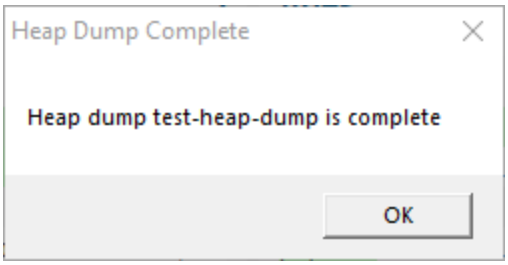


Heap Naming

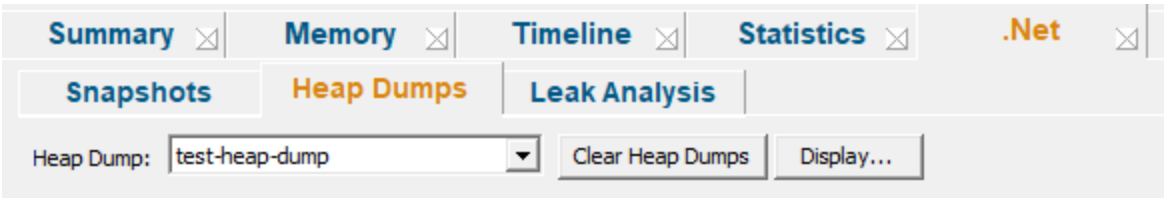
- **Automatically name heap dumps** › all heap dumps are automatically named
- **Prompt for heap dump name** › this dialog is displayed. The heap dump name is specified in the edit field.
- **Display a notification dialog** › when the heap dump is complete a dialog box is displayed

Heap Dump Complete

When the heap dump is complete a notification is displayed if it has been requested.



An entry for the heap dump is also added to the combo box on the Heap Dumps sub-tab of the main .Net tab.



Any attempt to fetch a new heap dump will fail until enough objects have been allocated to allow a garbage collection to run.

3.16.2 Garbage Collect

To cause a garbage collection to happen, choose the menu option below:

.Net Tools menu › choose **Garbage Detect** › the target program will cause a .Net garbage collection to happen

Or use the recycle icon on the .Net Tools Toolbar.

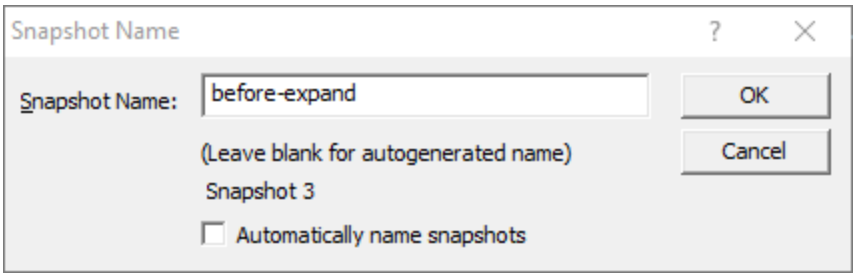
3.16.3 Snapshots

To create a .Net memory snapshot, choose the menu option below:

.Net Tools menu › choose **Snapshot...** › displays the Snapshot dialog

Or use the camera icon on the .Net Tools Toolbar.

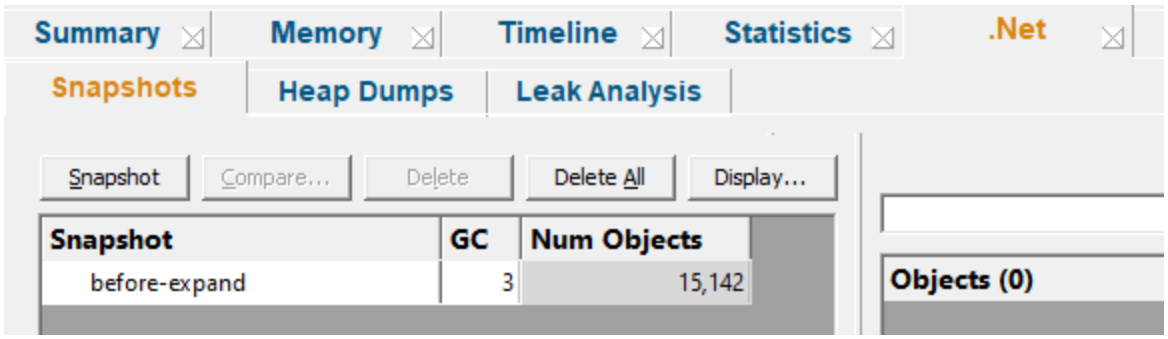
The snapshot dialog is displayed.



Snapshot Naming

- **Automaticlly name snapshots** › all snapshots are automatically named
- **Snapshot name** › enter the name for this snapshot. If you leave it empty a name will be created for this snapshot.

The snapshot is created an an entry for the snapshot is added to the list on the Snapshots sub-tab of the main .Net tab.



3.17 Sessions: Load, Save, Export, Close

Working with sessions

Sessions with Memory Validator can be saved to and loaded from a file so that you can:

- share the session with a colleague
- examine the session at a later date
- compare the session with another session

- create baseline sessions for use in regression tests

Sessions can be even exported in HTML and XML formats.

You can have multiple sessions open at once, necessary for manual session comparison or automated regression testing.

Closing a session

When you've finished working with a session, it can be closed.

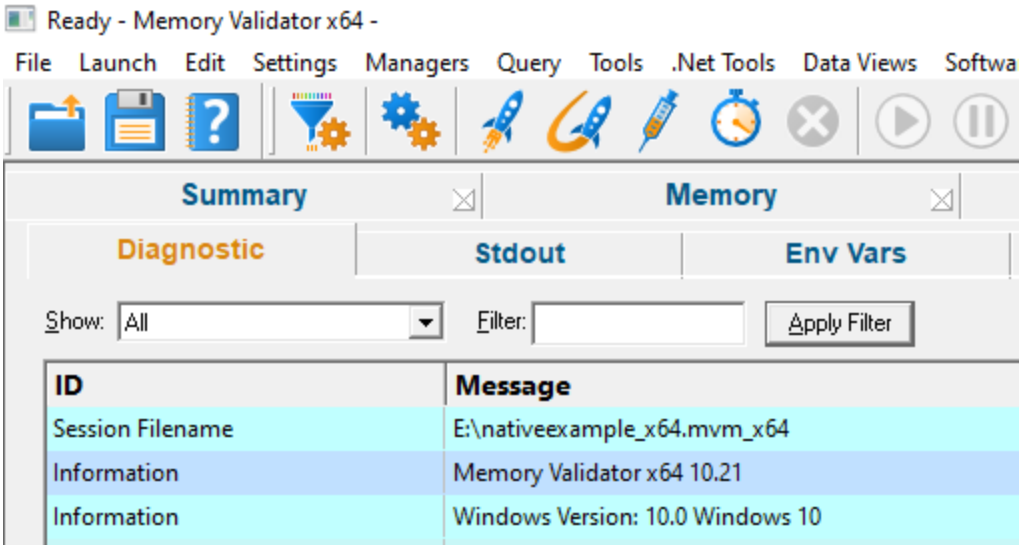
File menu › **Close Session...** › closes the session, clearing the displays

Closing a session may happen automatically if you start a new session and the session count limit is 1.

If the maximum session count allows, closed sessions still appear in the Session Manager, where they can be reopened or deleted.

Session Filename

The session filename is displayed as the first line of the diagnostic data on the Diagnostic tab.



3.17.1 Loading & Saving Sessions

Loading sessions

Load a session using any of the following options.

File menu › **Open Session...** › open a previously saved session from file (*.mvm)

or click on the Open Session icon on the standard toolbar.

or use the shortcut:

Ctrl + O Open session

If you have a limit of 1 session to be open at a time, any open session will be closed first, otherwise you can open multiple sessions at a time.

Saving sessions

Save a session using any of the following options.

File menu › **Save Session...** › saves all the session data to a file (*.mvm), prompting for a file name if necessary

File menu › **Save As...** › saves the session to a new file

or click on the Save Session icon on the standard toolbar.

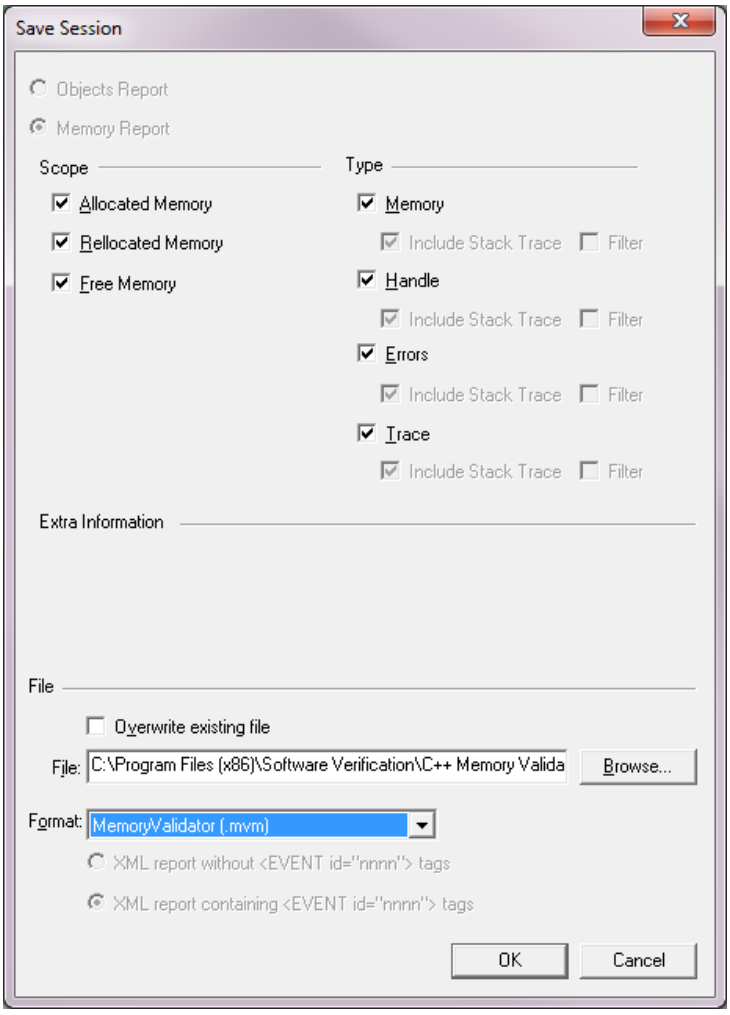
or use the shortcut:

Ctrl + S Save session

The Save Session dialog appears so you can choose the content and format of what gets saved.

The Save Session dialog

The session export dialog is very similar to the Export dialog, except here the format is set to .mvm.



Choose the scope and type of data you want to record in the saved session, and set the file name before saving.

Typically you'll accept the default settings of saving everything.

Scope section

- **Allocated Memory** › all memory and resource allocations not deallocated and information about damaged memory
- **Reallocated Memory** › all allocations that have been reallocated
- **Free Memory** › all allocations that have been deallocated

Type section

- **Memory** › memory allocations, reallocations and deallocations
- **Handle** › resource allocations and deallocations

- **Errors** › error conditions such as damaged memory, incorrect deallocation, uninitialized memory
- **Trace** › `TRACE()` and `OutputDebugString()` messages

Unlike exports where stack traces are optional, here they are saved unfiltered for all selected data types.

File section

The file format for a saved session is set as a .mvm file

- **File** › type the filename or **Browse** to a location
- **OK** › saves the session

  Check the **overwrite existing file** option if necessary.

3.17.2 Exporting Sessions

Exporting to HTML or XML

Exporting sessions allows you to use external tools to analyse or view session data for whatever reasons you might need.

You can export to HTML or XML format:

File menu › **Export Session...** › Choose **HTML Report** or **XML Report** › shows the Export Session dialog below

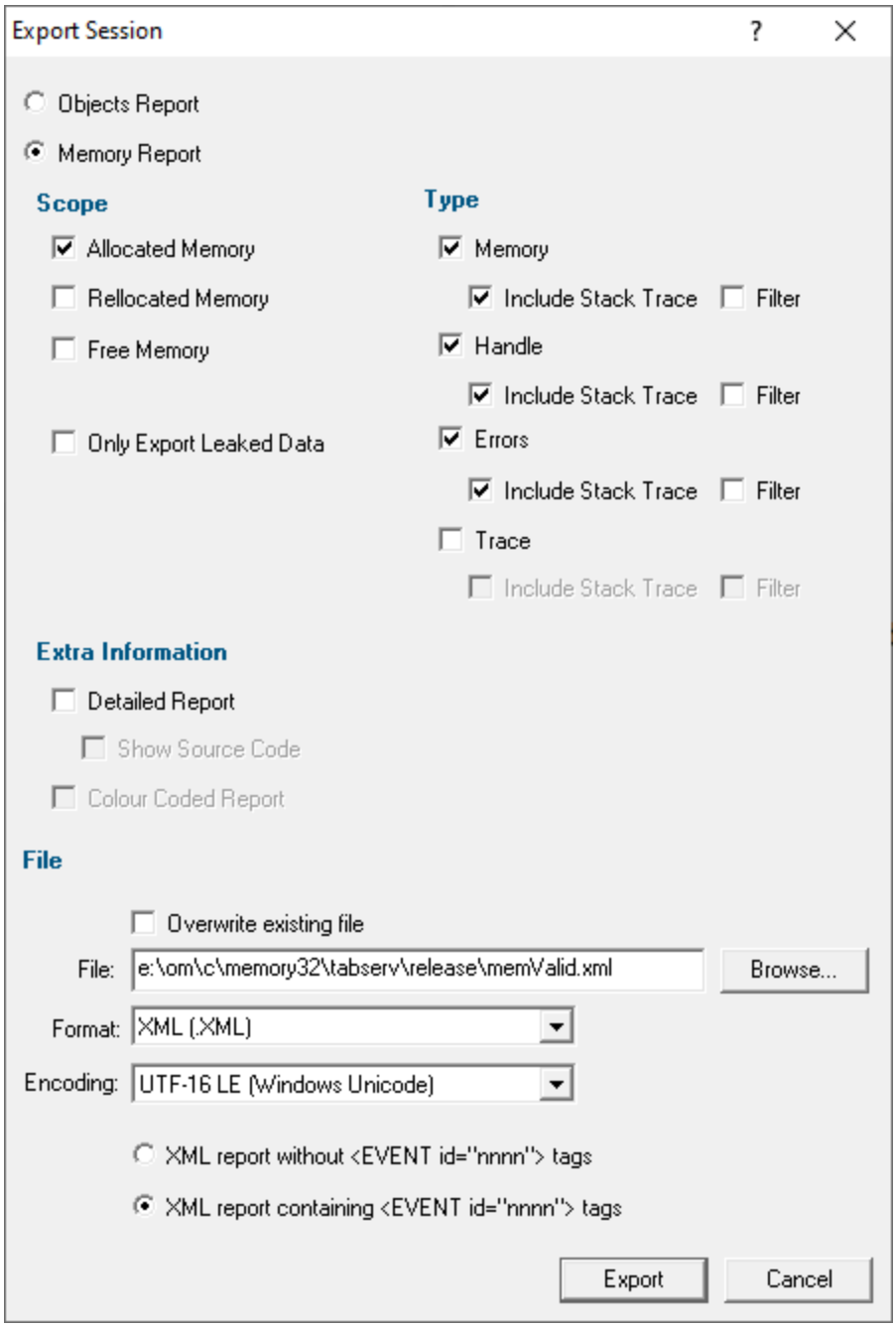
Exporting is not saving

You can't import session data.

Use save and load if you want to save session data for loading back into Memory Validator at a later date.

The Export Session dialog

The Export Session dialog looks very similar to the Save Session dialog, except there are more options enabled.



Memory or Objects

Choose to export object data or memory data:

- **Objects Report** > will export a set of data corresponding to the Objects tab view
- **Memory Report** > will export memory allocations, types, callstacks etc

An Objects Report will disable all the other options apart from the File settings.

If exporting a memory report, choose the options in the Scope, Type and Extra Information sections:

Scope section

- **Allocated Memory** › all memory and resource allocations not deallocated and information about damaged memory
- **Reallocated Memory** › all allocations that have been reallocated
- **Free Memory** › all allocations that have been deallocated
- **Only Export Leaked Data** › check to export memory *known* to have leaked

    Useful if using in-place leak detection during a session and want to export data before the session completes.

Type section

Choose what type of data you want to include

- **Memory** › memory allocations, reallocations and deallocations
- **Handle** › resource allocations and deallocations
- **Errors** › error conditions such as damaged memory, incorrect deallocation, uninitialized memory
- **Trace** › `TRACE()` and `OutputDebugString()` messages

For each type of data

- **Include Stack Trace** › includes the allocation stack trace information in the export
- **Filter** › filters the exported data according to the global and the session filters

    Leaving the export unfiltered (the default) will export all related data

Extra Information section

- **Detailed Report** › adds Thread ID and timestamp information to the report
- **Show Source Code** › if choosing a detailed report, includes the source code fragments displayed with each callstack
- **Colour Coded Report** › for HTML reports, exports a coloured HTML table layout

    The colour scheme is not configurable.

    If you want a custom style, export a detailed XML report and process that to generate the HTML report.

File section

File options are relevant whether exporting an Objects Report or a Memory Report.

- **File** › type the filename or **Browse** to a location
- **Format** › set whether exporting HTML or XML

  Defaults to the menu option selected, but included here to more easily export one format and then the other.
- **Encoding** › set whether UTF-16 LE, UTF-8 or ASCII encoding. By default the exported file is saved in the Windows Unicode format UTF-16 little endian. You can also save in UTF-8 and ASCII. ASCII has no byte order mark at the start of the file.
- **OK** › exports the session data

  Check the **overwrite existing file** option if you want to be warned about overwrites.

3.17.2.1 XML Export Tags

This section describes the XML tags used to export session data from Memory Validator.

Application and program details

An exported XML file starts with a few details about Memory Validator and the target program:

```
<XML>
    <VALIDATORINFO>Memory Validator information online</VALIDATORINFO>
    <VALIDATOR>Memory Validator name</VALIDATOR>
    <VALIDATORVERSION>Version</VALIDATORVERSION>
    <VALIDATORDATE>Build date</VALIDATORDATE>
    <VALIDATORTIME>Build time</VALIDATORTIME>
    <TITLE>Target program name</TITLE>
    <EXITCODE>Program exit status code and description</EXITCODE>
```

Session comparison data groups

The regression, improvements and leaks common to both sessions are in the following:

```
    <REGRESSIONS>...</REGRESSIONS>
    <IMPROVEMENTS>...</IMPROVEMENTS>
    <COMMONLEAKS>...</COMMONLEAKS>
```

Allocation types

Allocation events are listed in one of three containers

```
<ALLOCATED>...</ALLOCATED>
<REALLOCATED>...</REALLOCATED>
<FREE>...</FREE>
```

Allocation events

The next level of tags are shown in optional `<EVENT>` tags.

```
<EVENT>...</EVENT>
```

You can choose whether to include these event tags or omit them to produce a flat XML structure.

The options to omit the event tags are found:

- on the Session Export dialog
- on the Session Compare Export dialog
- via the -flatXMLSessionExport option when exporting sessions during regression testing
- via the -flatXMLSessionCompareExport option when exporting session comparisons during regression testing

Reports that don't include `event` tags will include all the same inner tags except `<ID>`.

Here's an example for a non-detailed report. The detail report includes thread and timestamp entries:

```
<EVENT>
 <ID>37</ID>
 <File>E:\OM\C\memory32\examples\nativeExample\nativeExample.CPP</File>
 <Line>161</Line>
 <Address>0x00372d00</Address>
 <Size>0x00000004</Size>
 <AllocType>Allocation</AllocType>
 <Leaked>TRUE</Leaked>
 <Uninitialised>FALSE</Uninitialised>
 <Damaged>FALSE</Damaged>
 <Unused>FALSE</Unused>
 <SizeError>FALSE</SizeError>
 <IncorrectUsage>FALSE</IncorrectUsage>
 <Type>Unknown </Type>
 <AllocationID>166</AllocationID>
 <STACKTRACE>
  <SYMBOL>0x00401905 nativeExample.exe CTeststakApp::CTeststakApp : [E:\OM\C\memory32\examp
  <SYMBOL>0x00401e58 nativeExample.exe $E320 : [E:\OM\C\memory32\examples\nativeExample\nat
  <SYMBOL>0x00401e33 nativeExample.exe $E323</SYMBOL>
  <SYMBOL>0x1020ad33 MSVCRTD.dll _initterm : [crt0dat.c Line 524]</SYMBOL>
  <SYMBOL>0x00412bfb nativeExample.exe wWinMainCRTStartup : [crtexe.c Line 274]</SYMBOL>
  <SYMBOL>0x7c816d4a KERNEL32.dll RegisterWaitForInputIdle</SYMBOL>
 </STACKTRACE>
</EVENT>
```

Not all of these tags will appear for a given data item in a session.

Some of them only appear when certain data items are monitored using Memory Validator.

Depending on how you use Memory Validator you may in fact never see some of these tags.

All hexadecimal numbers will have leading zeros.

- <ID> the sequence number of the event in the recorded history of all events
- <File> the source file location of the allocation event
- <Line> the source line number in the file
- <Address> the hexadecimal address of the allocated object
- <Size> the hexadecimal size of the allocated object
- <AllocType> a string indicating allocation, reallocation, etc
- <Leaked> TRUE or FALSE
- <Uninitialised> TRUE or FALSE
- <Damaged> TRUE or FALSE
- <Unused> TRUE or FALSE
- <SizeError> TRUE or FALSE
- <IncorrectUsage> TRUE or FALSE
- <Type> a string indicating the datatype of the allocated object, if known
- <AllocationID> the allocation ID of the allocation
- <Handle> the value of the allocated handle
- <HandleType> the type of handle allocated, if known
- <Heap> the handle of the heap from which the allocation was made
- <BytesOverwrite> the number of bytes overwritten in a buffer overflow/buffer underflow error
- <Process> the handle of the process which made this allocation
- <THREAD> the id of the thread in which the allocation was made.
- <TIME> the timestamp of the allocation. This is a relative 'ticker' time rather than an absolute time, and is not measured in hours/mins/secs.
- <ReportType> type of trace message
- <Message> a message from a **TRACE()** macro or **OutputDebugString()**
- <NumWords> the hexadecimal number of words that were found to be uninitialized in a data object

- <UserData> user specified data for a user defined allocation

  Only seen for data collected for custom heaps via the API functions mvUserCustomAlloc(), mvUserCustomReAlloc(), and mvUserCustomFree().

- <RefCount> user specified data for a user defined allocation

  Only seen for data collected for custom heaps via the API functions mvUserCustomRefCountDecrement(), and mvUserCustomRefCountIncrement().

Stacktrace tags

The stacktrace for the event is defined in the tags

```
<STACKTRACE>...</STACKTRACE>
```

In the stacktrace are a number of symbols

```
<SYMBOL>symbol data</SYMBOL>
```

The symbol data includes:

- hexadecimal address
- dll/exe name terminated by a semi-colon
- function name
- filename and line number in square brackets, if known

Example (from the XML fragment above):

```
<SYMBOL>0x00401905 nativeExample.exe CTeststakApp::CTeststakApp : [E:
\OM\C\memory32\examples\nativeExample\nativeExample.CPP Line 161]</SYMBOL>
```

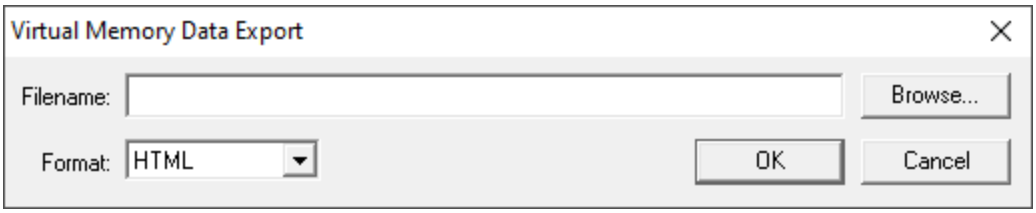
3.17.3 Exporting Virtual Memory Data

Exporting virtual memory data

On the Virtual tab of the main window, the Pages and Paragraphs tabs have an Export button at the top right.

- **Export** › enables the export of virtual memory data in HTML, XML or CSV file formats

  Displays the Virtual Memory Data Export dialog below.



The Virtual Memory Data Export dialog

Just set the file and format to export to:

- **Filename** › enter or **Browse** to set the filename to export the data to
- **Format** › HTML, XML, or CSV export formats,

  Auto selected based on the extension of the filename if you set that first

Examples of export formats

The following example fragments show the HTML, XML and CSV output for the same application.

Show exampe HTML fragment

```
<HTML>
 <HEAD>
  <META NAME="Validator Version" CONTENT="1.0">
  <META NAME="Validator Date" CONTENT="Aug 20 2005">
  <META NAME="Validator Time" CONTENT="15:09:41">
  <TITLE>nativeExample.exe:Sat Aug 20 15:10:20 2005 Virtual Memory</TITLE>
 </HEAD>
 <BODY>
  <H1>nativeExample.exe:Sat Aug 20 15:10:20 2005 Virtual Memory</H1>
   <TABLE BORDER>
    <TR><TH>Address</TH><TH>Size</TH><TH>Type</TH><TH>Description</TH></TR>
    <TR BGCOLOR="#ffffff"><TD>0x00000000</TD><TD>(64Kb) (0.06Mb)</TD><TD></TD><TD>F
    <TR BGCOLOR="#ffffff"><TD>0x00010000</TD><TD>(4Kb) (0.00Mb)</TD><TD>Private</TD

    ...

    <TR BGCOLOR="#ffc6ff"><TD>0x7c9c0000</TD><TD>(8272Kb) (8.08Mb)</TD><TD>Image</T

    ...

   </TABLE>
 </BODY>
</HTML>
```

Show exampe XML fragment

All the tags shown below are present in every report except the <TYPE> which is only present if type information is known.

```
<XML>
 <VALIDATORVERSION>1.0</VALIDATORVERSION>
 <VALIDATORDATE>Aug 20 2005</VALIDATORDATE>
 <VALIDATORTIME>15:17:19</VALIDATORTIME>
 <TITLE>nativeExample.exe:Sat Aug 20 15:17:45 2005 Virtual Memory</TITLE>
 <VIRTUAL>
  <DATA>
   <ADDRESS>0x00000000</ADDRESS>
   <SIZE>(64Kb) (0.06Mb)</SIZE>
   <DESCRIPTION>Free</DESCRIPTION>
  </DATA>
  <DATA>
   <ADDRESS>0x00010000</ADDRESS>
   <SIZE>(4Kb) (0.00Mb)</SIZE>
   <TYPE>Private</TYPE>
   <DESCRIPTION>Commited</DESCRIPTION>
  </DATA>

  ...

  <DATA>
   <ADDRESS>0x7c9c0000</ADDRESS>
   <SIZE>(8272Kb) (8.08Mb)</SIZE>
   <TYPE>Image</TYPE>
   <DESCRIPTION>DLL: c:\windows\system32\shell32.dll</DESCRIPTION>
  </DATA>

  ...

 </VIRTUAL>
</XML>
```

Show exampe CSV fragment

Each line contains four comma delimited fields:

- hexadecimal address
- size in Kb and Mb
- memory type, or a blank space if not known
- description

```
0x00000000, (64Kb) (0.06Mb), , Free,
0x00010000, (4Kb) (0.00Mb), Private, Commited,

...

0x7c9c0000, (8272Kb) (8.08Mb), Image, DLL: c:\windows\system32\shell32.dll,
0x7d1d4000, (38000Kb) (37.11Mb), , Free,
0x7f6f0000, (28Kb) (0.03Mb), Mapped, Commited,
0x7f6f7000, (996Kb) (0.97Mb), Mapped, Reserved,
0x7f7f0000, (7936Kb) (7.75Mb), , Free,
```

```
0x7ffb0000, (144Kb) (0.14Mb), Mapped, Commited,
0x7ffd4000, (20Kb) (0.02Mb), , Free,
0x7ffd9000, (4Kb) (0.00Mb), Private, Commited,
0x7ffda000, (4Kb) (0.00Mb), Private, Commited,
0x7ffdb000, (4Kb) (0.00Mb), Private, Commited,
0x7ffdc000, (4Kb) (0.00Mb), Private, Commited,
0x7ffdd000, (4Kb) (0.00Mb), Private, Commited,
0x7ffde000, (4Kb) (0.00Mb), Private, Commited,
0x7ffdf000, (4Kb) (0.00Mb), , Free,
0x7ffe0000, (4Kb) (0.00Mb), Private, Commited,
0x7ffe1000, (60Kb) (0.06Mb), Private, Reserved,
0x7fff0000, (60Kb) (0.06Mb), Private, Unknown,
```

3.18 Starting your target program

Starting options

There are seven ways to start a target program and have Memory Validator collect data from it.

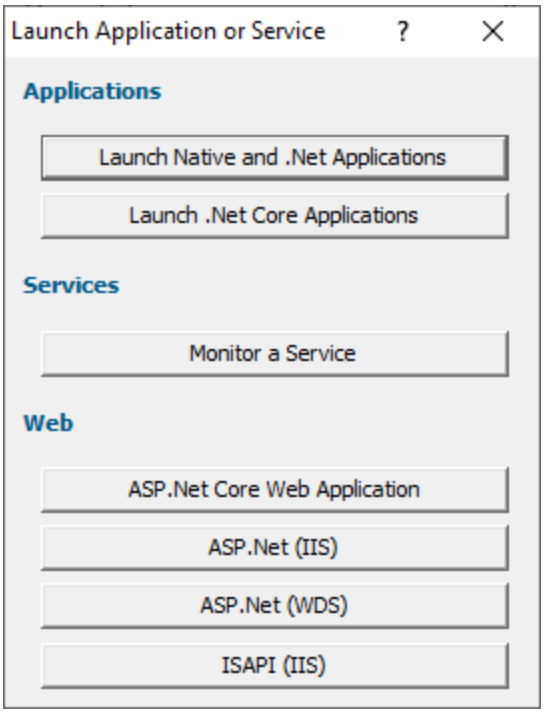
- Launch your program in a specified directory, with as many command line arguments as you want
- Inject Memory Validator into an already running program
- Wait until a specific program starts to run before attaching to it - e.g. for programs started as an OLE server
- Monitor a service
- Monitor IIS and ISAPI
- Use the Native API to start Memory Validator from code that you control
- Start Memory Validator from the command line, allowing you to automate your use of Memory Validator

If your program is linked statically to the C runtime libraries, you might want to read the topic before you start.

3.18.1 Launch chooser

The launch chooser is displayed when you click on the rocket icon on the toolbar.

There are multiple application types and services that you may wish to use. The launch chooser provides the mechanism for making that choice.



Each button will display the launch dialog associated with the instruction displayed on the button.

Applications

- Launch Native and .Net Applications
- Launch .Net Core Applications

Services

- Monitor a Service

Web

- ASP.Net Core Web Application
- ASP.Net with IIS
- ASP.Net with Web Development Server
- ISAPI with IIS

You can repeat the choice made using the launch chooser by using Ctrl + F4

3.18.2 Launching the program (native and .Net)

Launching the application

Having Memory Validator launch your program is the most common way to start up

When you're ready to start running a native or .Net program

Launch menu › **Applications** › **Launch Application** › Shows the launch program wizard or dialog below

Or use the shortcut

F4 Launch application

➜ You can easily re-launch the most recently run program.

User interface mode

There are two interface modes used while starting a program

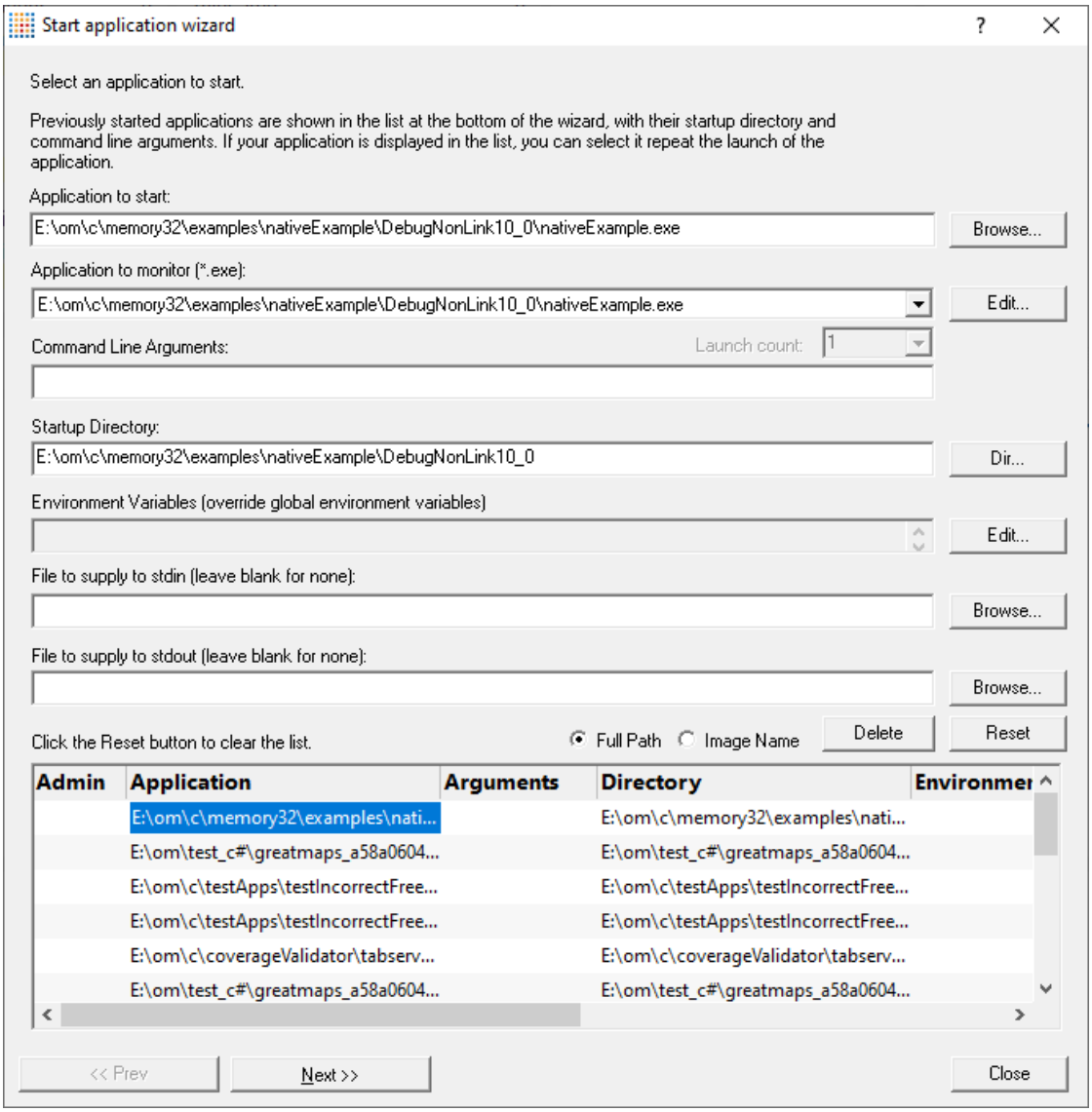
- Wizard mode guides you through the tasks in a linear fashion
- Dialog mode has all options contained in a single dialog

All the options are the same - just in different places

In this section we'll cover the Wizard mode first and the Dialog mode later.

The start application wizard

On first use, the wizard appears with fields cleared, but here's an example with fields set:



Enter the details for your program, or if you want to run a previous program select it from the application list to repopulate the details.

After entering details click **Next >>** for the next page of the wizard.

Administrator privileges when launching your program

The following applies only if you did *not* start Memory Validator in administrator mode.

Anywhere you see the icon indicates that administrator privileges will be required to proceed.

If you started Memory Validator in administrator mode, you won't see any of these warnings, and everything will behave as normal.

Page 1: Entering details

- **Application to start** › type or **Browse** to set the program name to launch

  You can also choose a batch file and the first executable started in the batch file will be launched.

  You can also choose a powershell script and the first executable started in the powershell script will be launched.

  Manually typing a path will show red text until a valid path is entered, after which the text becomes black.

- **Application to monitor** › choose the application that actually gets monitored

  This will typically just be the program that you set to start - unless otherwise specified.

  Alternatively you can monitor another application which will get launched by the start program.

  If the start application has already been added to the Applications to Monitor settings with a default application then that default will be entered here automatically.

  Otherwise, if nothing has been set up yet, you can do it from here:

  - **Edit...** › set the child applications that can be monitored for the start program

    This uses the Applications to Monitor dialog - which is exactly equivalent to using the Applications to Monitor settings page.

  A fallback option is to start monitoring `<<Any application that is launched>>`.

  If in doubt, just use the same as the start application.

  ➜ See also: Application to Monitor settings

- **Launch Count** › when monitoring a *child* application, set its n^{th} invocation as the one to monitor

  If the application to start is the same as the application to monitor then this is set to 1 and cannot be changed.

  This will be reset to 1 every time the Application to Monitor field selection changes.

  If in doubt, leave it set to 1.

  ➜ See also: Launch Count.

- **Command Line Arguments** › enter program arguments exactly as passed to the target program

- **Startup Directory** › enter or click **Dir...** to set the directory for the program to start in

  When setting your target program, this will default to the location of the executable

- **Environment Variables** › click **Edit...** to set any additional environment variables before your program starts

  These are managed in the Environment Variables Dialog.

- **File to supply to stdin** › optionally enter or **Browse** to set a file to be read and piped to the standard input of the application

- **File to supply to stdout** › optionally enter or **Browse** to set a file to be written with data piped from the standard output of the application

Page 1: Using details from a previous run

The list at the bottom of the wizard shows previously run programs.

Selecting an item in the list populates all the details above as used on the last run for that program.

You can still edit those details before starting.

- **Full path** › shows the full path to the executable in the list
- **Image Name** › shows the short program name without path
- **Delete** › removes a selected program from the list
- **Reset** › clears all details in the wizard - including the list of previously run applications below

Page 2: Data collection and redirection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?

  - **Native Only** › Ignore all .Net data in the target application.
  - **.Net Only** › Ignore all Native data in the target application.
  - **Mixed Mode** › Collect both Native and .Net data from the target application

  This setting cannot be changed after the application is launched

- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  ➡ See the section on controlling data collection for how to turn collection on and off after launch.

- **Redirect standard output** › Controls redirection of stdout and stderr

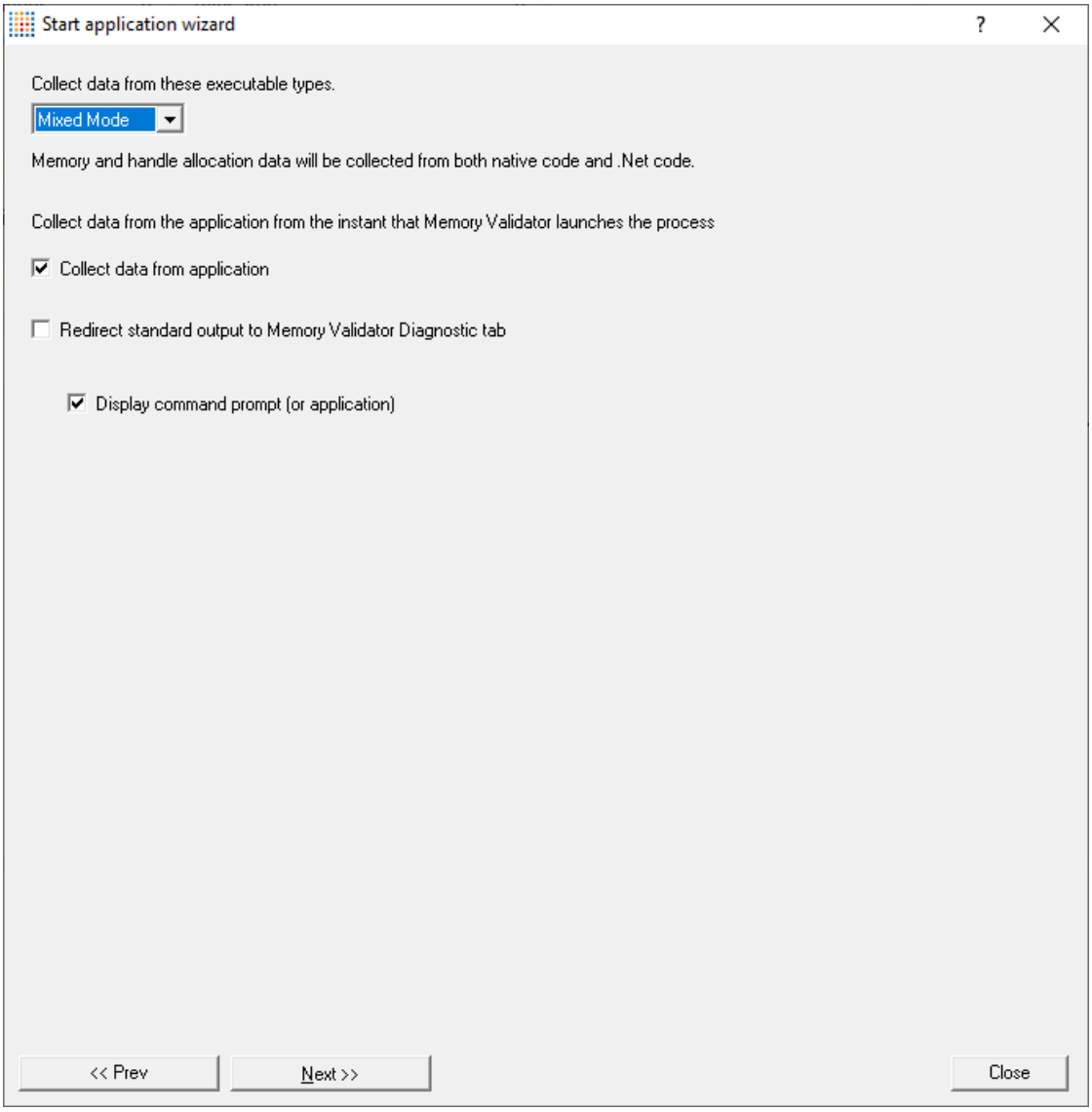
  Use this option if you want to collect the output of stdout and stderr for later analysis.

  Be aware that if the output of the program under test generates a lot of data via stdout or stderr that this data will need to be stored in memory and could exhaust Memory Validator's memory.

- **Display command prompt** › Shows or hides the launched application.

  If you are collecting stdout and stderr you may not be interested in viewing the application (or the command prompt if it is a console application). This provides you the option to hide the application when it is running.

  Be aware that if you hide a command prompt you will not be able to type anything into the application.



Page 3: Summary and starting your program

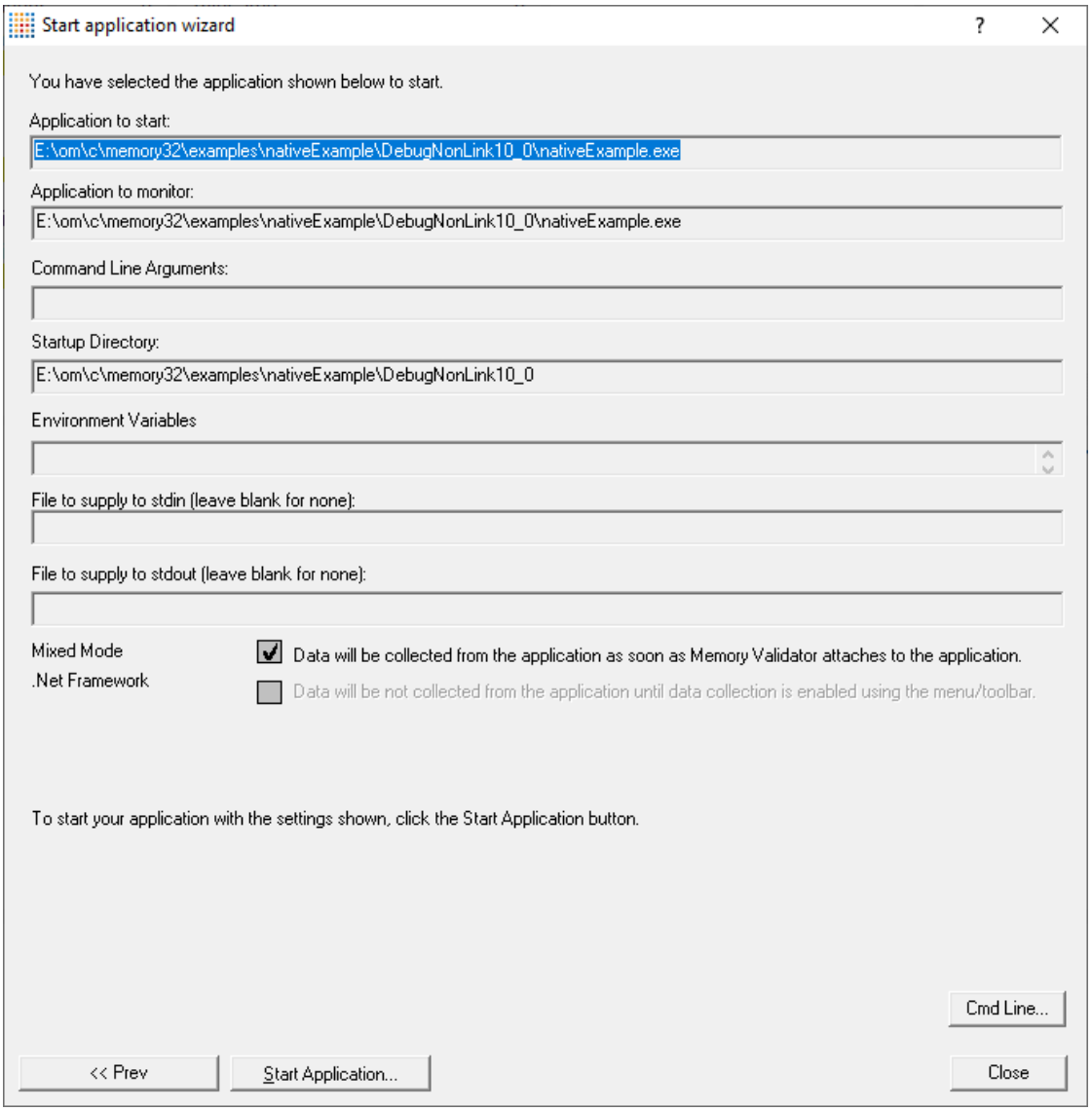
The last page is just a summary of the options you have chosen.

Something missing? The choice of launch method is no longer necessary and has been removed.

If you're happy with the settings, go ahead:

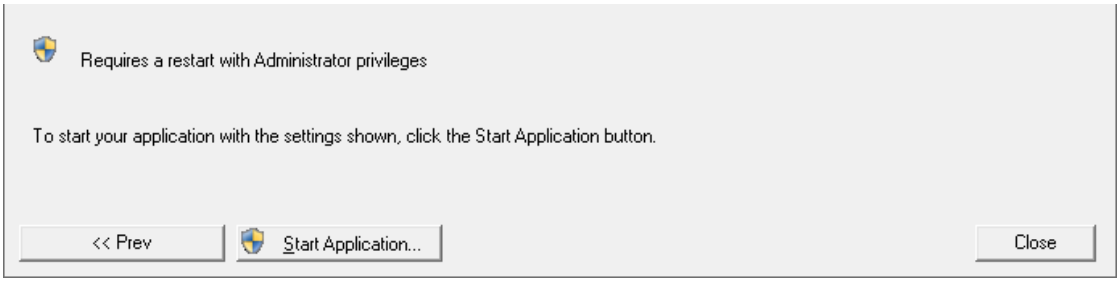
- **Start Application...** › start your program and attach Memory Validator to it
- **Cmd Line...** › display the command line builder

If your program is linked statically to the C runtime libraries, you might want to read the topic before you start.

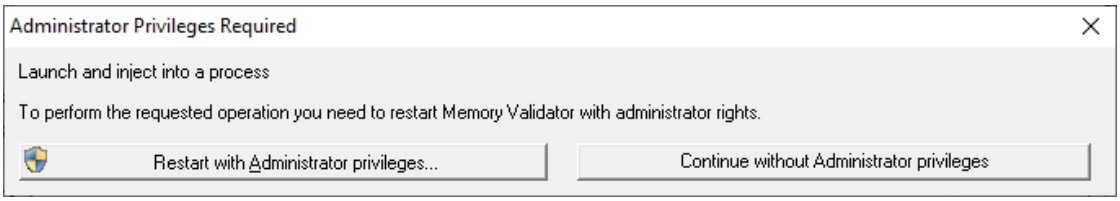


Administrator privileges in wizard mode

If administrator privileges are required you'll be reminded of the need to restart here:



- **Start Application...** › shows the Administrator Privileges Required confirmation dialog before restarting.



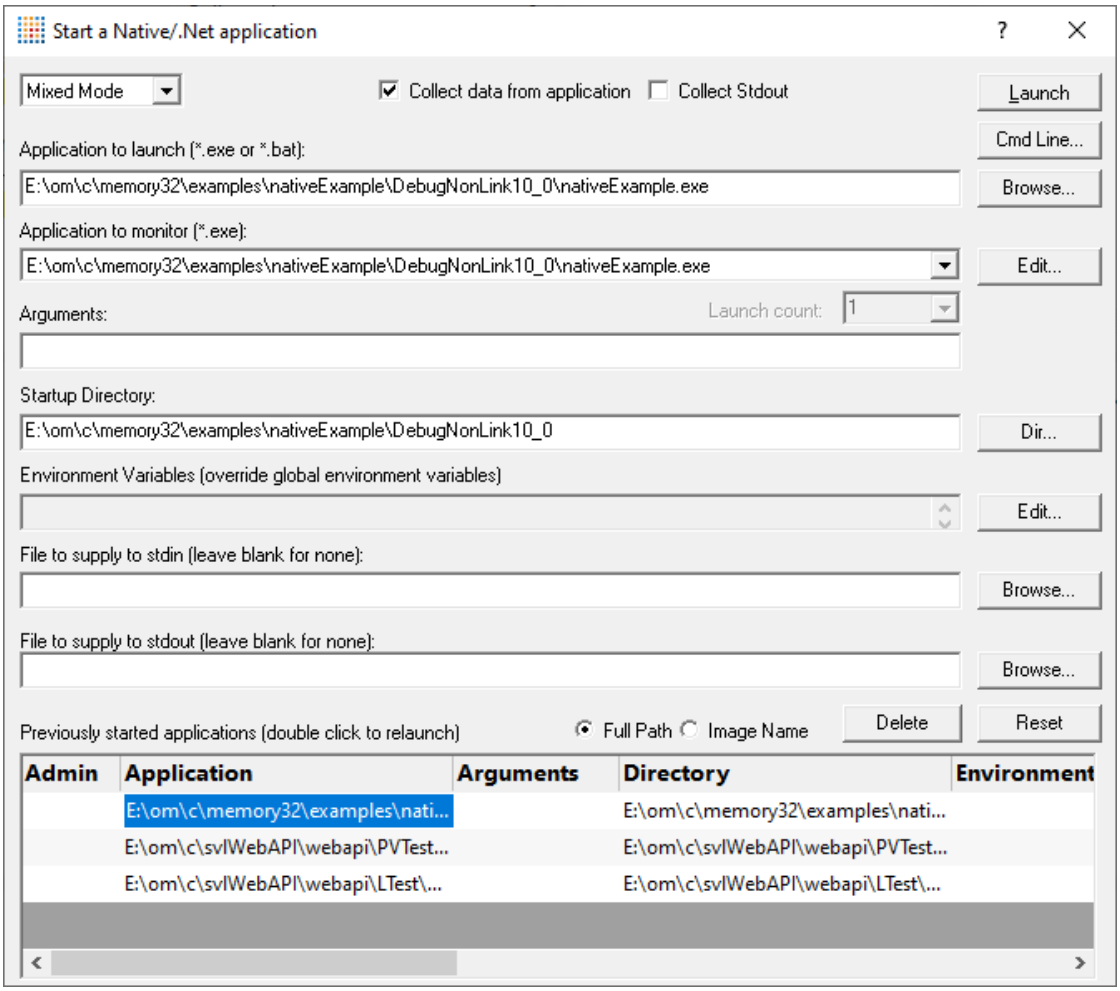
Dialog mode

In Dialog mode, all the settings are in one dialog which looks very much like the first page of the launch wizard above.

The option to start collecting data is at the top.

- **Launch** › start your program and attach Memory Validator to it
- **Cmd Line...** › display the command line builder

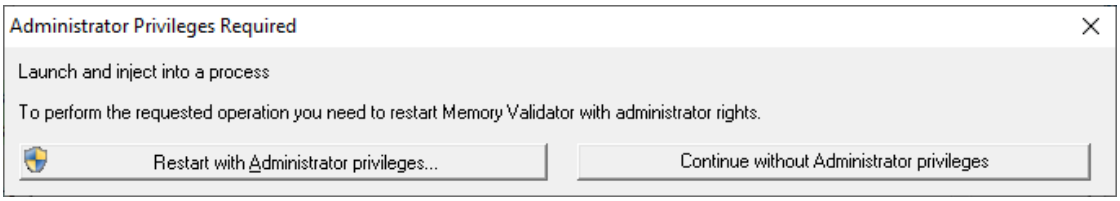
  Double clicking a program in the list will also start it immediately.



Administrator privileges in dialog mode

If administrator privileges are required, the Launch button will show the privileges icon reminding you of the need to restart.

- **Launch** > shows the Administrator Privileges Required confirmation dialog before restarting



If you started Memory Validator in administrator mode, you won't see any of these warnings, and everything will behave as normal.

How do I use Application to Monitor and Launch Count?

The three fields **Application to Start**, **Application to Monitor** and **Launch Count** work together to control which application actually gets monitored by Memory Validator.

Let's say we have a program **P**.

In the simplest case, simply:

- *start* **P**
- *monitor* **P**
- the Launch Count defaults to **1** and cannot be changed.

If **P** launches an application and you just want to monitor whatever that is:

- *start* **P**
- *monitor* `<<Any application that is launched>>`
- leave the Launch Count at **1**

If **P** launches an application **A** and maybe others as well, and you specifically want to monitor only **A** as it's launched:

- use the Application to Monitor settings to add a definition for P and child applications **A**
- *start* **P**
- *monitor* **A**
- leave the Launch Count at **1**

If **P** launches an application **A** many times and you specifically want to monitor the third invocation:

- use the Application to Monitor settings to add a definition for P and child applications **A**
- *start* **P**
- *monitor* **A**
- set the Launch Count to **3**

3.18.3 Launching the program (.Net Core)

Launching the application

Having Memory Validator launch your program is the most common way to start up

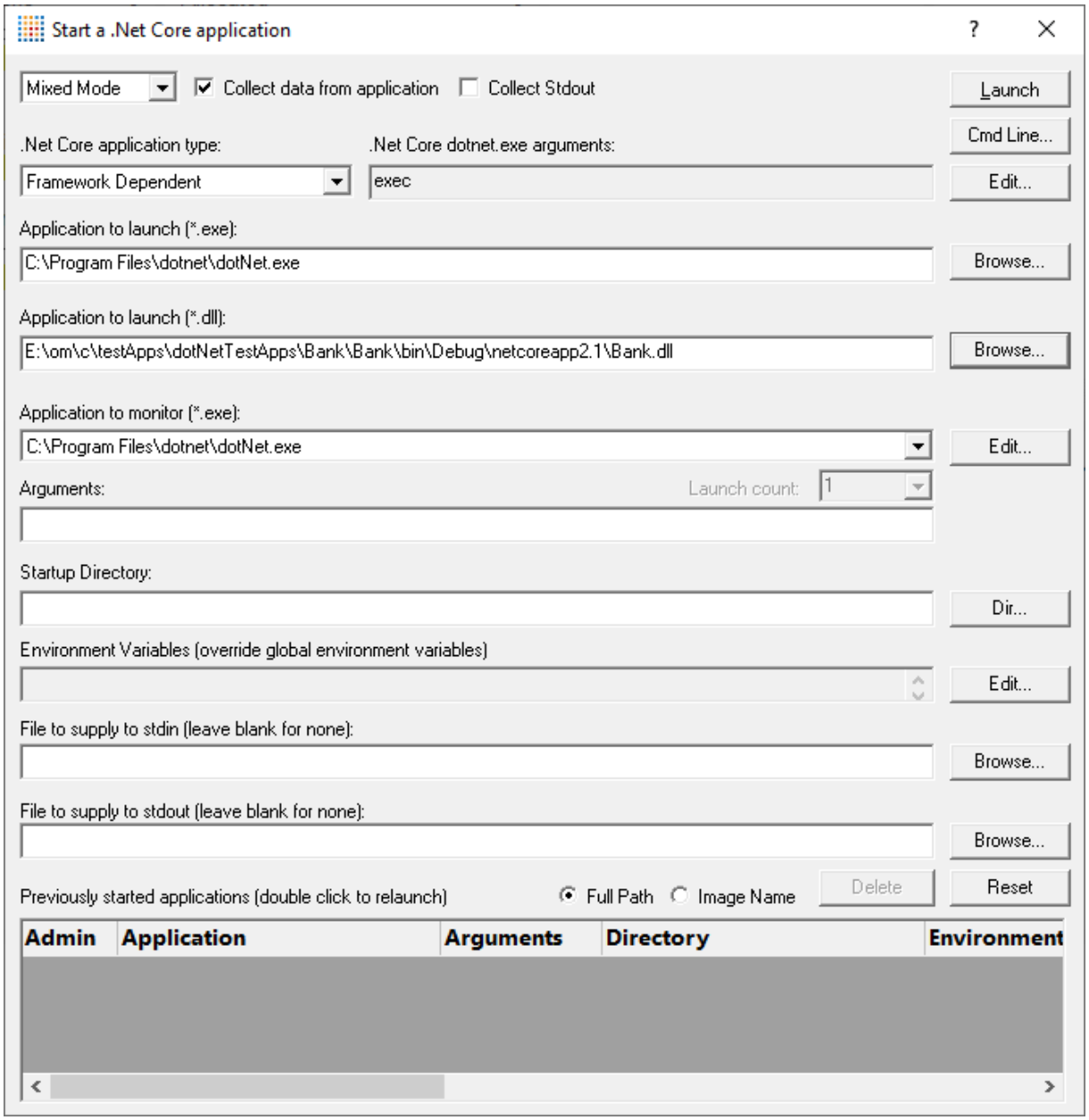
When you're ready to start running a .Net Core program

Launch menu › **Applications** › **Launch .Net Core Application** › Shows the .Net Core launch dialog below

Or use the shortcut

Shift + F4 Launch .Net Core Application

➜ You can easily re-launch the most recently run program.



.Net Core Application Type

.Net Core applications can be self contained or framework dependent. This changes how the launch dialog works.

- **.Net Core application type** › choose which type of .Net Core application you are launching

.Net Core Self Contained Application

- **Application to launch (*.exe)** › type or **Browse** to set the program name (*.exe) to launch

  When you set this value the Application to launch DLL field will be automatically populated to have the same name as the EXE field but with a .DLL extension.

- **Application to launch (*.dll)** › type or **Browse** to set the program name (*.dll) to launch

- **Application to monitor** › choose the application that actually gets monitored

  This will typically just be the program that you set to start - unless otherwise specified.

  Alternatively you can monitor another application which will get launched by the start program.

  If the start application has already been added to the Applications to Monitor settings with a default application then that default will be entered here automatically.

  Otherwise, if nothing has been set up yet, you can do it from here:

  - **Edit...** › set the child applications that can be monitored for the start program

    This uses the Applications to Monitor dialog - which is exactly equivalent to using the Applications to Monitor settings page.

  A fallback option is to start monitoring `<<Any application that is launched>>`.

  If in doubt, just use the same as the start application.

  ➡ See also: Application to Monitor settings

.Net Core Framework Dependent Application

- **Application to launch (*.exe)** › type or **Browse** to set the program name (*.exe) to launch

  We don't auto-populate this field when you choose the Framework dependent application type. This because you may have your .Net Core runtime stored in a location that we can't auto-detect.

  To accommodate alternate locations for the .Net Core runtime we only auto-populate this field if it is empty when you choose the application DLL.

- **Application to launch (*.dll)** › type or **Browse** to set the program name (*.dll) to launch

  If you set this when Application to launch EXE field is empty, the EXE field will be automatically populated with the path to the system .Net Core framework dependent runtime.

This is typically **c:\program files\dotnet\dotnet.exe**.

- **Application to monitor** › choose the application that actually gets monitored

  This will typically just be the program that you set to start - unless otherwise specified.

  Alternatively you can monitor another application which will get launched by the start program.

  If the start application has already been added to the Applications to Monitor settings with a default application then that default will be entered here automatically.

  Otherwise, if nothing has been set up yet, you can do it from here:

  - **Edit...** › set the child applications that can be monitored for the start program

    This uses the Applications to Monitor dialog - which is exactly equivalent to using the Applications to Monitor settings page.

  A fallback option is to start monitoring `<<Any application that is launched>>`.

  If in doubt, just use the same as the start application.

  ➜ See also: Application to Monitor settings

- **.Net Core dotnet.exe arguments** › any arguments that will be passed to the .Net Core runtime to control how the .Net Core runtime behaves.

- **Edit...** › displays the .Net Core runtime arguments editor

Fields common to all .Net Core applications

- **Launch Count** › when monitoring a *child* application, set its n^{th} invocation as the one to monitor

  If the application to start is the same as the application to monitor then this is set to 1 and cannot be changed.

  This will be reset to 1 every time the Application to Monitor field selection changes.

  If in doubt, leave it set to 1.

  ➜ See also: Launch Count.

- **Command Line Arguments** › enter program arguments exactly as passed to the target program

- **Startup Directory** › enter or click **Dir...** to set the directory for the program to start in

  When setting your target program, this will default to the location of the executable

- **Environment Variables** › click **Edit...** to set any additional environment variables before your program starts

  These are managed in the Environment Variables Dialog.

- **File to supply to stdin** › optionally enter or **Browse** to set a file to be read and piped to the standard input of the application
- **File to supply to stdout** › optionally enter or **Browse** to set a file to be written with data piped from the standard output of the application

Using details from a previous run

The list at the bottom of the wizard shows previously run programs.

Selecting an item in the list populates all the details above as used on the last run for that program.

You can still edit those details before starting.

- **Full path** › shows the full path to the executable in the list
- **Image Name** › shows the short program name without path
- **Delete** › removes a selected program from the list
- **Reset** › clears all details in the wizard - including the list of previously run applications below

Data collection and redirection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?
  - **Native Only** › Ignore all .Net data in the target application.
  - **.Net Only** › Ignore all Native data in the target application.
  - **Mixed Mode** › Collect both Native and .Net data from the target application

  This setting cannot be changed after the application is launched
- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  ➜ See the section on controlling data collection for how to turn collection on and off after launch.
- **Redirect standard output** › Controls redirection of stdout and stderr

  Use this option if you want to collect the output of stdout and stderr for later analysis.

Be aware that if the output of the program under test generates a lot of data via stdout or stderr that this data will need to be stored in memory and could exhaust Memory Validator's memory.

- **Display command prompt** › Shows or hides the launched application.

  If you are collecting stdout and stderr you may not be interested in viewing the application (or the command prompt if it is a console application). This provides you the option to hide the application when it is running.

  Be aware that if you hide a command prompt you will not be able to type anything into the application.

The option to start collecting data is at the top.

- **Launch** › start your program and attach Memory Validator to it

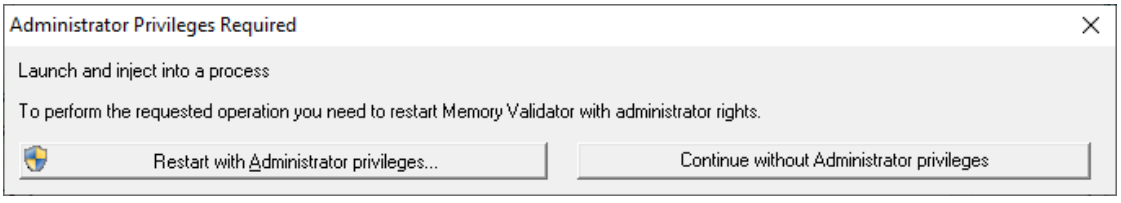
  Double clicking a program in the list will also start it immediately.

- **Cmd Line...** › display the command line builder

Administrator privileges in dialog mode

If administrator privileges are required, the Launch button will show the privileges icon reminding you of the need to restart.

- **Launch** › shows the Administrator Privileges Required confirmation dialog before restarting



If you started Memory Validator in administrator mode, you won't see any of these warnings, and everything will behave as normal.

How do I use Application to Monitor and Launch Count?

The three fields **Application to Start**, **Application to Monitor** and **Launch Count** work together to control which application actually gets monitored by Memory Validator.

Let's say we have a program **P**.

In the simplest case, simply:

- *start* **P**
- *monitor* **P**
- the Launch Count defaults to **1** and cannot be changed.

If **P** launches an application and you just want to monitor whatever that is:

- *start* **P**
- *monitor* `<<Any application that is launched>>`
- leave the Launch Count at **1**

If **P** launches an application **A** and maybe others as well, and you specifically want to monitor only **A** as it's launched:

- use the Application to Monitor settings to add a definition for P and child applications **A**
- *start* **P**
- *monitor* **A**
- leave the Launch Count at **1**

If **P** launches an application **A** many times and you specifically want to monitor the third invocation:

- use the Application to Monitor settings to add a definition for P and child applications **A**
- *start* **P**
- *monitor* **A**
- set the Launch Count to **3**

3.18.4 Re-launching the program

Re-launching the application

It's very easy to start another session using the most recently run program and settings:

Launch menu › **Applications** › **Re-Start Application...** › starts the most recently launched program

or click on the re-launch icon on the session toolbar.

or use the shortcut

F5 Re-start application

If the previously launched program was Native, .Net or .Net Core the application will be restarted immediately. No wizards or dialogs appear.

If the previously launched program was a service the appropriate monitor service dialog will be displayed.

➜ In the general questions see *Why might Inject or Launch fail?* for troubleshooting launch problems.

There is no difference between wizard and dialog interface mode when re-launching.

3.18.5 Injecting into a running program

Injecting into a running program

Memory Validator attaches to a running process by injecting the stub into the process so it can start collecting data.

Choose one of these methods of starting the injection:

Launch menu › **Applications** › **Inject...** › shows the Attach to Running Process wizard or dialog below

or click on the Inject icon on the session toolbar.

or use the shortcut

F3 Inject into running application

Injecting into a service?

If your process is a service, Memory Validator won't be able to attach to it.

Services can't have process handles opened by third party applications, even with Administrator privileges.

In order to work with services, you can use the NT service API and monitor the service

User interface mode

There are two interface modes used while starting a program

- Wizard mode guides you through the tasks in a linear fashion

- Dialog mode has all options contained in a single dialog

All the options are the same - just in slightly different places

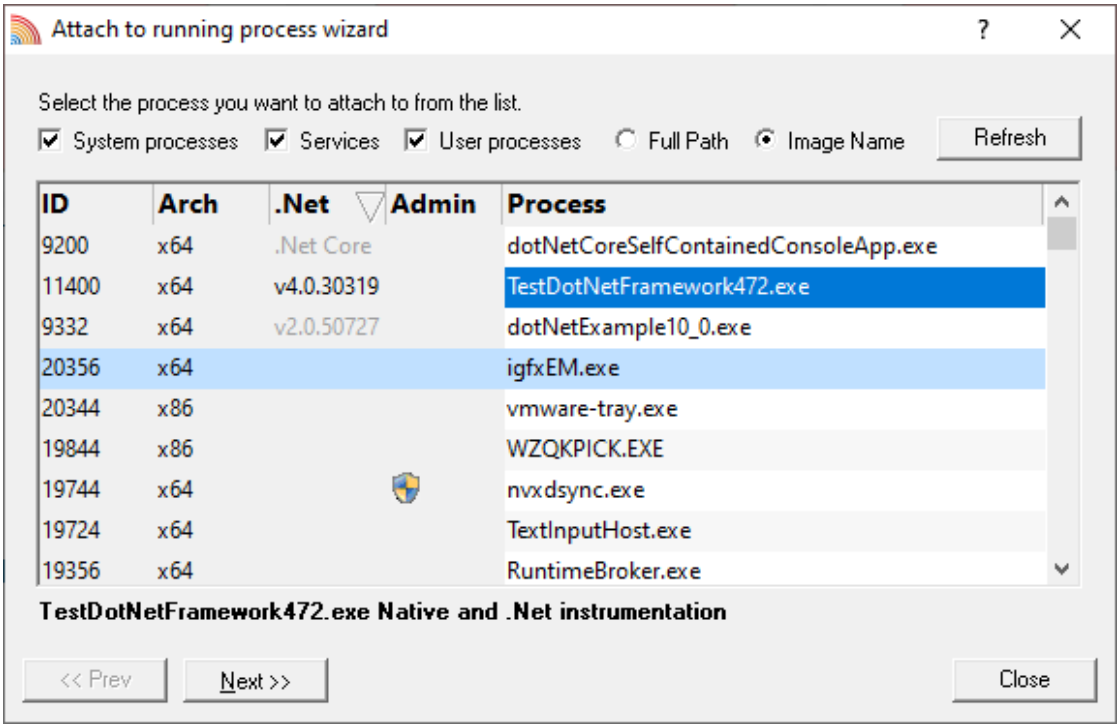
In this section we'll cover the Wizard mode first and the Dialog mode later.

The attach to running process wizard

The first page of the wizard shows a list of running system and user processes.

The **Arch** column is not shown when running 32 bit Memory Validator because only 32 bit processes are listed.

Any processes that have grayed out **.Net** values cannot instrument the .Net part of the application (native components will be instrumented).



Choose the process and click **Next >>** for the next page of the wizard.

Page 1: Choosing the process

- **System processes / Services / User processes** › show either of system or services or user processes in the list, or both
- **Full path** › shows the full path to the process executable in the list
- **Image Name** › shows the short program name without path

- **Refresh** › update the list with currently running processes

Clicking on the headers of the list will sort them by ID or by name using the full name or short name, depending on what's displayed.

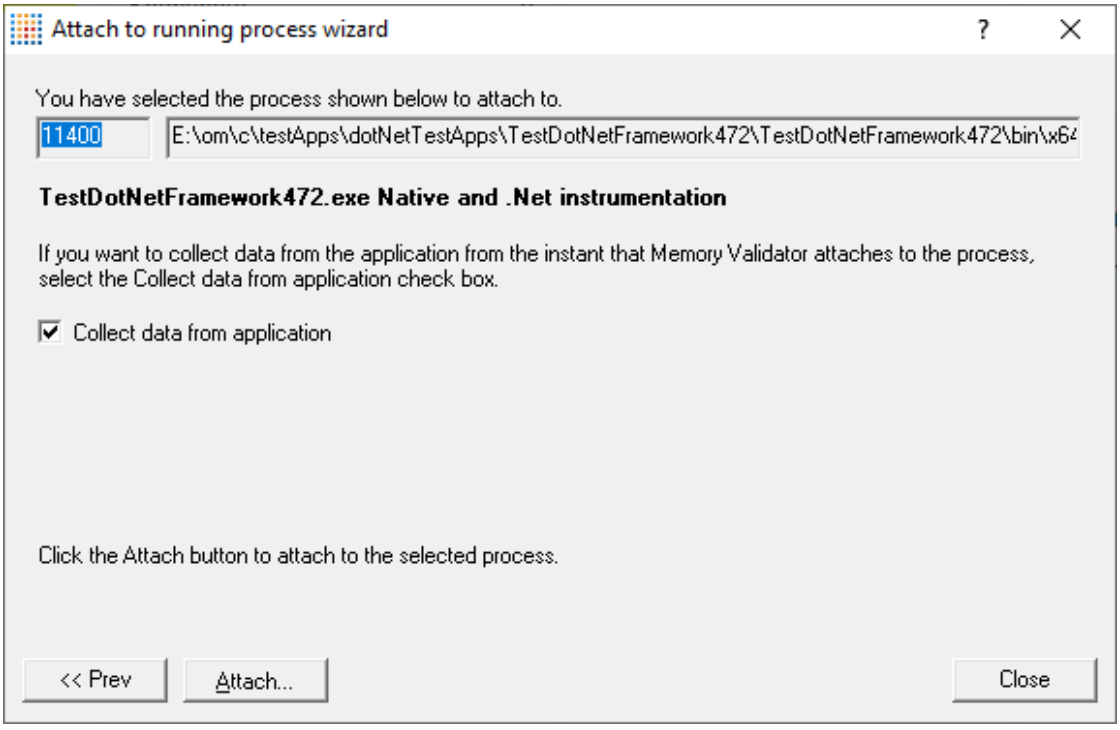
Page 2: Data collection

Depending on your application, and what you want to validate, you may want to start collecting data as soon as injection happens, or do it later.

If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

If it's the startup procedure you want to validate, obviously start collecting data from launch.

➜ See the section on controlling data collection for how to turn collection on and off after launch.



Currently we only support attaching to native applications and the native part of mixed mode applications.

Summary and starting your program

The second page confirms the process you have selected to inject into, and prompts you to attach:

- **Attach...** › injects Memory Validator into the specified process, showing progress status

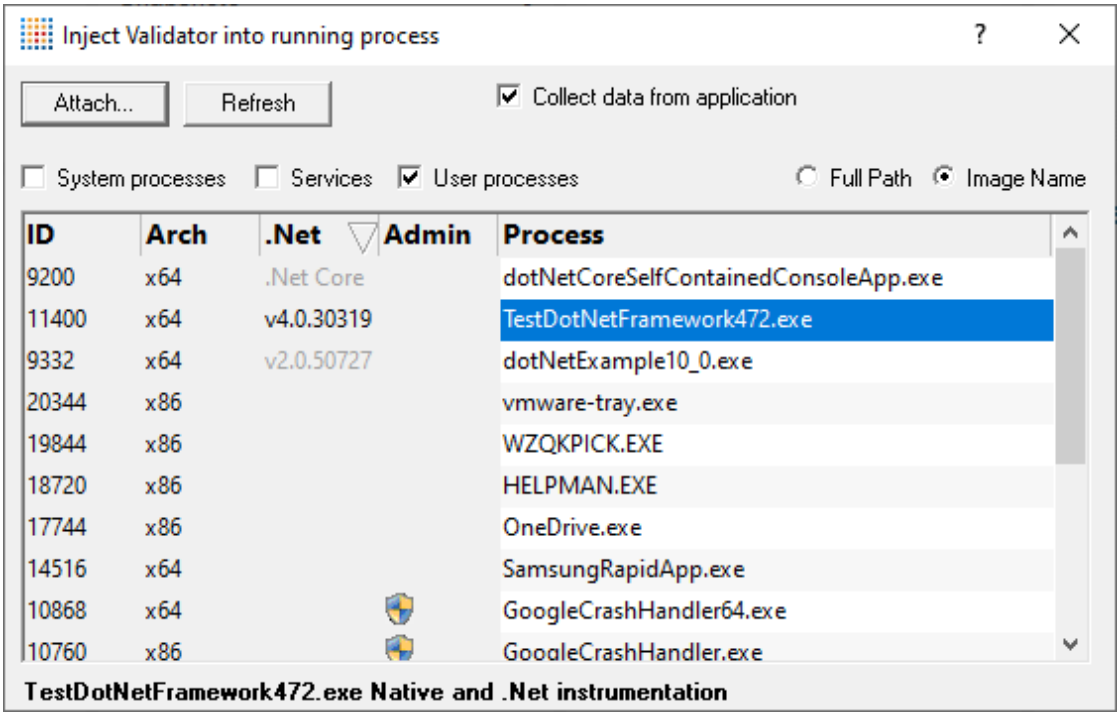
If your program is linked statically to the C runtime libraries, you might want to read the topic before you start.

➜ In the general questions see *Why might Inject or Launch fail?* for troubleshooting launch problems.

Dialog mode

In Dialog mode, all the settings are in one dialog which looks very much like the first page of the wizard above.

The option to start collecting data is at the top, as is the **Attach...** button



3.18.6 Waiting for a program

Waiting for a program

Waiting for a program is essentially the same as injection except that instead of injecting into a running program, Memory Validator watches for the process starting up and *then* injects.

If the process is a service, Memory Validator won't be able to attach to it as services can't have process handles opened by third party applications, even with Administrator privileges.

Choose one of these methods of waiting:

Launch menu > **Applications** > **Wait for Application...** > shows the Wait for application wizard or dialog below

or click on the Wait (timer) icon on the session toolbar.

or use the shortcut

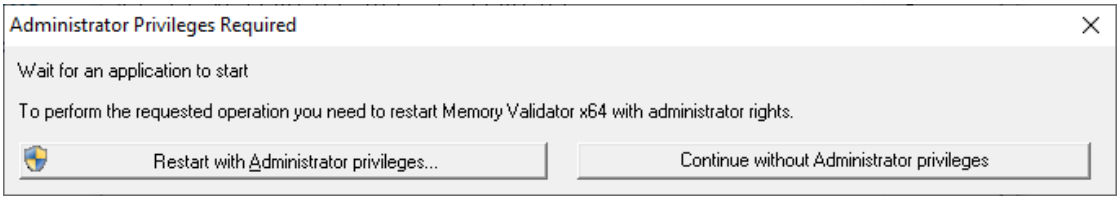
F2 Wait for application

Administrator privileges

The following applies only if you did *not* start Memory Validator in administrator mode.

If the application you want to wait for is running with Administrator privileges, Memory Validator will also need to run with Administrator privileges.

When choosing the 'wait for program' method described in this topic, a restart of Memory Validator with administrator privileges will be required to proceed.



Waiting for a service?

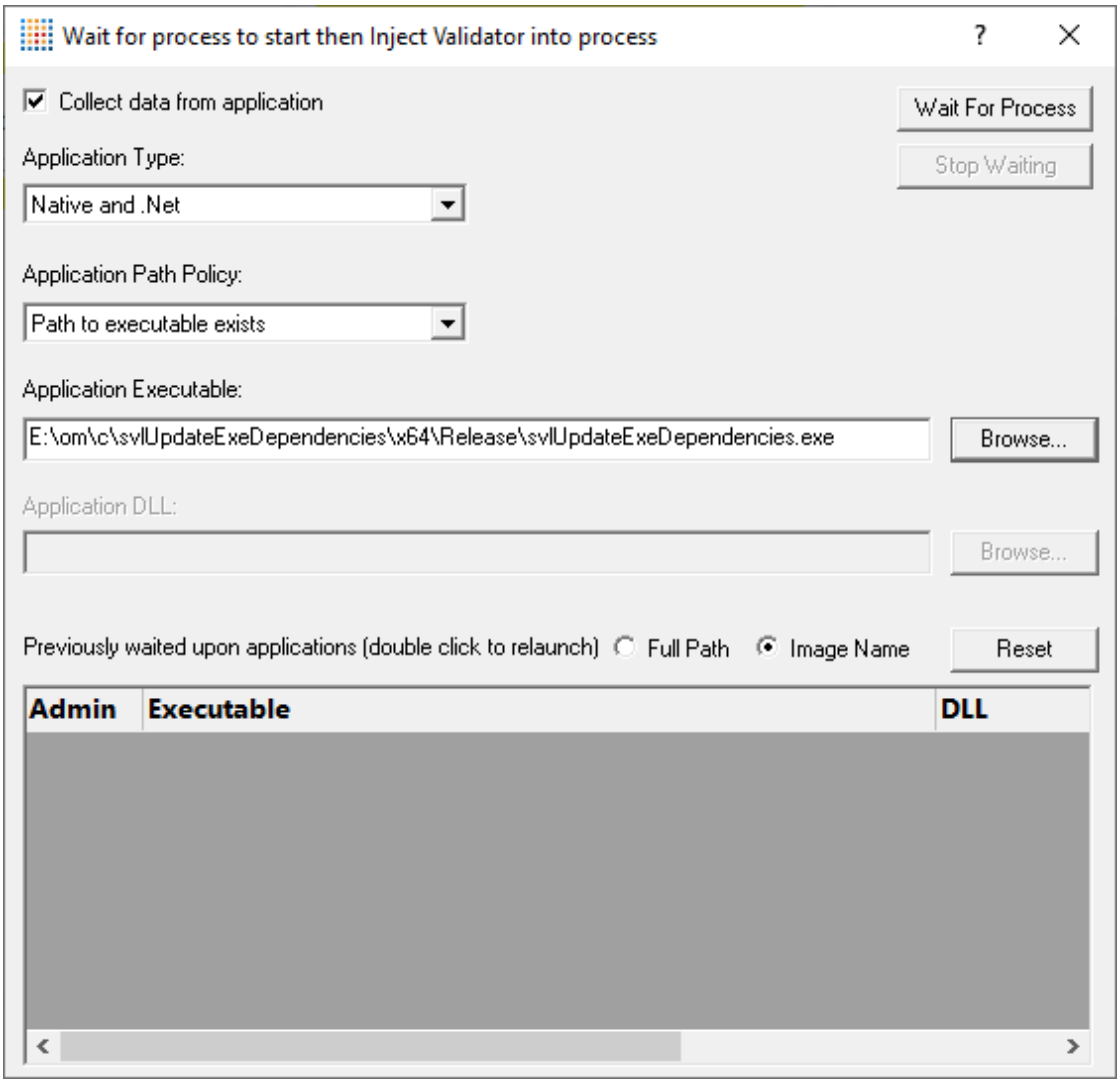
If your process is a service, Memory Validator won't be able to attach to it.

Services can't have process handles opened by third party applications, even with Administrator privileges.

In order to work with services, you can use the NT service API and monitor the service

The wait for application dialog

The wait for application dialog lets you specify the application or choose one that you've waited for previously.



- **Collect data from application** › do want to collect data from the instant you attach to the application?

  Depending on your application, and what you want to validate, you may want to start collecting data as soon as injection has happened, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  If it's the startup procedure you want to validate, obviously start collecting data from launch.

  ➡ See the section on controlling data collection for how to turn collection on and off after launch.

- **Application Path Policy** › specify how the specified executable is treated

  - Path to executable exists › the executable will be checked that it exists and is appropriate for Memory Validator to work with

- Path to executable is created dynamically › most pre-wait checks are not performed - use this if the path the executable is on does not exist at the time you start waiting for the process to start

- **Application type** › choose the type of application
  - Native and .Net
  - .Net Core (Framework Dependent)
  - .Net Core (Self Contained)

- **Application Executable** › edit or **Browse...** the application to wait to start.

  The name of the executable. For example **c:\unitTests\test.exe** or **test.exe**.

  If Application Path Policy is **Path to executable exists** this must be the full path to the executable. For example **c:\unitTests\test.exe**.

  For native applications this is the application executable.

  For .Net Framework applications this is the application executable.

  For .Net Core Framework-dependent applications this is most likely going to be **c:\program files\dotnet\dotnet.exe**.

  For .Net Core Self-contained applications this is the application executable.

- **Application DLL** › edit or **Browse...** the application DLL to wait to start. This field is only needed for .Net Core applications.

  The name of the DLL. For example **c:\unitTests\test.dll** or **test.dll**.

  If Application Path Policy is **Path to executable exists** this must be the full path to the dll. For example **c:\unitTests\test.dll**.

  For native applications this is not used.

  For .Net Framework applications this is not used.

  For .Net Core Framework-dependent applications this is the application dll. (the name of the dll that you would pass to dotnet.exe on the command line).

  For .Net Core Self-contained applications this is the dll that has the same name as the application executable. (for theApp.exe, the dll name is theApp.dll).

- **Full path** › shows the full path to the process executable in the list

- **Image Name** › shows the short program name without path

- **Reset** › clears the list

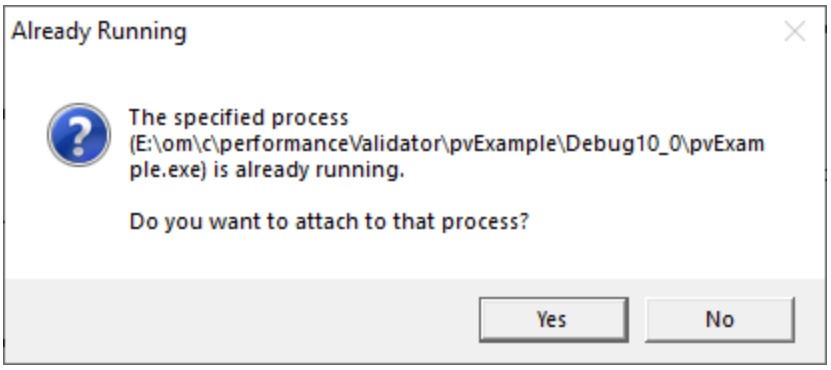
- **Wait for Process...** › starts waiting and then injects Memory Validator into the specified process, showing progress status

- **Stop Waiting** › stops the wait

If your program is linked statically to the C runtime libraries, you might want to read the topic before you start.

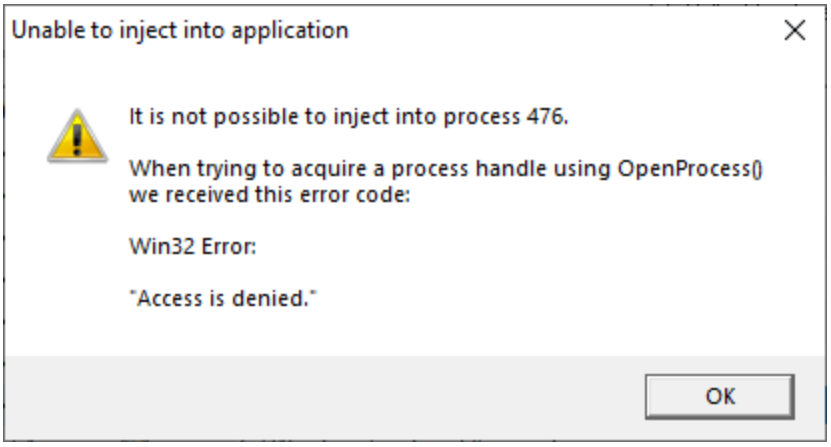
What could go wrong?

The program you're waiting for might already be running, in which case you'll be given the option to cancel or attach to the existing process:



Timing issues are inherit with native injecting into a program as it starts up.

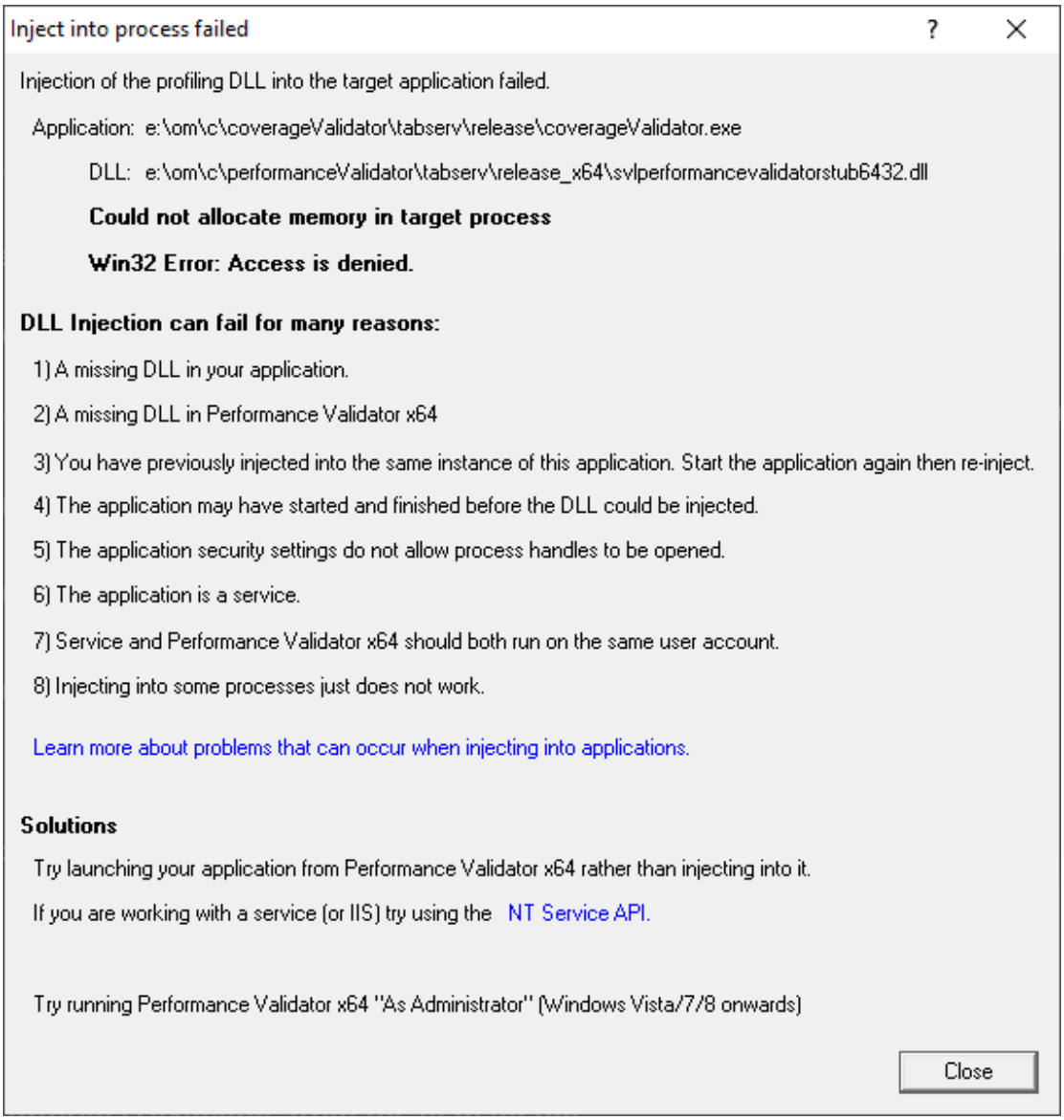
This could cause the injection to fail in unpredictable ways and you may see dialogs like that below:



One case when this dialog can occur is if the program needs to run at an elevated privilege and is waiting for the user to give permission via the UAC dialog.

Injection may fail for different reasons and you might see the following information dialog showing:

- messages relating to the specific failure
- a selection of reasons why failure might be occurring
- some possible solutions to the problem

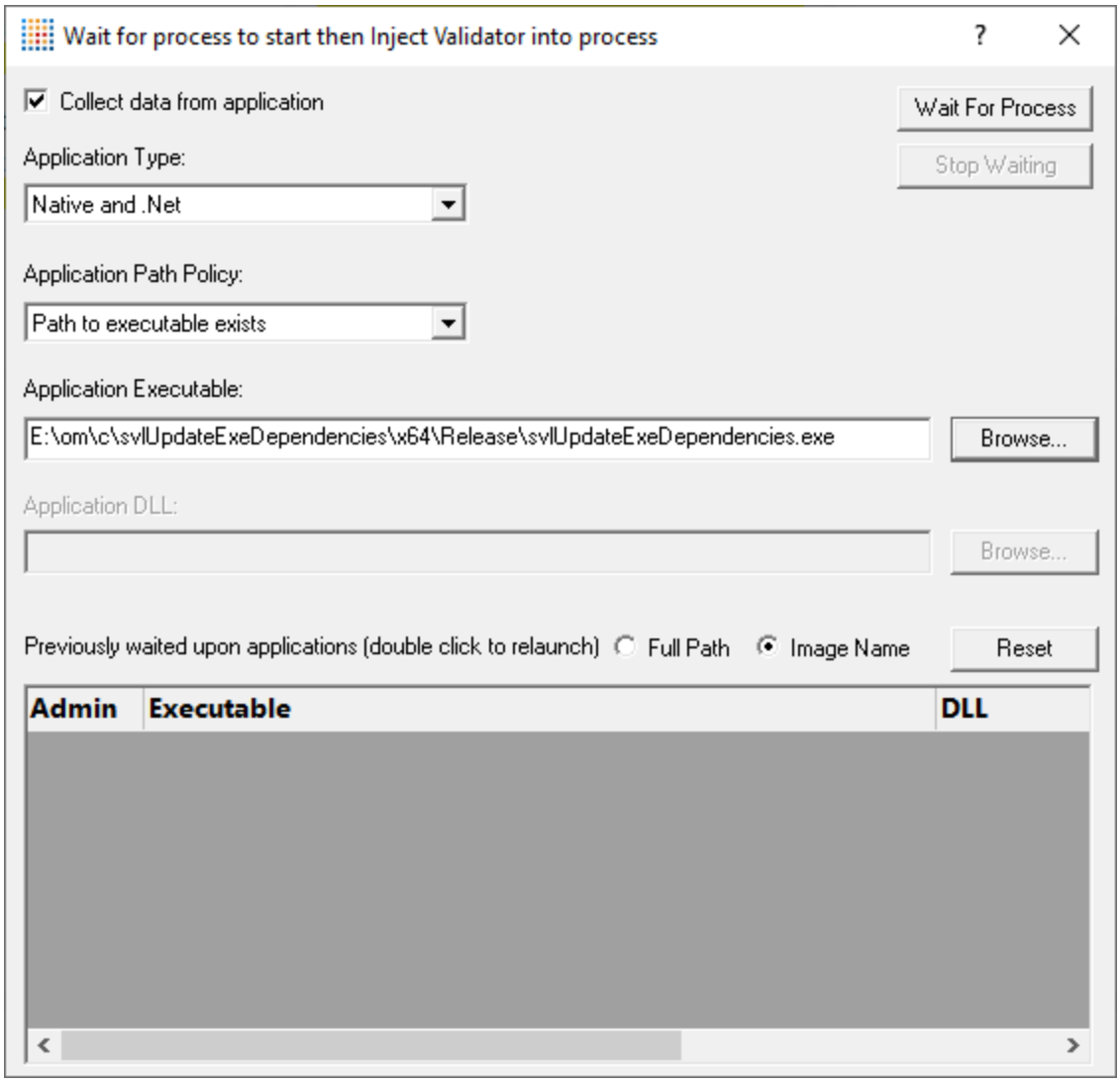


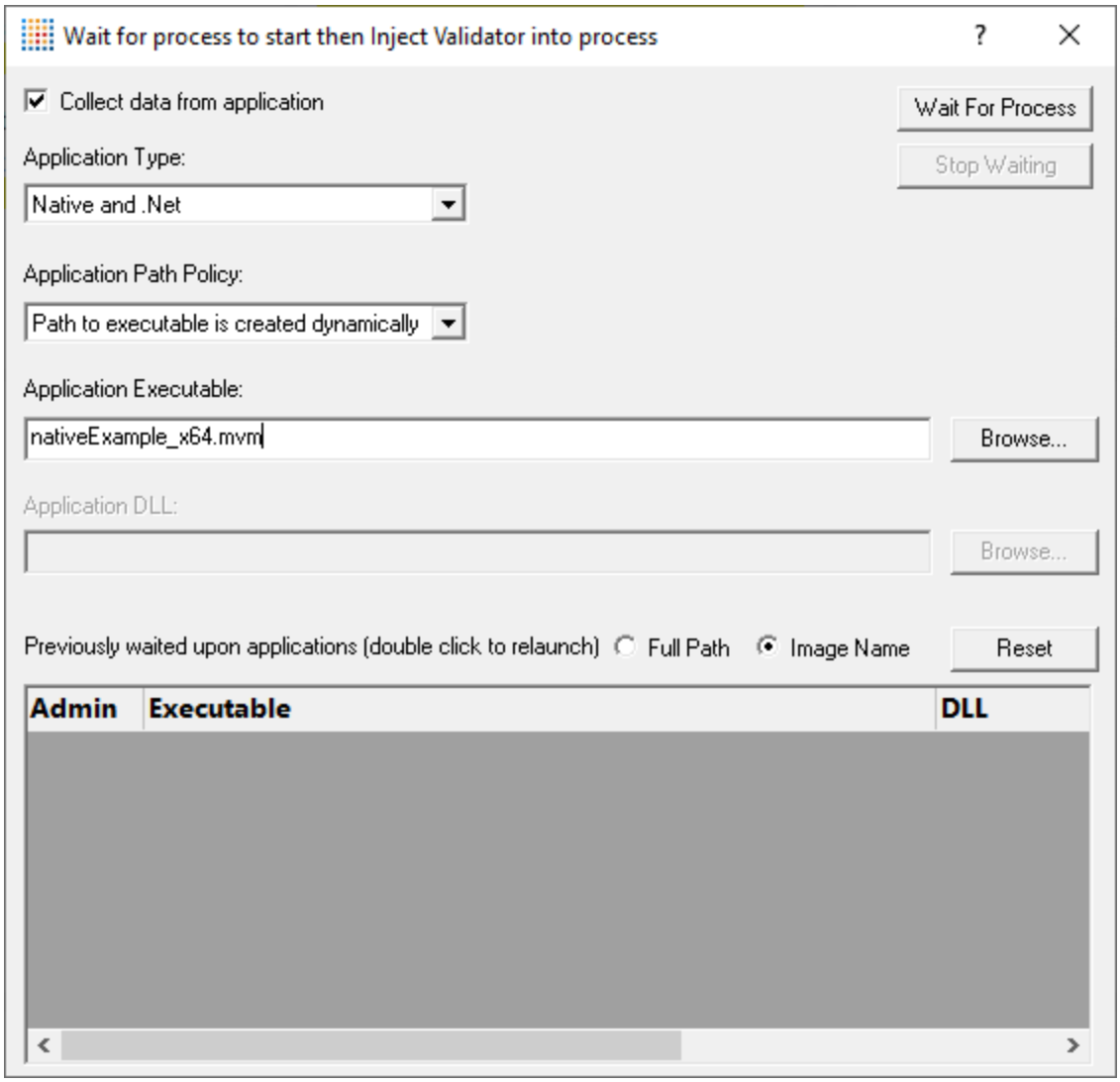
Sometimes retrying a few times might catch a better moment for attaching to the process.

➡ In the general questions see *Why might Inject or Launch fail?* for troubleshooting launch problems.

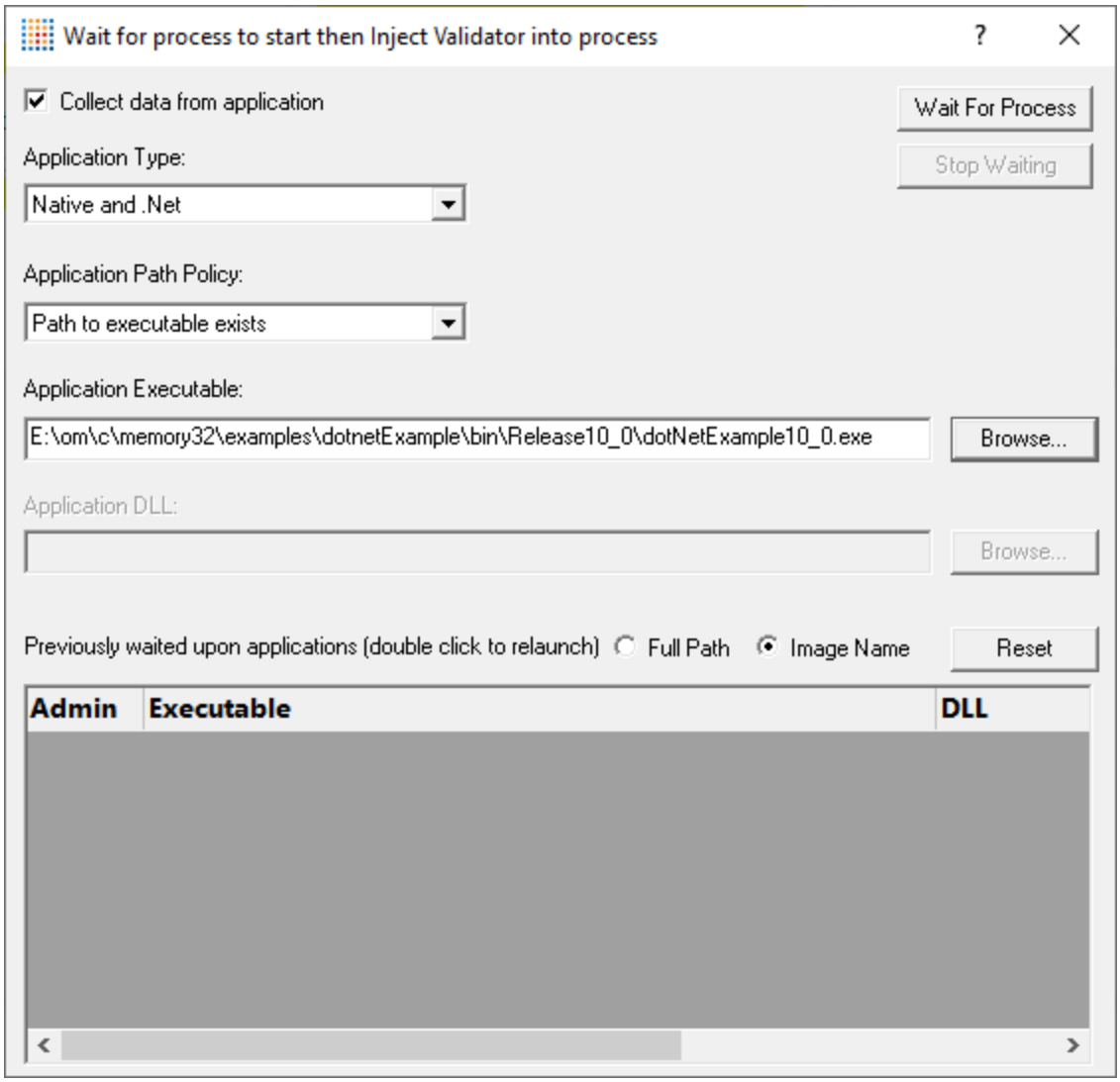
Example Dialogs

Native

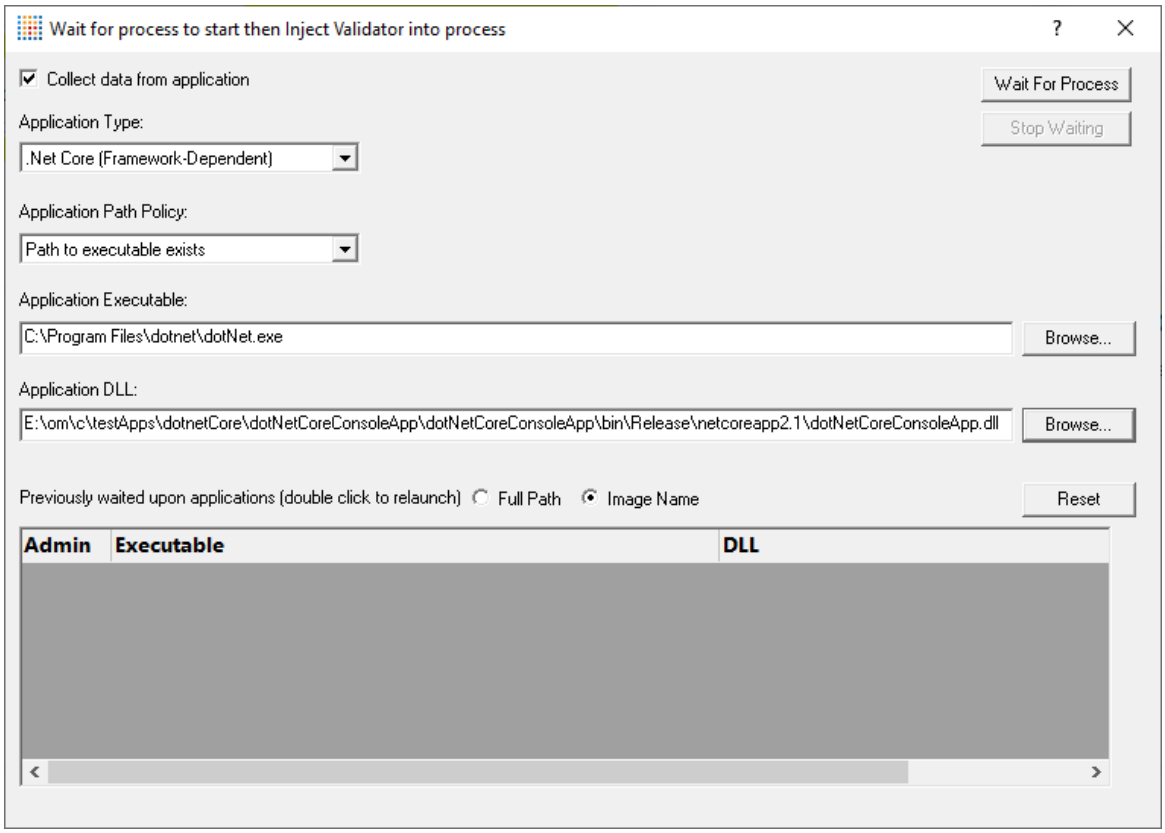




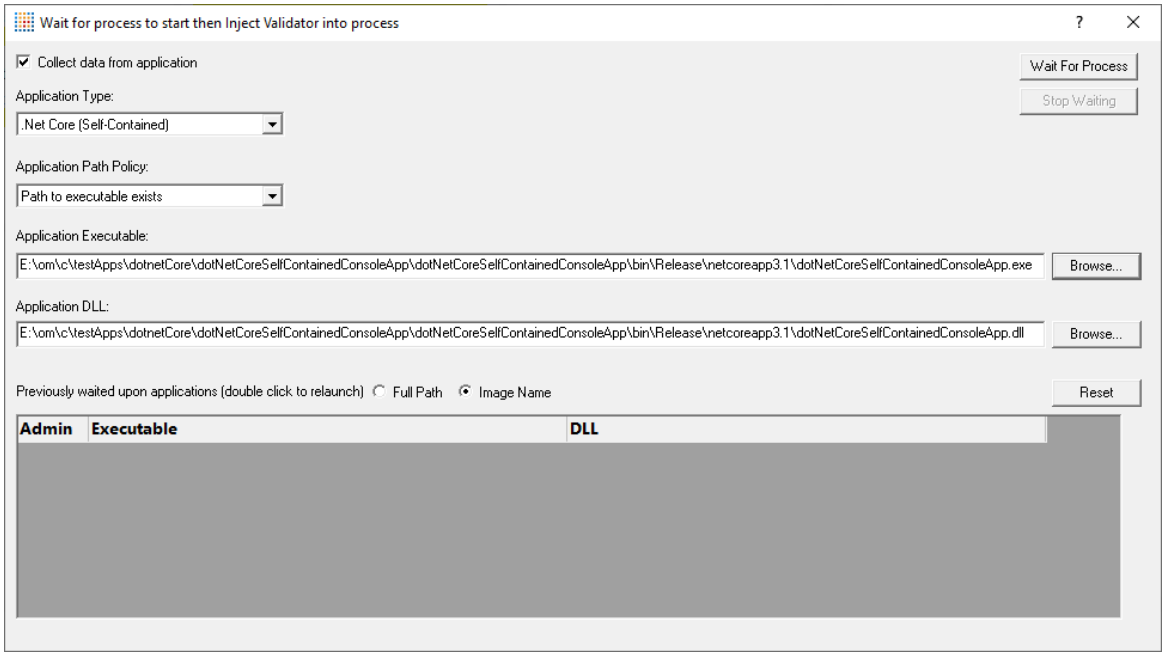
.Net



.Net Core (Framework-dependent)



.Net Core (Self-contained)



3.18.7 Monitor a service

Monitoring a service

Monitoring a service works for:

- native services
- .Net services
- mixed mode services.

Native Services

If you are working with native services you must use the NT Service API in your service *as well as* using the Monitor a service method below.

.Net Services

Memory Validator won't attach until some .Net code is executed.

If there is native code being called prior to the .Net code, Memory Validator won't monitor that code, only the native code called after the first .Net code that is called.

To monitor any native code called *prior* to your .Net code, use the NT Service API.

When working with Memory Validator and services, you still start the service the way you normally do - e.g. with the service control manager.

The code that you have embedded into your service then contacts Memory Validator, which you should have running before starting the service.

To start monitoring a service:

Launch menu › **Services** › **Monitor a service...** › shows the Monitor a service dialog below

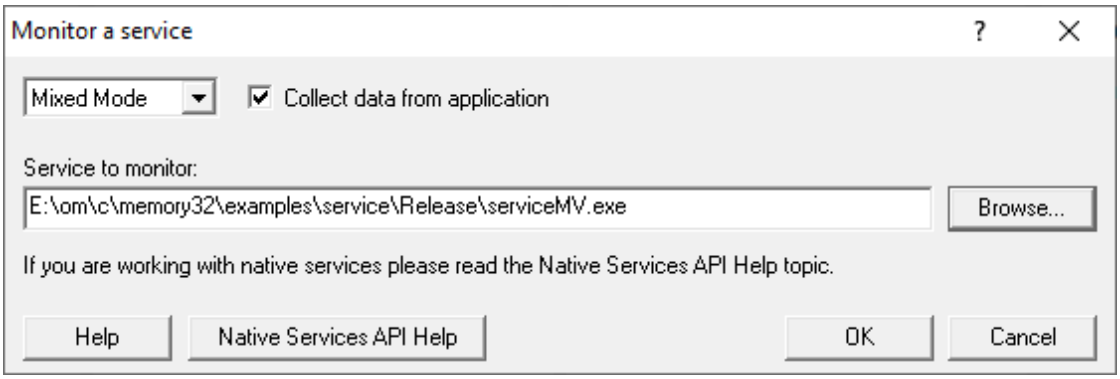
Or use the shortcut

F6 Monitor a service

The monitor a service dialog

First ensure the service is installed, but not running.

Set the service to monitor, choose whether to start collecting data right away, and click OK.



- **Service to monitor** › type or **Browse** to set the service name to monitor
- **OK** › waits for the service to start before injecting into it

Start the service in the normal manner, e.g. from the control panel, the command line or programmatically.

Data collection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?
  - **Native Only** › Ignore all .Net data in the target application.
  - **.Net Only** › Ignore all Native data in the target application.
  - **Mixed Mode** › Collect both Native and .Net data from the target application

  This setting cannot be changed after the application is launched
- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  ➜ See the section on controlling data collection for how to turn collection on and off after launch.

Examples

Example demonstrating how to monitor a service.

Example demonstrating how to monitor an application launched from a service (how to monitor any application running on a service account).

3.18.8 IIS

3.18.8.1 Monitor IIS and ISAPI

Monitoring ISAPI

Monitoring ISAPI works for:

- Native ISAPI extensions.

Native ISAPI

If you are working with native ISAPI you must use the NT Service API in your service *as well as* using the Monitor ISAPI method below.

To start monitoring ISAPI:

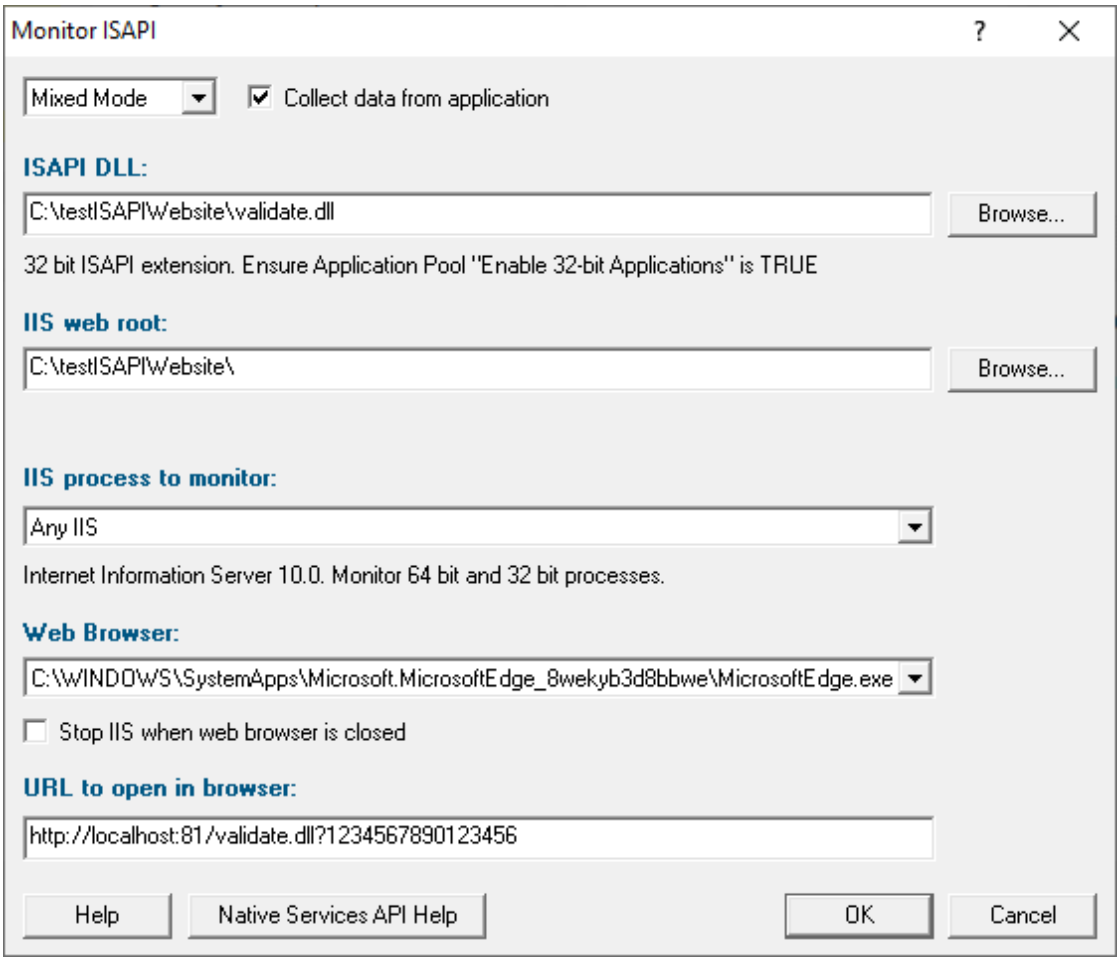
Launch menu › **Services** › **Monitor IIS and ISAPI...** › shows the Monitor ISAPI dialog below

Or use the shortcut

F7 Monitor IIS and ISAPI

The monitor ISAPI dialog

Set the dll to monitor, the web root, the IIS process, an optional web browser to use and an optional url to launch, and click OK.



- **ISAPI DLL** › type or **Browse** to set the ISAPI DLL that we're monitoring
- **IIS web** › type or **Browse** to set the web root for the IIS website we're working with
- **IIS process to monitor** › select the IIS process we're working with
- **Web Browser** › select the web browser that you're going to use to load the web page
- **URL to open in browser** › type the web page and arguments you want to load to cause the ISAPI to be loaded in IIS
- **OK** › resets IIS, setups all the variables, copies DLLs and settings into the web root and starts the web browser to load the specified web page

IIS is a protected process and can only execute, read and write files in specific directories. That's why Memory Validator copies data to the web root so that it can be read, written or executed.

Data collection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?

- **Native Only** › Ignore all .Net data in the target application.
- **.Net Only** › Ignore all Native data in the target application.
- **Mixed Mode** › Collect both Native and .Net data from the target application

This setting cannot be changed after the application is launched

- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  ➜ See the section on controlling data collection for how to turn collection on and off after launch.

3.18.8.2 Monitor IIS and ASP.Net

Monitoring ASP.Net Application (IIS)

To start monitoring ASP.Net application running in IIS:

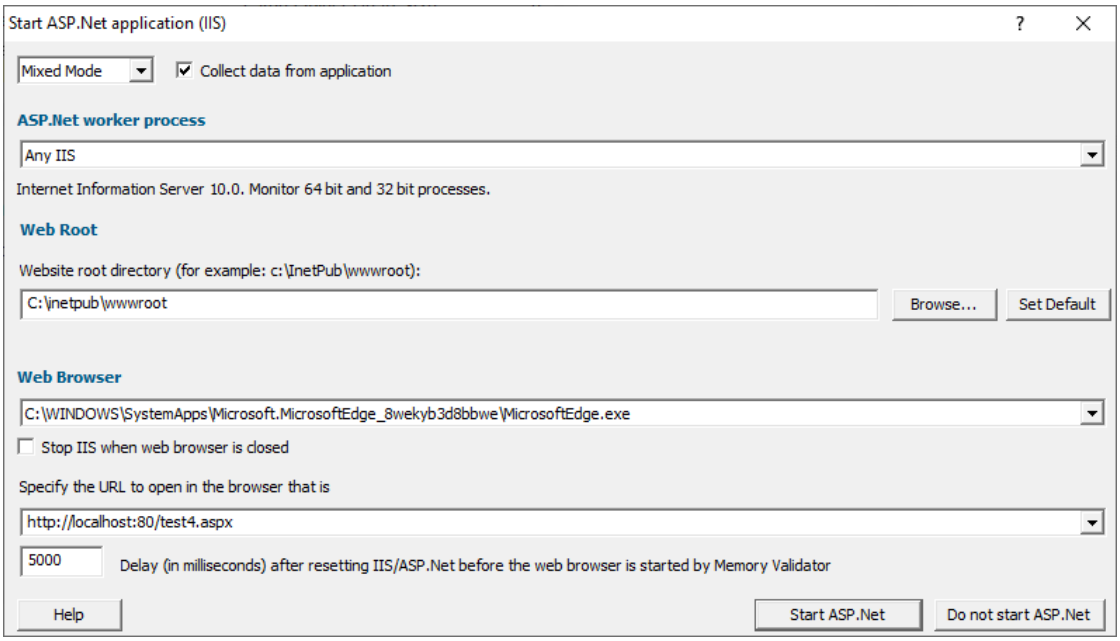
Launch menu › **Services** › **Monitor IIS and ASP.Net...** › shows the Start ASP.Net Application dialog below

Or use the shortcut

F8 Monitor IIS and ASP.Net

The Start ASP.Net application dialog

Set the asp worker process, the web root, an optional web browser to use and an optional url to launch, and click OK.



- **ASP.Net worker process** › select the IIS process we're working with. This can be any ASP.Net process or a specific one. The default is **Any IIS**.
- **IIS web** › type or **Browse** to set the web root for the IIS website we're working with
- **Web Browser** › select the web browser that you're going to use to load the web page
- **URL to open in browser** › type the web page and arguments you want to load to cause the ISAPI to be loaded in IIS
- **Start ASP.Net** › resets IIS, setups all the variables, copies DLLs and settings into the web root and starts the web browser to load the specified web page

IIS is a protected process and can only execute, read and write files in specific directories. That's why Memory Validator copies data to the web root so that it can be read, written or executed.

Data collection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?
  - **Native Only** › Ignore all .Net data in the target application.
  - **.Net Only** › Ignore all Native data in the target application.
  - **Mixed Mode** › Collect both Native and .Net data from the target application

  This setting cannot be changed after the application is launched

- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  ➡ See the section on controlling data collection for how to turn collection on and off after launch.

Slow Startup

The first time you work with IIS and Memory Validator you may experience a delay during startup. This is most likely because symbols are being downloaded from Microsoft's symbol servers to match the DLLs and assemblies on your machine.

3.18.8.3 Reset & Stop IIS

Reseting IIS

Launch menu › **Services** › **Reset IIS** › resets Internet Information Server (stops it and starts it again).

The session is not discarded when IIS is reset.

Stopping IIS

Launch menu › **Services** › **Stop IIS** › stops Internet Information Server.

The session is not discarded when IIS is stopped.

3.18.9 Web Development Server

3.18.9.1 Monitor Web Development Server and ASP.Net

Monitoring ASP.Net Application (WDS)

To start monitoring ASP.Net application running in IIS:

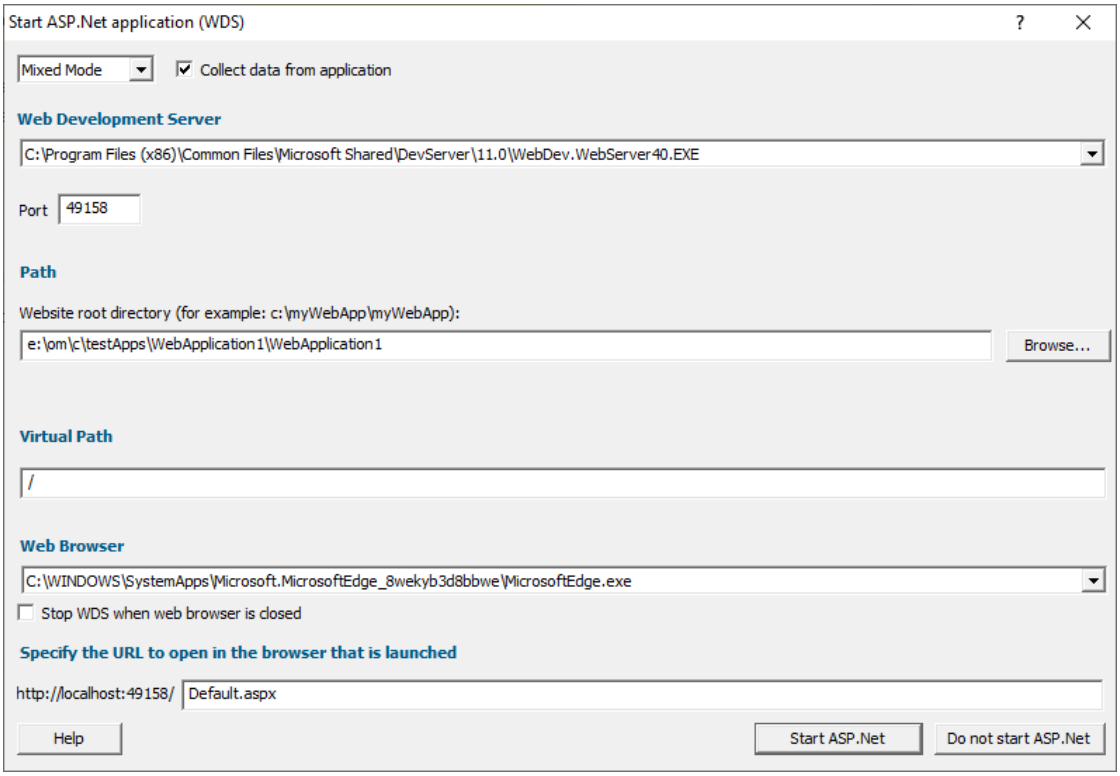
Launch menu › **Services** › **Monitor Web Development Server and ASP.Net...** › shows the Start ASP.Net Application dialog below

Or use the shortcut

F9 Monitor Web Development Server and ASP.Net

The Start ASP.Net application dialog

Set the web development server, the port to use, path to the web application, virtual path, an optional web browser to use and an optional url to launch, and click OK.



- **Web Development Server** › select the WDS process we're working with. The default is the most recent version installed on the computer.
- **Port** › select the port that the server will serve pages on. The default is 49158 (the same value that Visual Studio uses).
- **Path** › type or **Browse** to set the path to the ASP.Net application.
- **Virtual Path** › type the path on the server that corresponds to the web application. The default is /.
- **Web Browser** › select the web browser that you're going to use to load the web page.
- **URL to open in browser** › type the web page and arguments you want to load to cause the ISAPI to be loaded in IIS.
- **OK** › resets IIS, setups all the variables, copies DLLs and settings into the web root and starts the web browser to load the specified web page.

Web Development Server is not a protected process like IIS. This can make working with WDS much easier than working with IIS.

Data collection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?
  - **Native Only** › Ignore all .Net data in the target application.
  - **.Net Only** › Ignore all Native data in the target application.
  - **Mixed Mode** › Collect both Native and .Net data from the target application

  This setting cannot be changed after the application is launched
- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  ➜ See the section on controlling data collection for how to turn collection on and off after launch.

Slow Startup

The first time you work with Web Development Server and Memory Validator you may experience a delay during startup. This is most like because symbols are being downloaded from Microsoft's symbol servers to match the DLLs and assemblies on your machine.

3.18.9.2 Stop Web Development Server

Stopping Web Development Server

Launch menu › **Services** › **Stop Web Development Server** › stops Web Development Server.

The session is not discarded when Web Development Server is stopped.

3.18.10 ASP.Net Core Web Application

Enter topic text here.

3.18.10.1 Start ASP.Net Core Web Application

Monitoring ASP.Net.Core Web Application

To start monitoring ASP.Net application running in IIS:

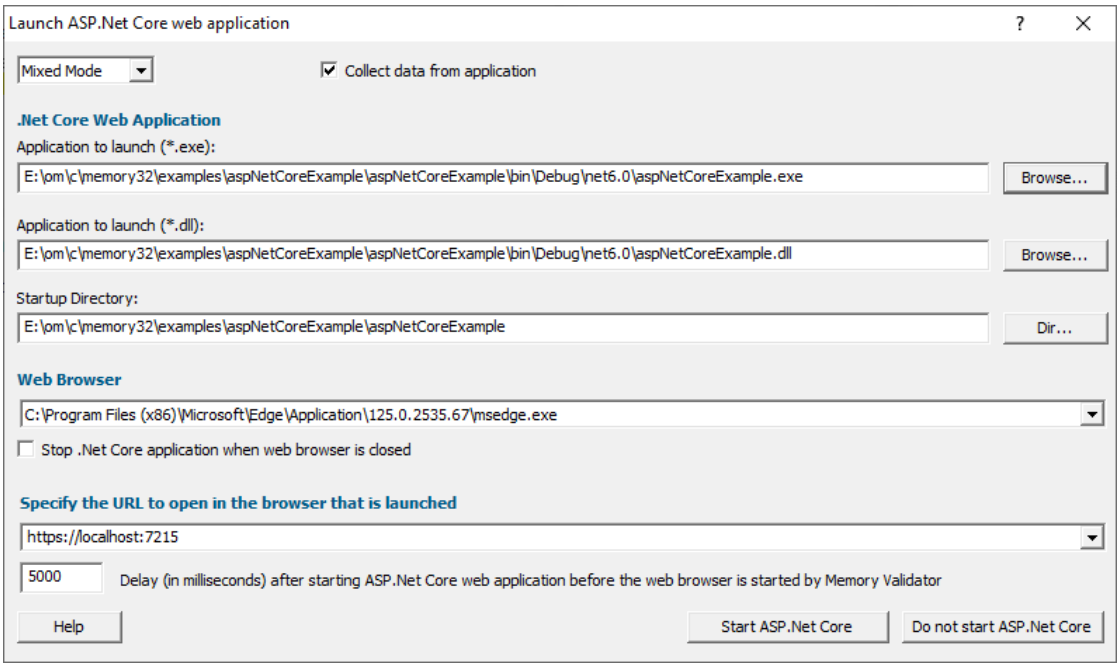
Launch menu › **Web** menu › **ASP.Net Core Web Application...** › shows the Start ASP.Net Core Web Application dialog below

Or use the shortcut

F7 Start ASP.Net Core Web Application

The Start ASP.Net Core application dialog

Set the web development server, the port to use, path to the web application, virtual path, an optional web browser to use and an optional url to launch, and click OK.



- **.Net Core Web Application (exe)** › select your ASP.Net Core web application to launch.
- **.Net Core Web Application (dll)** › select your ASP.Net Core web application to launch.
- **Startup Directory** › type or **Dir...** to set the path to the ASP.Net application.

  This value will be auto-populated based on the path you specify for your application.
- **Web Browser** › select the web browser that you're going to use to load the web page.
- **URL to open in browser** › type the web page, port and arguments you want to load to cause the ISAPI to be loaded in IIS.

  This value will be auto-populated based on the path you specify for your application.

  Example: https://localhost:7215
- **Start ASP.Net Core** › starts your ASP.Net web application, then starts the web browser to load the specified web page.

Data collection

- **Type of data collection** › Are you only interested in Native data, .Net data or both Native data and .Net data?
  - **Native Only** › Ignore all .Net data in the target application.
  - **.Net Only** › Ignore all Native data in the target application.
  - **Mixed Mode** › Collect both Native and .Net data from the target application

  This setting cannot be changed after the application is launched
- **Collect data from application** › If it's the startup procedure you want to validate, obviously start collecting data from launch.

  Depending on your application, and what you want to validate, you may want to start collecting data as soon as injection has happened, or do it later.

  If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

  If it's the startup procedure you want to validate, obviously start collecting data immediately.

  ➔ See the section on controlling data collection for how to turn collection on and off after launch.

Slow Startup

The first time you work with ASP.Net Core and Memory Validator you may experience a delay during startup. This is most like because symbols are being downloaded from Microsoft's symbol servers to match the DLLs and assemblies on your machine.

3.18.10.2 Stop ASP.Net Core Web Application

Stopping ASP.Net Core Web Application

Launch menu › **Web** menu › **Stop ASP.Net Core Web Application** › stops ASP.Net Core web application.

The session is not discarded when the ASP.Net Core web application is stopped.

3.18.11 Linking to a program

Why link Memory Validator into your program?

There are cases when you might need to link Memory Validator directly into your program.

Sometimes the normal methods of launching and injecting aren't enough to get the data needed for a particular debugging task.

For example:

- maybe the data to be monitored has already been allocated before the stub was successfully injected
- maybe there is conflict with DLLs or a timing problem stopping the injection process from work as well as normal

These situations are rare, but given the variety of different applications, can happen.

In other cases you might want to use advanced features such as the extensions DLLs or the Memory Validator API.

Linking to your program

The library that you must link to is:

- `svlMemoryValidatorStubLib.lib` for 32 bit
- `svlMemoryValidatorStubLib_x64.lib` for 64 bit

When linked and started, your program will automatically start Memory Validator.

More details on linking with the libraries can be found in the API section.

The libraries should be linked to your program's **.exe**, not to a DLL that is loaded into your program.

If your program is linked statically to the C runtime libraries, you might want to read the topic before you start.

3.18.12 Environment Variables

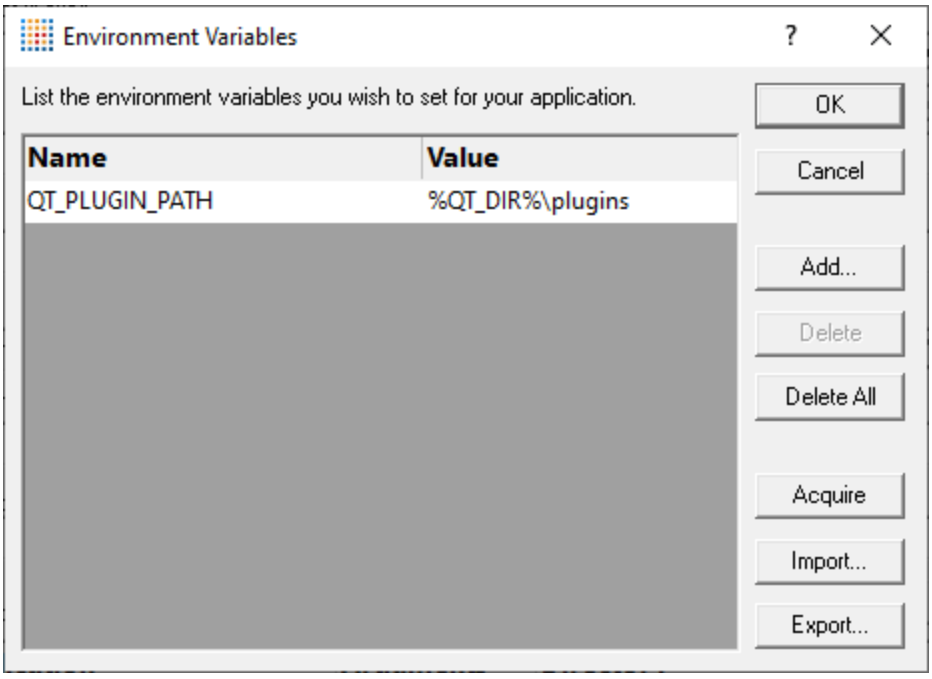
When launching an application, you might want to pass in some environment variables to your program.

The Environment Variables dialog lets you manage name/value pairs, including importing and exporting for use between programs or sessions.

The Environment Variables dialog

The dialog initially has no entries.

The example below shows the equivalent of `set QT_PLUGIN_PATH=%QTDIR%\plugins`



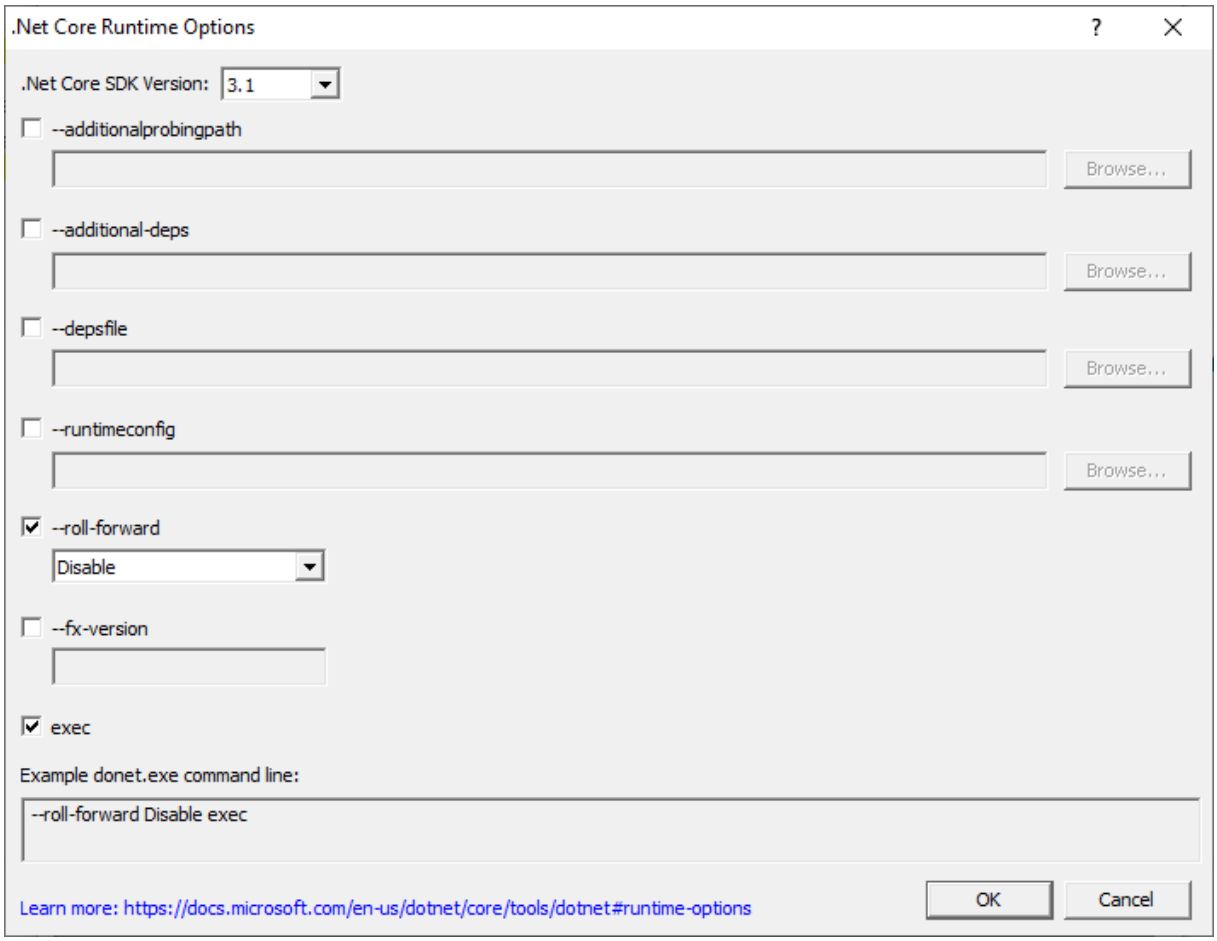
- **Add...** › adds a new item to the list › enter name in the first column, value in the second
- **Delete** › deletes a selected item in the list
- **Delete All** › clears the list
- **Acquire** › fetches all system environment variables, *adding* them to the list
- **Import...** › loads variables from a previously exported file, *adding* them to the list
- **Export...** › saves all entries in the list to a file of your choice

  The exported file is a simple ascii file with one entry per line of the form `name=value`
- **OK** › accepts all changes
- **Cancel** › ignores changes

3.18.13 .Net Core Runtime Arguments Editor

The .Net Core runtime arguments editor allows you to choose which options you pass to the .Net Core runtime.

Typically no arguments or just "exec" are passed to the runtime, assuming everything that is needed is in the same directory.



Enabling each check box will add the appropriate option to the runtime arguments.

The field can populated using the **Browse...** button to display a file or folder browser, or edited directly if you wish to add more than one path.

Some fields change depending on the SDK version chosen.

- **--additionalprobingpath** › path containing probing policy and assemblies to probe
- **--additional-deps** › path to an additional .deps.json file
- **--depsfile** › path to the deps.json file.
- **--runtimeconfig** › path to a runtimeconfig.json file
- **--roll-forward** › how to apply roll forward for .Net Core SDK 3.0 and above
- **--roll-forward-on-no-candidate-fx** › how to apply roll forward for .Net Core SDK 2.x
- **--fx-version** › version of .Net runtime to use to run the application
- **exec** › adds the "exec" keyword at the end of the arguments list

If you don't know what these options are you should read the .Net runtime options document shown in the link below before changing any of them.

We cannot advise you on how to set these values - except to say that if you're not setting them when using .Net Core outside of this software tool you shouldn't be setting them when using this software tool.

.Net Core Runtime Options

.Net Core provides some options that allow you to control how .Net Core performs.

Detailed information on these options is provided by Microsoft at https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet#runtime-options.

3.19 Stopping your target program

Stopping the application

You can stop or kill your program at any time using the task manager, or debugger.

You can also stop your program from within Memory Validator.

File menu › **Abandon Application...** › stop the target program

or click on the red cross icon on the session toolbar.

The target program is ended using `ExitProcess()` from inside the stub.

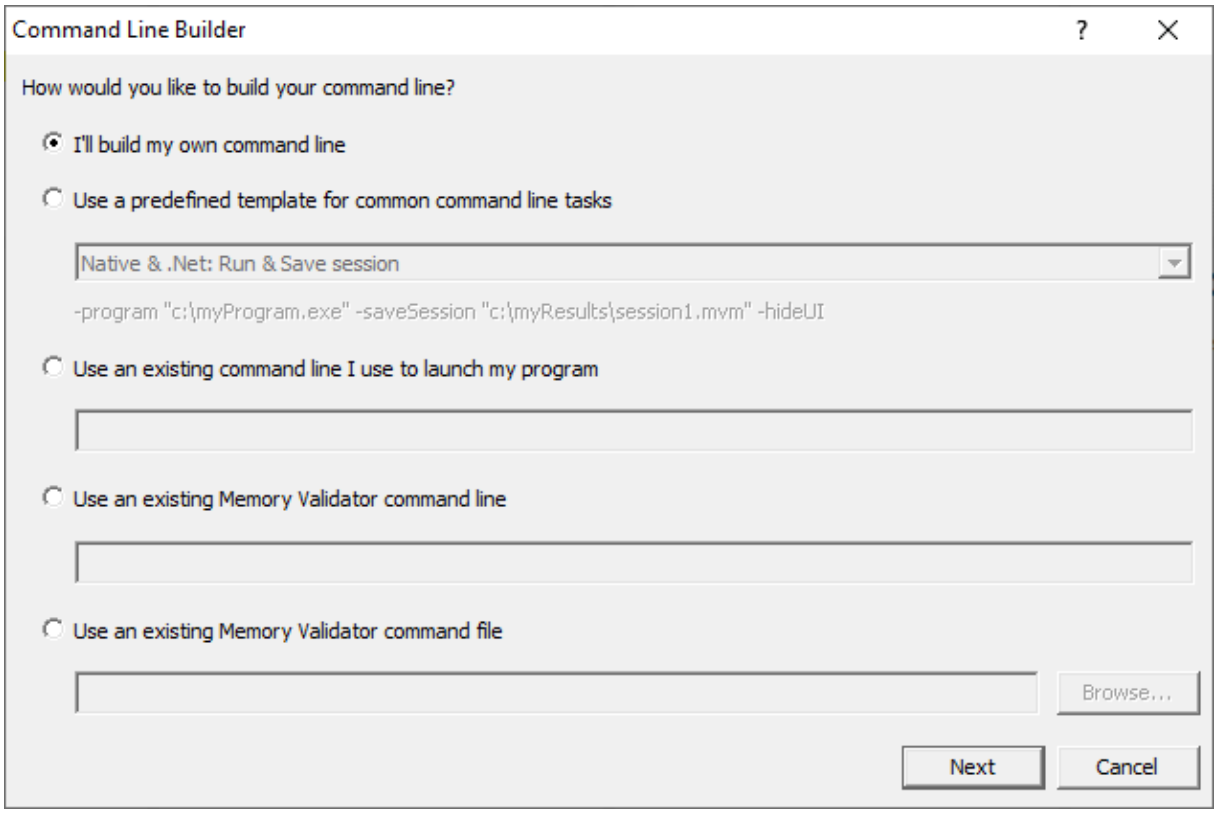
Since the session is discarded, using Memory Validator to stop the target program is usually quicker and more convenient than external stop methods.

➡ You can easily re-launch the program again using the same settings as before.

3.20 Command Line Builder

The command line builder helps you create command lines with valid options.

The command line builder is a two stage process, the first stage helping your choose how you want to build the command line, and the second stage actually building the command line based on the choices in the first stage.



There are five options for building your command line:

- **I'll build my own** › you'll build your command line from scratch, with no predefined options
- **Use a predefined template** › choose from a list of predefined command lines that you can customize

The predefined templates cover a range of tasks you may want to perform from the command line. These include running sessions, saving sessions, exporting to HTML, XML, and comparing memory leak data.

Examples are provided for both Native and .Net applications, and .Net Core applications.

- **Use an existing command line** › use the command line you use to start the tool you want to detect memory leaks for

Example: `e:\dev\paintpot\release\paintpot.exe e:\testimages\venn.png -invert -mirror -phaseChange`

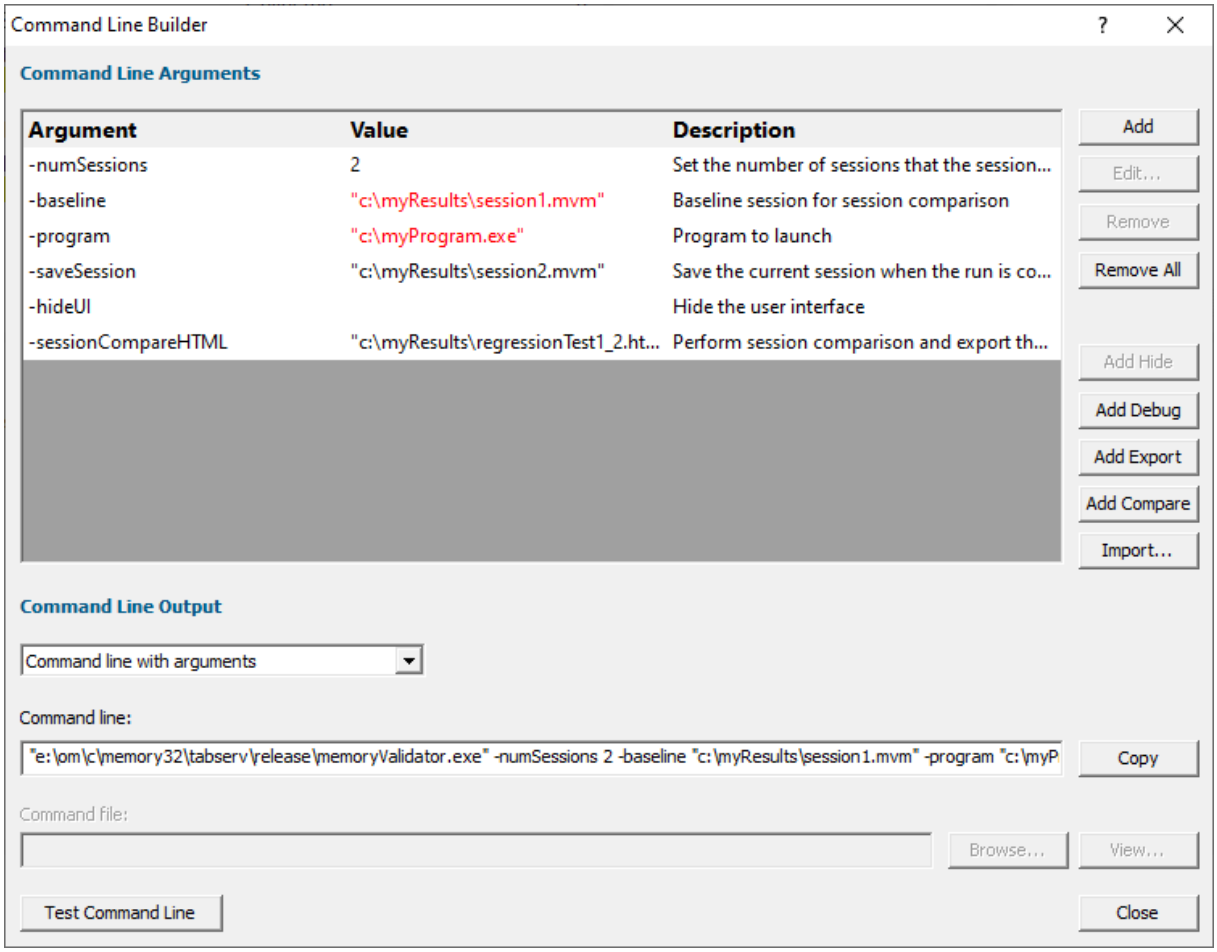
- **Use an existing Memory Validator command line** › use an existing Memory Validator command line and customize that

Example: `-program e:\dev\paintpot\release\paintpot.exe -hideUI -exportAsHTML e:\testResults\gannt.html -allArgs e:\testimages\gantt.png -inflate:3`

- **Use an existing Memory Validator command file** › use an existing Memory Validator command file and customize that

  Example: `-commandFile e:\commandFiles\paintpot_gantt.cf`

When you have made your choice the **Next** button moves you to the customization part of the command line builder.



The image above shows the command line builder populated with one of the predefined template choices. You can see a few entries refer to directories and files that do not exist on disk (they are red).

These are items you will need to customize to match the program you are testing.

For example, the **-baseline** session will need to have been created (by saving an appropriate session from Memory Validator) created before you run the test.

Any entries that will only exist after they have been created by the test will also be shown in red.

Editing

To edit an argument, double click the argument. A combo box will display a list of valid arguments you can choose.

To edit a value, double the value. If the argument type has a list of known values a combo box will be provided, directories will display a directory chooser, files will display a file chooser, numbers will only allow numeric editing. All other values will be edited as text.

- **Add** › add a new argument to the grid
- **Remove** › remove the selected item
- **Remove All** › removes all items in the grid
- **Add Hide** › adds a -hideUI argument which will cause Memory Validator to hidden when running. Memory Validator will close after the target program finishes running
- **Add Debug** › adds various arguments which will cause Memory Validator to display error messages if there are problems with the command line.
- **Add Export** › adds export options that will cause Memory Validator to export html and/or xml reports after the target program finishes running
- **Add Compare** › adds compare options that will cause Memory Validator to load a session and compare sessions after the target program finishes running
- **Import...** › you can import a command file, the contents of which will replace all the items in the grid

Command Line Output

There are two command line output styles.

- **Command line with arguments** › generates a command line containing all arguments and values shown in the grid
- **Command line with command file** › generates a command file containing all arguments and values shown in the grid, and a command line that references the command file

  When this option is chosen the command file edit field and the **Browse...** and **View...** buttons are enabled, allowing you to specify a command file name, and to view it's contents.

  If a command file has not been specified when this option is selected you will be prompted to select a name for the command file.

  When the command file name is selected the command file will be created with the arguments and values shown in the grid.
- **Copy** › copies the command line to the clipboard so that you can paste the command line in cmd prompts, batch files and automated scripts (Jenkins etc)

- **Browse...** › opens a Windows file dialog to allow you to specify the command file location
- **View...** › opens the command file using the Windows shell, this allows you to view the command file in your favourite editor

Testing

If you wish to test the command line, you have two options:

- **Manual test** › use the **Copy** button to copy the command line, then paste it into a cmd prompt and press return.
- **Test Command Line** › a new instance Memory Validator is started with the specified command line.

3.21 Data Collection

Collecting data

Once you've launched or injected into a program, you can stop and start data collection whilst the program is running.

This is a high level switch that controls *all* data collection, regardless of any other settings.

With data collection off, the target program runs at close to normal speed.

Temporarily turning off collection can be a good idea if you need to take actions to get the program into the right state for validation.

You can also turn data off from the start and only turn it on when you need it.

Starting and stopping data collection

File menu › **Start collecting data...** › starts collecting data immediately

or click on the green icon on the session toolbar to start collecting.

File menu › **Stop collecting data...** › stops collecting

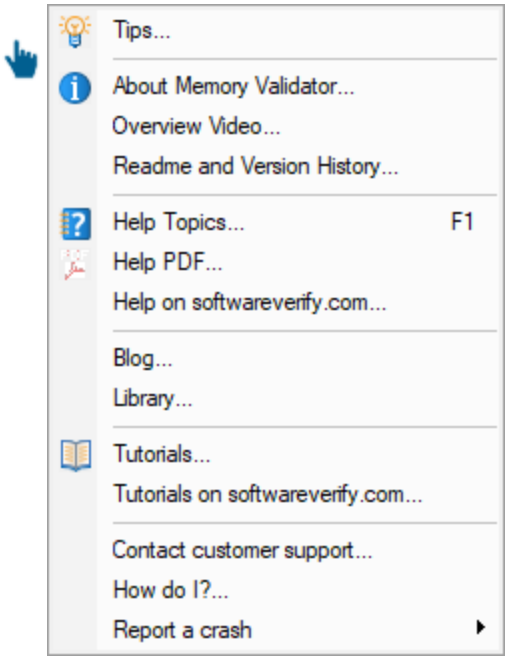
or click on the red icon on the session toolbar to stop.

3.22 Help

The help menu

The help menu provides access to useful help, tips and tutorials.

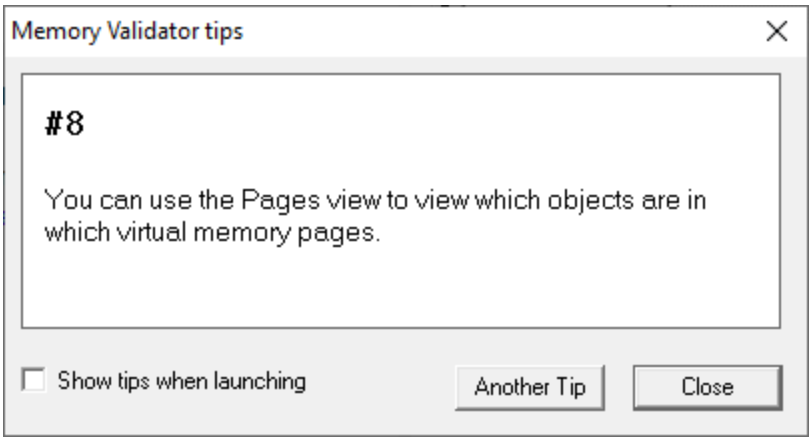
Each item is covered briefly below, in menu order.



Tips

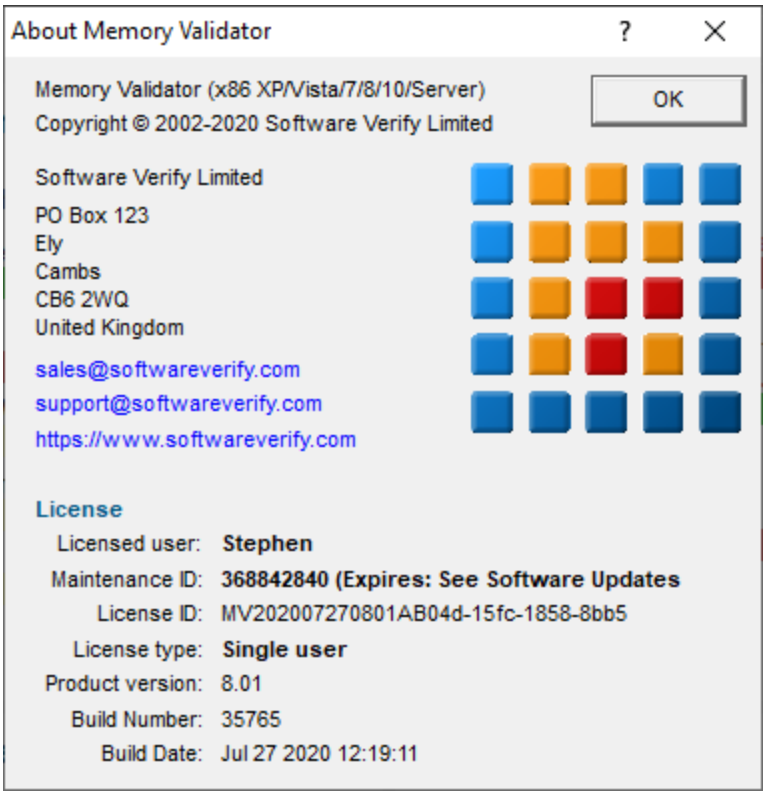
Help menu › **Tips...** › shows the tip dialog where you can browse tips in random order

Here you can also choose whether to display the tips dialog while launching programs.



About box

Help menu › **About Memory Validator...** › shows contact and copyright information, as well as details of your license

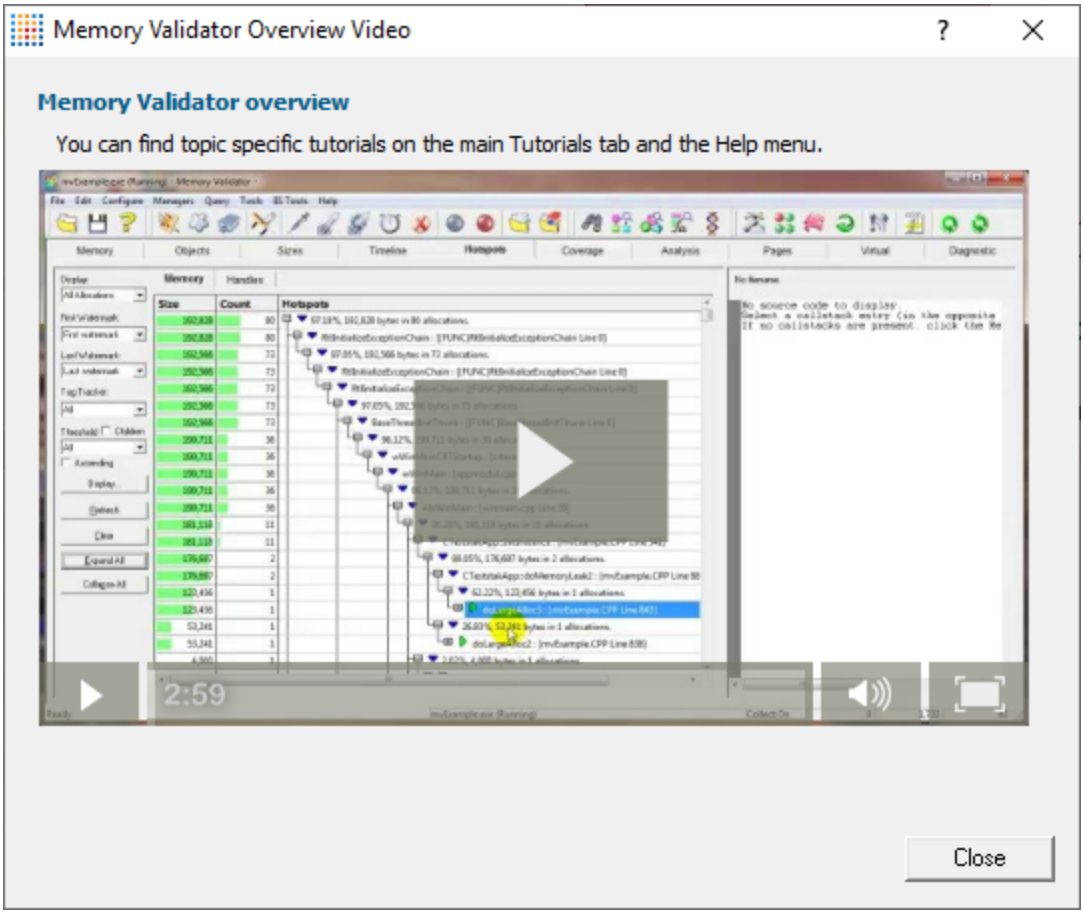


Overview Video

Help menu › **Overview Video...** › opens a new dialog showing a short video

The video has sound and does not play automatically.

The video is also available on the product website. Visit https://www.softwareverify.com/products.php and find the product link for Memory Validator.



Readme and version history

Help menu › **Readme...** › opens the readme.html (from your installation) in your browser.

The readme file contains all the latest information about Memory Validator including:

- basic information about getting started and where to go for support
- known issues
- version history

To see what's changed since the version you have installed see the latest version history.

Help HTML

Help menu › **Help topics...** › shows the HTML help dialog

You might be reading this right now!.

Or click on the question mark icon on the standard toolbar:

The F1 key also shows the help, but has the added bonus of jumping directly to the page relevant for the current view or dialog.

We occasionally get reports of customers seeing exception errors while viewing the HTML help. Unfortunately, we don't have a solution for this!

Help PDF

Help menu > **Help PDF** > shows the PDF version of this help

You will need a suitable PDF reader such as Adobe Acrobat Reader, but do beware of unwanted add-on installs.

PDF help for *all* our products are online.

Help on softwareverify.com

Help menu > **Help on softwareverify.com** > shows the online version of this help

Blog

Help menu > **Blog** > shows the Software Verify Blog.

Library

Help menu > **Library** > shows the Software Verify Library - all our best articles organised into related topics for easy access.

Tutorials

The tutorials are intended to guide you through learning how to use aspects of Memory Validator.

All tutorials are available online in the form of short videos and examples covering popular topics.

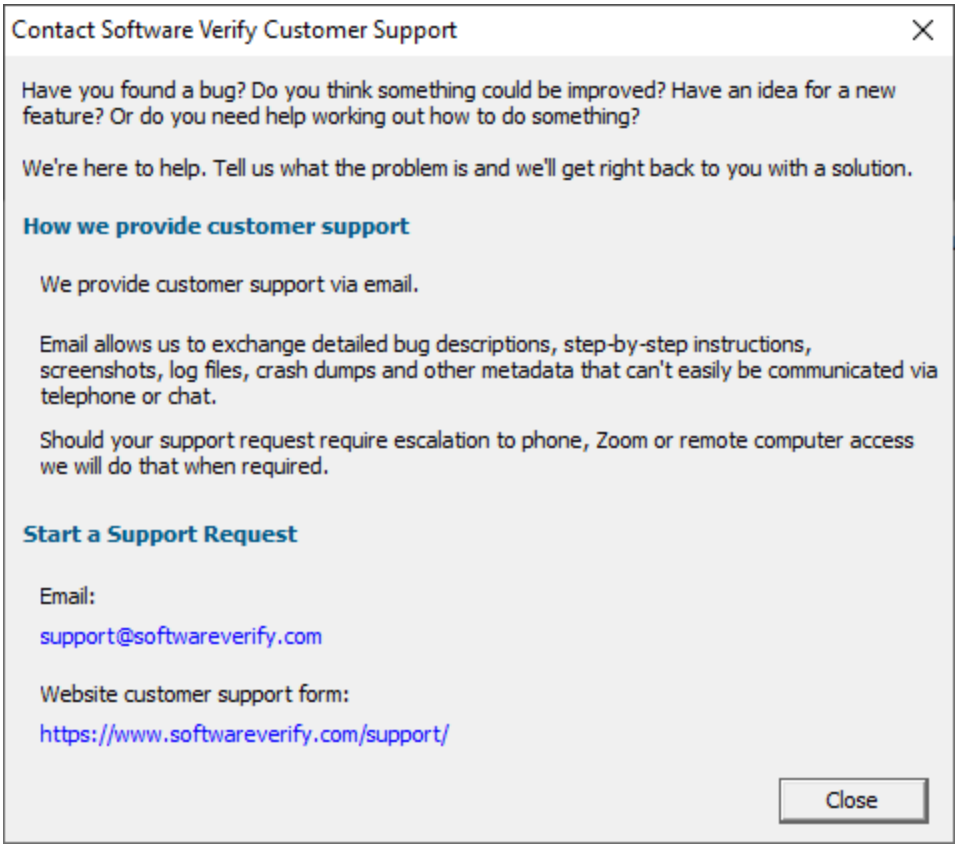
Help menu > **Tutorial** > simply selects the Tutorial tab to show a list of the tutorials

Double click on the row of a tutorial in the list to open it in a browser.

Help menu › **Tutorials on softwareverify.com...** › opens the online tutorials in a browser

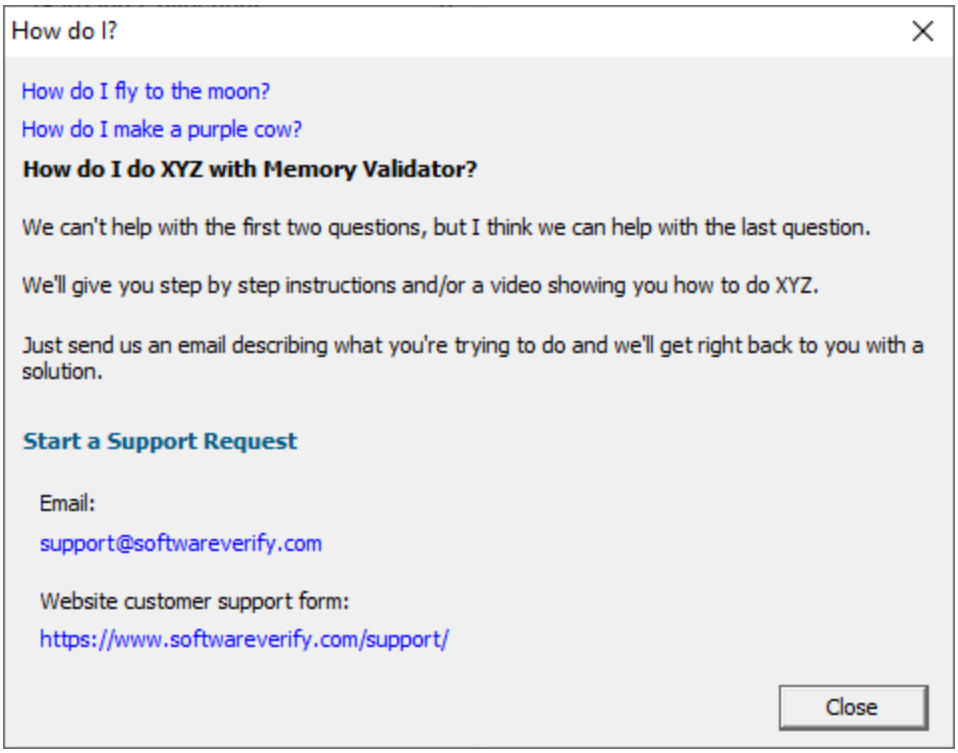
Contact customer support

Help menu › **Contact customer support** › shows the Contact customer support dialog.



How do I?

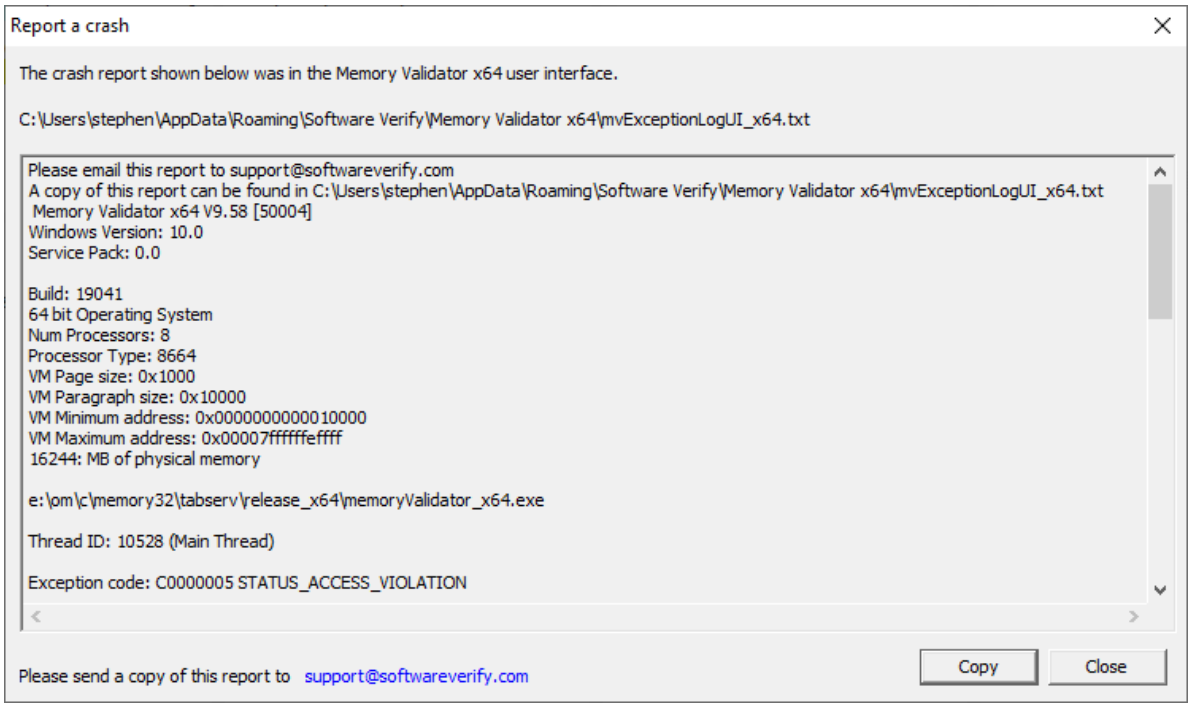
Help menu › **How do I?** › shows the How do I? dialog.



Report a crash

Help menu › **Report a crash** › displays the options for reporting a crash.

If an exception report for the Memory Validator user interface, or an exception report for an application that Memory Validator was monitoring is available it will be displayed with options to copy it to the clipboard and contact customer support at Software Verify.



3.23 Software updates

This topic covers the three items on the Software Updates menu:

- checking for software updates
- configuring your update schedule
- renewing your software maintenance
- setting your software update credentials
- setting the software update directory

Software updates

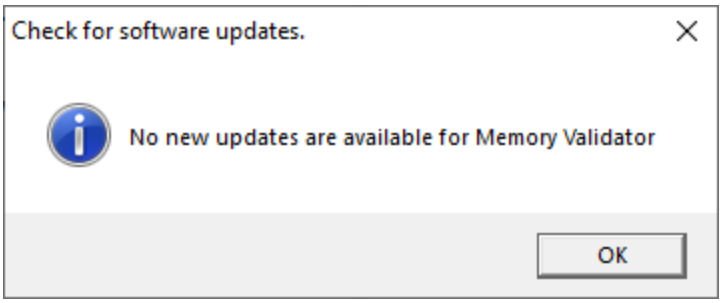
If you've been notified of a new software release to Memory Validator or just want to see if there's a new version, this feature makes it easy to update.

Software Updates menu › **Check for software updates** › checks for updates and shows the software update dialog if any exist

An internet connection is needed to be able to make contact with our servers.

Before updating the software, close the help manual, and end any active session by closing target programs.

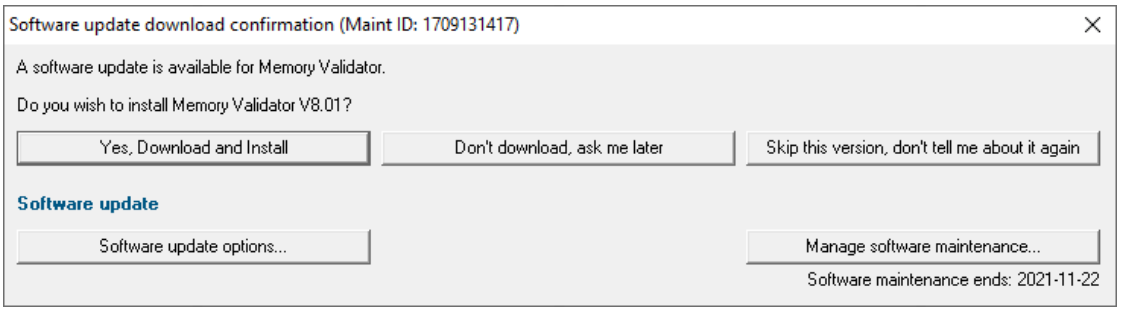
If no updates are available, you'll just see this message:



Note that evaluation versions cannot be updated.

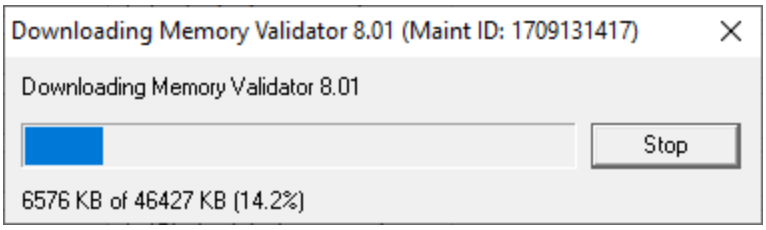
Software Update dialog

If a software update is available for Memory Validator you'll see the software update dialog, unless your maintenance has expired.



- **Download and install** › prompts you for login details if not known, and then downloads the update, showing progress

  You may be asked for your login credentials, which you'll have received when you purchased Memory Validator.

  Once the update has downloaded, Memory Validator will close, run the installer, and restart.

  You can stop the download at any time, if necessary.

- **Don't download** › Doesn't download, but you'll be prompted for it again next time you start Memory Validator

- **Skip this version** › Doesn't download the update and doesn't bother you again until there's an even newer update
- **Software update options...** › edit the software update schedule
- **Manage software maintenance...** › opens your browser ready for maintenance renewal

Problems downloading or installing?

If for whatever reason, automatic download and installation fails to complete:

- Log in to https://www.softwareverify.com/authdownload.php with the details provided when you purchased Memory Validator
- Download the latest installer manually, via one of the .exe, .xyz or .zip files that are available

Make some checks for possible scenarios where files may be locked by Memory Validator as follows:

- Ensure any open sessions are completed
- Ensure any target programs started by Memory Validator are closed
- Ensure Memory Validator and its help manual is also closed
- Ensure any error dialogs from the previous installation are closed

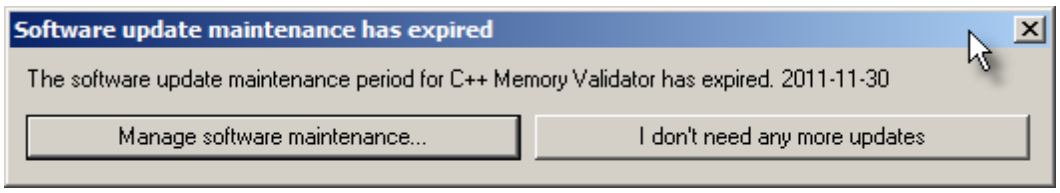
Have your license details handy as you may need to copy information into the license dialog

You should now be ready to run the new version.

Software maintenance expiry

If the software maintenance period has expired you won't be able to automatically update Memory Validator as above.

Instead, you'll see the software update maintenance expiry dialog:



You can manage your software maintenance or choose to stop receiving any more software updates.

Software update schedule

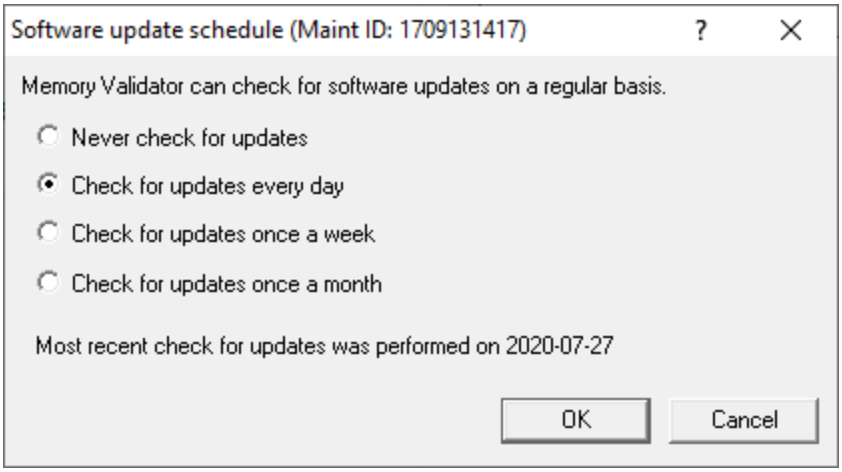
Memory Validator can automatically check to see if a new version of Memory Validator is available for downloading.

Software Updates menu › **Configure software updates** › shows the software update schedule dialog

The update options are:

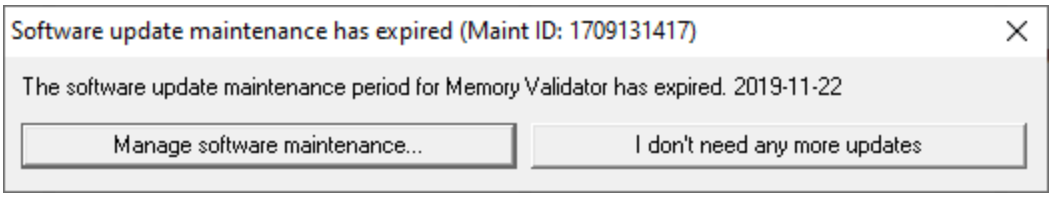
- never check for updates
- check daily (the default)
- check weekly
- check monthly

The most recent check for updates is shown at the bottom.



Managing software maintenance

Software Updates menu › **Renew software updates** › shows the software update maintenance renewal dialog



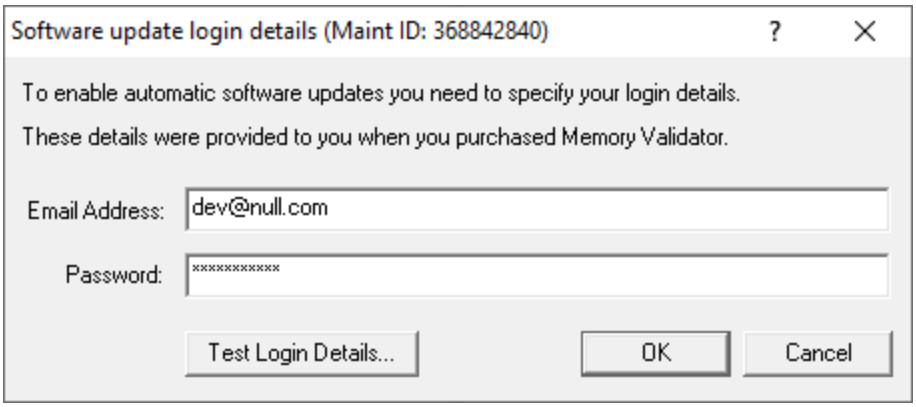
- **Renew software maintenance** › Opens your browser, logging you in to our website from which you can purchase maintenance

  Your maintenance expiry date is shown. If you don't need to do anything just **Close** the dialog.

Managing software update credentials

You can configure your software update credentials within the application.

Software Updates menu › **Set software update credentials** › shows the Software update login details dialog

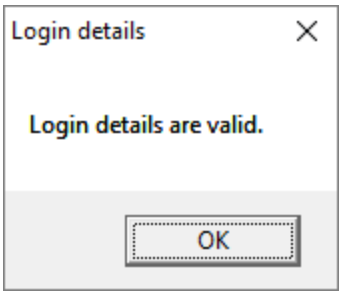


The text will be shown in red if the email address looks incorrectly formatted.

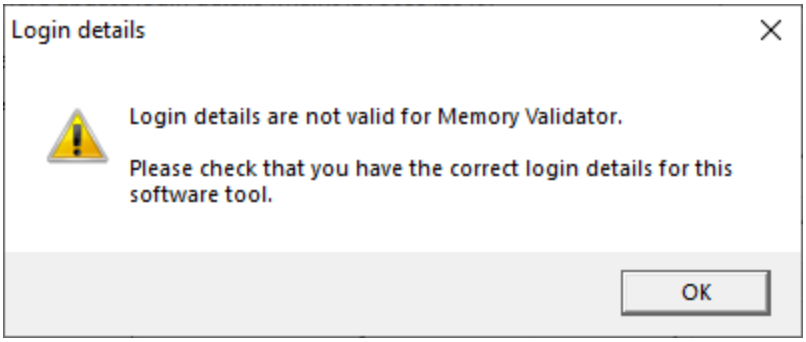
Testing the login details checks they're valid:

- **Test login details** › check your entered details are valid (requires an internet connection)

  Valid details will be confirmed:



  Invalid details may mean you entered credentials for another application in the Validator suite, or they could have been entered incorrectly.



  You should have received the correct credentials when you purchased Memory Validator, or with any software update emails.

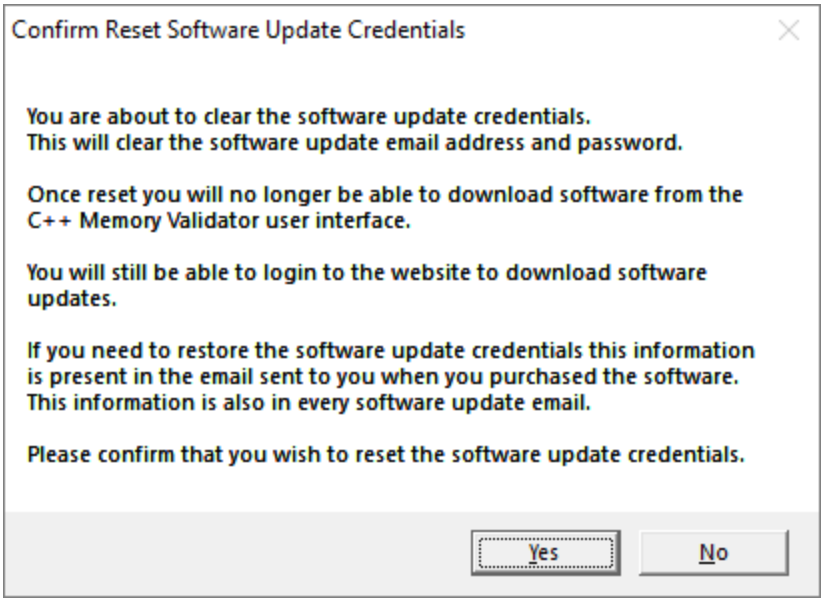
  If you experience problems, check with your system administrator or contact Software Verify.

If you need to clear the update credentials, you can do this directly from the menu.

Software Updates menu › **Reset software update credentials** › clears the email and password details stored in the application

You will be asked to confirm the reset. After resetting the credentials, no software updates will occur.

If you later need to restore your credentials, you should have received that information when you purchased Memory Validator, or with any software update emails.

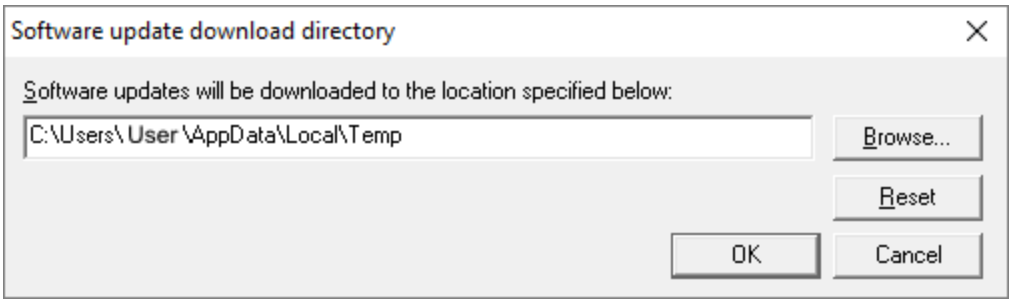


Software update directory

It's important to be able to specify where software updates are downloaded to because of potential security risks that may arise from allowing the `TMP` directory to be executable. For example, to counteract security threats it's possible that account ownership permissions or antivirus software blocks program execution directly from the `TMP` directory.

The `TMP` directory is the default location but if for whatever reason you're not comfortable with that, you can specify your preferred download directory. This allows you to set permissions for `TMP` to deny execute privileges if you wish.

Software Updates menu › **Set software update directory** › shows the Software update download directory dialog



An invalid directory will show the path in red and will not be accepted until a valid folder is entered.

Example reasons for invalid directories include:

- the directory doesn't exist
- the directory doesn't have write privilege (update can't be downloaded)
- the directory doesn't have execute privilege (downloaded update can't be run)

When modifying the download directory, you should ensure the directory will continue to be valid. Updates may no longer occur if the download location is later invalidated.

- **Reset** › reverts the download location to the user's `TMP` directory

  The default location is `c:\users\[username]\AppData\Local\Temp`

Part

4 Tag Tracking

Tag trackers

Tag trackers are a powerful method of classifying data at runtime.

An allocation or group of allocations can be marked with a tag that can later be used to filter allocation events.

Tag tracking is implemented using a `svlDataTracker` C++ class which allows the tag tracking to have automatic scope on the stack, so that cleanup is not required.

Even if you're not using C++, do continue reading, as there are API functions you can call to get the same effect.

4.1 Data Tracking with svlDataTracker

Tag trackers allow allocations to be marked with a tag that can later be used to filter allocation events.

Examples of using tag trackers in code

Here's a simple example function using tag trackers:

```
void exampleFunc()
{
   int   i;

   for(i = 0; i < getNumWorkUnits(); i++)
   {
      svlDataTracker   trackWorkUnits("workunit"); // Set current tracker to be 'workunit'

      processWorkUnits(i);
   }
}
```

In this example all memory allocations, reallocations and deallocations that happen inside `processWorkUnits()` have the tag `workunit`.

In the more complex code fragment below, three tags are used to classify the actions of the functions called.

Note that when the `Flowers` tracker starts, the `Cats` tracker is *temporarily* suspended, to be resumed when the `Flowers` tracker falls out of scope.

```
createAFish("Pike");                    // no tracker

{
   svlDataTracker   tracker_cats("Cats");

   createAnAnimal("Lion");              // cats tracker
   createAnAnimal("Tiger");             // cats tracker
   createAnAnimal("Panther");           // cats tracker

   {
      svlDataTracker   tracker_flowers("Flowers");

      createAFlower("Daffodil");        // flowers tracker
      createAFlower("Rose");            // flowers tracker

      {
         svlDataTracker   tracker_trees("Trees");

         createATree("Oak");            // trees tracker
         createATree("Sycamore");       // trees tracker
         createATree("Ash");            // trees tracker
         createATree("Horse Chestnut");// trees tracker
      }

      createAFlower("Lily");            // flowers tracker
   }

   createAnAnimal("Leopard");           // cats tracker
   createAnAnimal("Cheetah");           // cats tracker
   createAnAnimal("Cougar");            // cats tracker
}

createAFish("Salmon");                  // no tracker
```

svlDataTracker class usage

The svlDataTracker class used in the examples above is defined in `svlDataTracker.h` and `svlDataTracker.cpp` which should be in the `API` folder of the Memory Validator installation directory.

The class is written so as not to include any dependencies on Memory Validator into your application.

The class will only activate the Memory Validator functionality if using Memory Validator with your application, and otherwise does nothing.

To use svlDataTracker in your application:

- `#include svlDataTracker.h` into each file you will be using svlDataTracker
- use the svlDataTracker class as shown above

 Do not *dynamically* create svlDataTracker objects as this will break the tag scoping rules and make them behave unpredictably.

Not using C++?

If you're not using C++, here's a couple of options:

An object oriented solution

If you're using an object oriented language such as Delphi you can create a Delphi equivalent to svlDataTracker.

You'll find the source code for svlDataTracker in `svlDataTracker.cpp` and this should be easy enough to understand for you to be able to create a suitable implementation for your object orientated language.

An API solution

The C++ and object orientated solutions both use the Memory Validator API.

You can also link your own program with the Memory Validator stub, and use the API directly.

The API provides two functions (among others) exported from svlMemoryValidatorStub.dll:

- `mvPushTracker(char *trackerName)` › pushes a tag tracker on to the tag stack, making it the 'current' tag
- `mvPopTracker()` › pops the current tag tracker from the top of the tag stack, making the next one (if any) current

You can find out more about the API solution for tag tracking as well as other API functions in these topics:

- The Native API
- Calling the API functions using GetProcAddress
- Calling the tag tracker functions above

Part V

5 Command Line / Regression Testing

Manual or automated testing

There are two ways you can use Memory Validator for regression testing:

- Manual (interactive) testing of a session with any previous session

  Interactively comparing sessions allows a flexible approach in looking for regressions (or improvements!).

  This frequently forms part of the day-to-day development process.

- Automated testing, comparing against a *baseline* session

  Compare the results of multiple independent test runs against the baseline and create a suite of results to act on.

  Often part of a continuous quality assurance program, especially where test suites can run to many thousands of tests.

Need some help building the command line?

If you find creating command lines from nothing to be a bit daunting we've created a Command Line Builder tool to help you build command lines.

You'll still need to complete some details, but the builder will help prevent you making some mistakes.

Command Lines and Floating Licences

Memory Validator works from the GUI and from the command line with all types of software licence (floating licences and non-floating licences).

When floating licences are being used Memory Validator will wait to acquire a floating licence before proceeding to process the command line options.

There are no options to:
- Checkout a floating licence
- Release a floating licence
- Query for available licences

These options are managed automatically by Memory Validator, there is no need for such options to be manually controlled from the command line.

5.1 Manual Regression Testing

Manual testing by comparing sessions

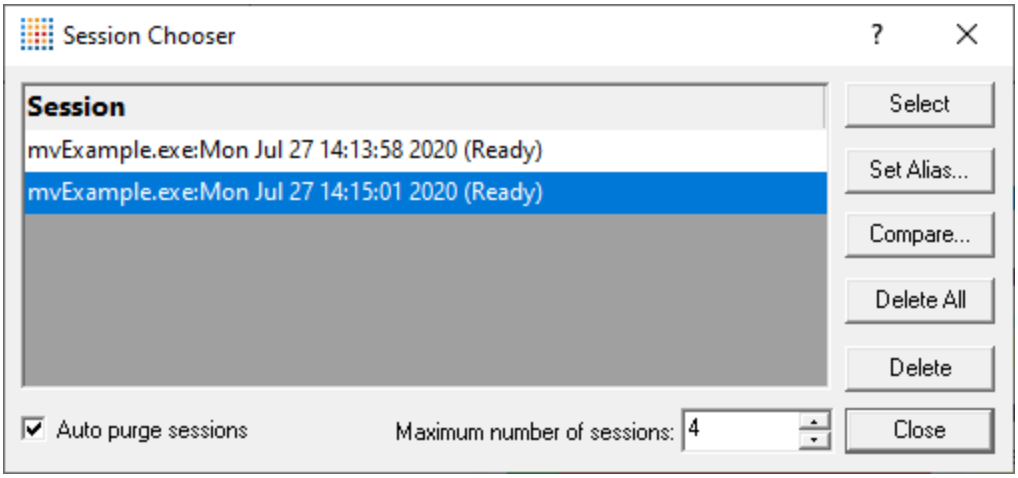
Comparing sessions will identify:

- Regressions since a baseline session
- Improvements from the baseline session to the other
- Leaks common to both sessions

Choosing the sessions to compare

Manual regression testing is possible using the Session Manager to compare two sessions with each other.

Managers menu › **Session Manager...** › shows the Session Chooser dialog below, highlighting the current session

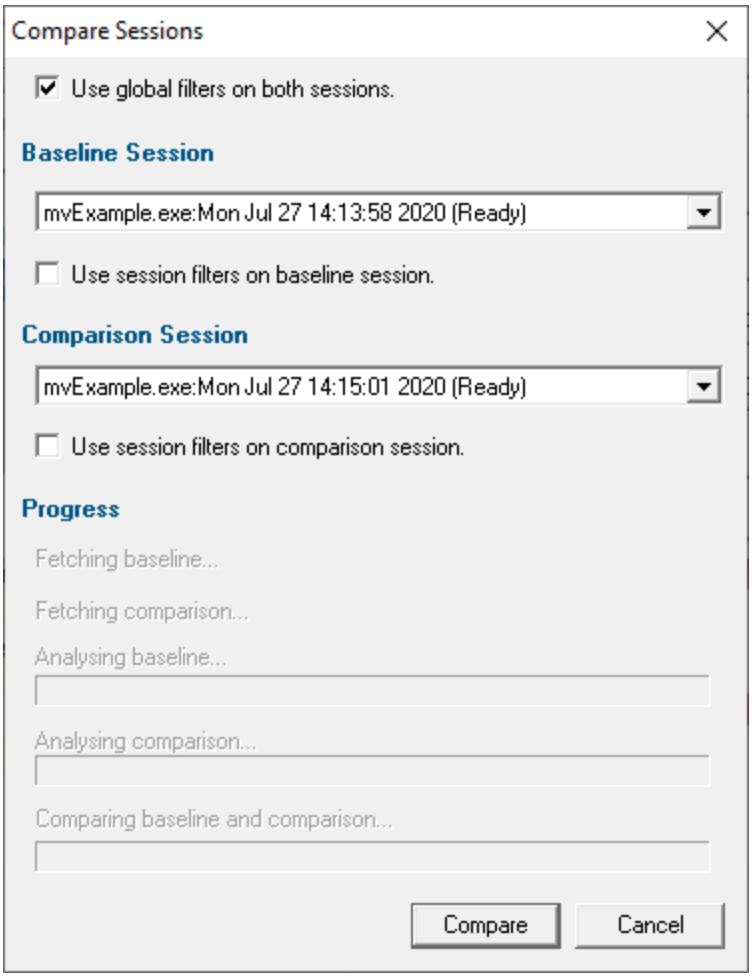


- **Compare...** › shows the Compare Sessions dialog

  Only enabled when at least two different sessions are loaded.

For there to be at least two loaded sessions, you need to switch off *auto purge* or increase the maximum number of sessions so as not to limit the number of open sessions.

Choose two sessions from the drop down lists in the Compare Sessions dialog, one as the baseline session and the other to compare against it.



Check the sessions in the drop down lists are the ones you want to compare.

- **Compare...** > shows progress of analysis and then the Session Memory Comparison dialog

Using filters when comparing sessions

Each session will have some session filters that may or may not be the same.

Checkboxes at the top of the dialog and under each selected session let you choose whether to apply global or session filters during comparison

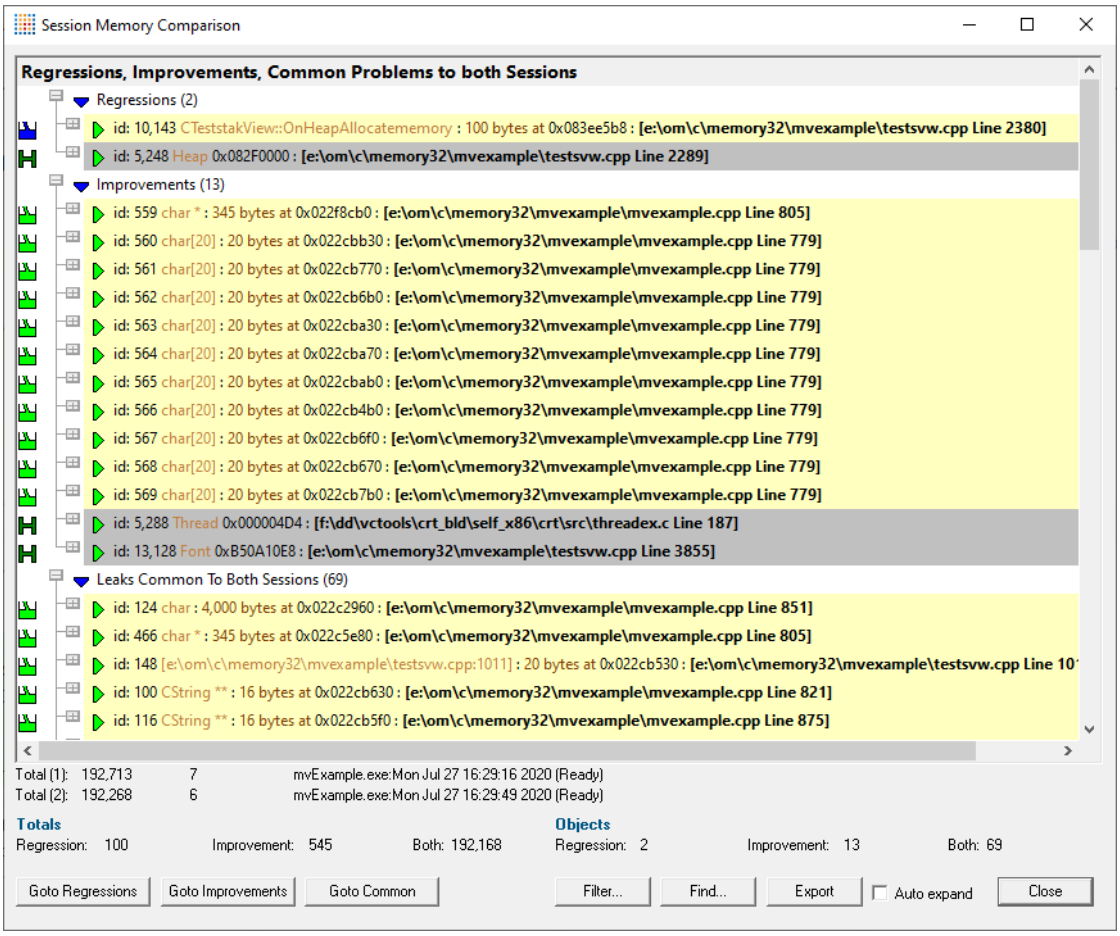
Although you can apply the session filters during the comparison, you need to be sure that the filters are the same, in order for the comparison to be fair.

During comparison, it's generally best to apply global filters rather than session filters.

The session memory comparison dialog

The dialog shows details of memory leaks grouped as follows:

- **Regressions**: leaks that occur in the second session but not in the first (the baseline)
- **Improvements**: leaks that were in the first session but not in the second
- **Common Leaks**: those leaks found in both sessions



Underneath the memory information are some statistics:

- **Total (1) and (2)**: The total amount of memory leaked and number of handles leaked for each session
- **Totals**: The amount of leaked memory attributed to regressions, improvements and common leaks
- **Objects**: The number of leaked objects for each of regressions, improvements and common leaks

The comparison data may be quite long so there are a few convenience buttons to scroll the data to each section

- **Goto Regressions** › scrolls to the start of the regression data
- **Goto Improvements** › scrolls to improvements
- **Goto Common** › scrolls to common data

The data in this comparison dialog is displayed the same as other tab views such as the Memory tab.

You can expand the items to view the allocation callstack and source code snippets.

There's an option to expand lower level data automatically:

- **Auto Expand** › if checked, causes every item's callstack to be expanded automatically when expanding one of the three top level items

Reducing the amount of data displayed

You might well find that you have a lot of data, or you might be looking for regressions of a particular nature.

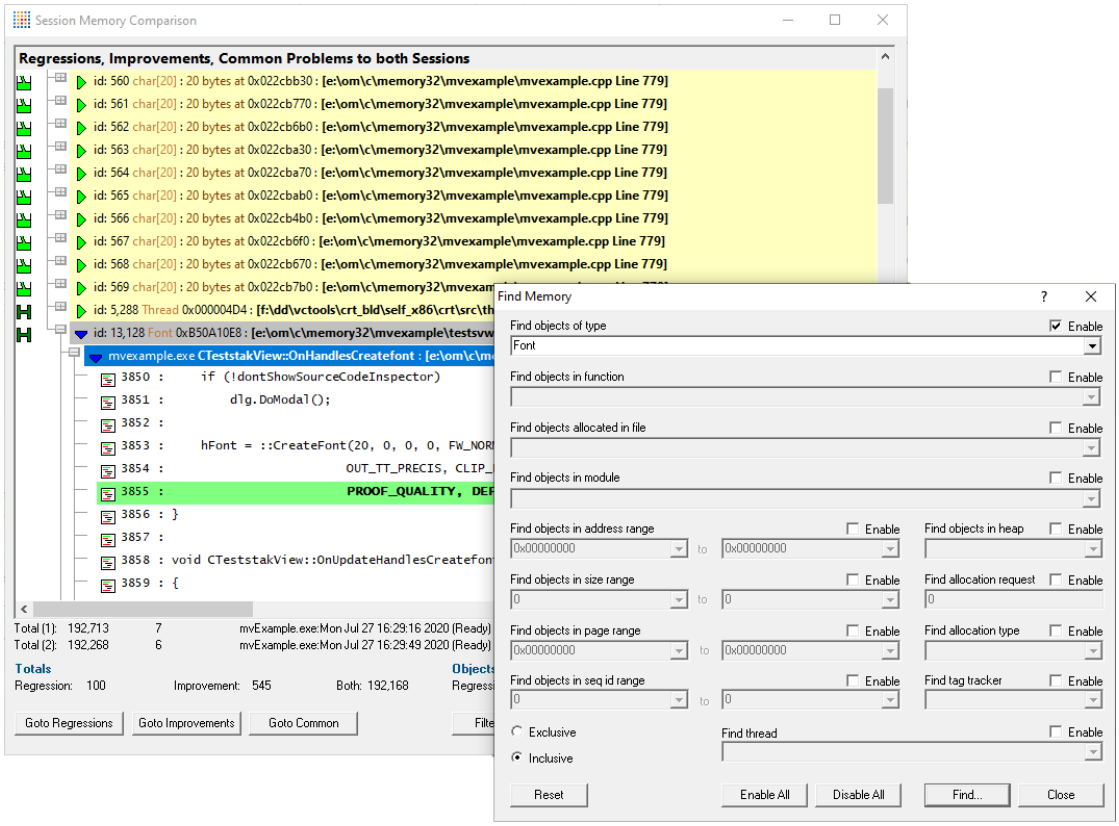
- **Filter...** › shows the session comparison private filters dialog

  The comparison filters are used in an identical way to Local Filters, using the Filter Definition dialog.

- **Find...** › shows the Find Memory dialog to search the comparison data for particular allocation criteria or object types

  The Find Memory dialog is the same as that used when searching for memory in the Memory tab views.

  Items in the comparison dialog that match the search criteria are highlighted gray:



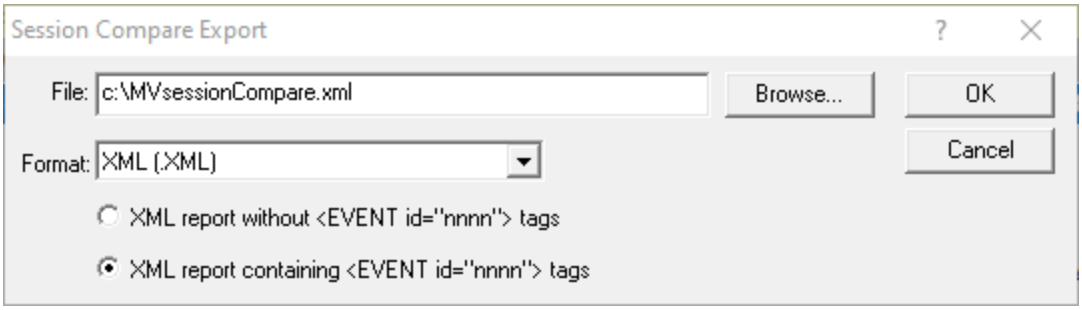
Exporting the session comparison

You might need to export the session comparison data to share it with team members who don't have Memory Validator

- **Export** › shows the Session Compare Export dialog

  Choose the file location and the format: XML or HTML.

  If exporting as XML, you can optionally include the **event id** tags in the data.



5.2 Automated Regression Testing

Automated testing overview

While manual regression testing is possible using the session comparison feature, automated unattended testing is purely command line driven.

Typically, command line options allow Memory Validator to run by specifying:

- the target program to run
- arguments to pass to the target program
- the working directory to run in
- a baseline session to compare with
- where and how to save results
- whether to run with or without the user interface

Usually Memory Validator would exit between automated tests, but it can be made to stay running if necessary.

➡ See the command line reference for an alphabetical listing all the available commands.

To run 32 bit memory validator run `C:\Program Files (x86)\Software Verify\Memory Validator x86\memoryValidator.exe`

To run 64 bit memory validator run `C:\Program Files (x86)\Software Verify\Memory Validator x64\memoryValidator_x64.exe`

Command line argument usage

There are a few basic rules to remember when using the command line arguments:

- separate arguments by spaces
- quote arguments if they contain spaces
- some arguments are only useful in conjunction with others
- some arguments are incompatible with others

If your command line is very long, consider using **-commandFile** to specify a command file for your arguments.

5.2.1 Example Command Lines

Typical command line examples

The following examples demonstrate a few different scenarios in which you might want to use Memory Validator via the command line.

To run 32 bit memory validator run **C:\Program Files (x86)\Software Verify\Memory Validator x86\memoryValidator.exe**

To run 64 bit memory validator run **C:\Program Files (x86)\Software Verify\Memory Validator x64\memoryValidator_x64.exe**

Example 1 - running a session (native or .Net)

A command line to run a program in a directory, using two arguments, and save the session without showing the Memory Validator interface:

memoryValidator_x64.exe **-hideUI** **-program** c:\myProgram.exe **-arg** " -macro c:\macros\testMacro1.vba" **-arg** "secondArg" **-directory** c:\testbed **-saveSession** c:\results\testMacro1.mvm

A brief explanation of each argument:

Option	Argument	Description
-hideUI		Don't show the user interface during the test
-program	c:\myProgram.exe	The target program to launch
-arg	" -macro c:\macros\testMacro1.vba"	An argument to the target program
-arg	"secondArg"	A second argument to the program
-directory	c:\testbed	The current directory for the application to work in
-saveSession	c:\results\testMacro1.mvm	Where to save the session after the application finishes

Add the following to the first example to compare against a baseline session and export the comparison results in HTML and XML format:

-baseline c:\baselines\testMacroBaseline.mvm **-sessionCompareHTML** c:\regression\testMacro1.html **-sessionCompareXML** c:\regression\testMacro1.xml

Option	Argument	Description
-baseline	c:\baselines\testMacroBaseline	The baseline session to compare against (loaded at startup)
-sessionCompareHTML	c:\regression\testMacro1.html	Saved HTML format results of comparison
-sessionCompareXML	c:\regression\testMacro1.xml	Saved XML format results of comparison

To show the ui and leave the Memory tab open to inspect leaks after completion, omit **-hideUI** in the first example and add **-refreshMemory**

Example 2 - running a session (.Net Core, Self Contained)

This example starts a .Net Core application, showing no progress dialog whilst attaching to the process.

On completion, the resulting session is saved, and some tabs are refreshed.

The last tab refreshed is displayed, resulting in the Functions tab being the current tab.

```
memoryValidator_x64.exe -program "c:\myDotNetCoreApp.exe" -dotNetCoreLaunchType
SelfContained -saveSession "c:\myResults\session2.mvm" -displayUI
```

A brief explanation of each argument:

Option	Argument	Description
-program	"c:\myDotNetCoreApp.exe"	The target program to launch
-dotNetCoreLaunchType	SelfContained	The .Net Core program is self contained
-saveSession	"c:\myResults\session2.mvm"	After the application finishes, the session should be saved in this file
-displayUI		Show the user interface during the performance test

Example 3 - running a session (.Net Core, Framework Dependent)

This example starts a .Net Core application, showing no progress dialog whilst attaching to the process.

On completion, the resulting session is saved, and some tabs are refreshed.

The last tab refreshed is displayed, resulting in the Functions tab being the current tab.

```
memoryValidator_x64.exe -program "c:\dotNetCoreApp.dll" -dotNetCoreLaunchType
FrameworkDependent -saveSession "c:\myResults\session3.mvm" -displayUI
```

A brief explanation of each argument:

Option	Argument	Description
-program	"c:\dotNetCoreApp.dll"	The target program to launch with the .Net runtime
-dotNetCoreLaunchType	FrameworkDependent	The .Net Core program is framework dependent

-saveSession	"c:\myResults\session3.mvm"	After the application finishes, the session should be saved in this file
-displayUI		Show the user interface during the performance test

5.2.2 Environment variables

Environment variables can be referenced on the command line.

This allows you to set an environment variable outside of Memory Validator (cmd prompt, batch file, etc) and then reference it on the command line.

For example:

```
-program %BUILD_DIR%\testProgram.exe
```

If the BUILD_DIR environment variable is set to e:\dev\debug the above would evaluate to `-program e:\dev\debug\testProgram.exe`

What if I can't set an environment variable?

There are situations where you it isn't desirable, or possible to set the environment variable value prior to starting Memory Validator.

In those situations you can set the environment variable on the command line using -setenvironment.

```
-setenvironment BUILD_DIR=e:\dev\debug -program %BUILD_DIR%\testProgram.exe
```

Problems with environment variable substitution

If you are running from a command prompt, or batch file, or any process that will handle environment variable substitution using %ENV_VAR% you will find that referencing the environment variable on the command line won't work when using -setenvironment, because by the time Memory Validator sees the command line the %ENV_VAR% values have already been substituted.

To get around this, using ENV_VAR instead of %ENV_VAR%.

```
-setenvironment BUILD_DIR=e:\dev\debug -program $BUILD_DIR$\testProgram.exe
```

-setenvironment

Set environment variables for Memory Validator, as a series of name/value pairs.

Use this option once for each environment variable you wish to set.

Usage of **-setenvironment** for any environment variable must appear on the command line prior to any reference to that environment variable on the command line.

To pass quotes along with the string, escape a pair of inner quotes like the example below

Examples:

-setenvironment APP_FLAG=ON;
-setenvironment "APP_FAG=ON;"
-setenvironment "APP_COMMS=ON; APP_DEBUG=OFF;"
-setenvironment "APP_MSG=\"A quoted string with spaces\";"
-setenvironment BUILD_DIR=e:\dev\debug

Note that this is not the same as -environment, which allows you to specify environment values that you can pass to the program being launched.

5.2.3 Development environment

The following options allow you to specify the development environment you used to build the application being tested.

-devIDE

Specifies the development environment or compiler used to build the application being tested.

These values correspond to the values on the Symbol Lookup part of the settings dialog.

- clang
- codeWarrior
- cppBuilder
- delphi
- devC++
- mingw
- other
- rust
- salfordFortran
- visualBasic6
- visualStudio. If you specify this you also need to specify **-devVisualStudioVersion** or **-devVisualStudioYear**
- visualStudioCustomDLL
- visualStudioDontSet

Examples:

-devIDE visualStudio
-devIDE delphi

-devVisualStudioVersion

If **-devIDE** has been specified as **visualStudio** then **-devVisualStudioVersion** can be used to specify the version of Visual Studio.

The version of Visual Studio needs to be installed on the machine for this to work.

The following versions are valid:

- 6
- 7
- 7.1
- 8
- 9
- 10
- 11
- 12
- 14
- 15
- 16
- 17

Examples:

-devVisualStudioVersion 6
-devVisualStudioVersion 10
-devVisualStudioVersion 17

*If you use **-devVisualStudioVersion** then you don't need to use **-devVisualStudioYear**.*

-devVisualStudioYear

If **-devIDE** has been specified as **visualStudio** then **-devVisualStudioYear** can be used to specify the version of Visual Studio.

The version of Visual Studio needs to be installed on the machine for this to work.

The following years are valid:

- 1996
- 2002
- 2003
- 2005
- 2008
- 2010
- 2012
- 2013
- 2015
- 2017
- 2019
- 2022

Examples:

-devVisualStudioYear 1996
-devVisualStudioYear 2010
-devVisualStudioYear 2022

*If you use **-devVisualStudioYear** then you don't need to use **-devVisualStudioVersion**.*

5.2.4 Target Program & Start Modes

Specifying the target application

The folllowing options let you launch a program (with various startup modes), inject into a running program or wait for a program to start before attaching.

Launching a program

-program

Specifies the full file system path of the executable target program to be started by Memory Validator, including any extension.

Not compatible with -injectName, -injectID, -waitName or -monitorAService.

See -arg below to pass arguments to your program, and -directory to set where it runs.

See -programToMonitor to monitor a different program than the program you launch.

Examples:

-program c:\testbed.exe
-program "c:\new compiler\version2\testbed.exe"

-programToMonitorEXE
-programToMonitor

-programToMonitor has been replaced by **-programToMonitorEXE**. **-programToMonitor** will be honoured to provided backward compatibility.

Specifies the full path of the program from which the data is collected, but *does not change* which process is initially launched. *Include the extension*.

This program will be monitored by Memory Validator only when the program specified using -program starts it.

If no path is specified, the first child process that has the same name will be monitored.

To monitor *any* program that is launched specify `<<Any>>` as the program argument. In batch files and powershell scripts you will need to quote this to get it accepted by the file parser.

See -programToMonitorLaunchCount to change which instance of the program is monitored.

Only valid in conjunction with -program.

Examples:

-programToMonitorEXE c:\testbed-child-process.exe
-programToMonitorEXE "c:\new compiler\version2\testbedChildProcess.exe"

-programToMonitorEXE testbed-child-process.exe
-programToMonitorEXE "<<Any>>"

-program c:\testbed.exe **-programToMonitorEXE** c:\testbed-child-process.exe

In this last example `c:\testbed.exe` is launched but not monitored. Only when `testbed.exe` launches a child process `c:\testbed-child-process.exe` is that child process monitored.

-programToMonitorDLL

This option provides a qualifying DLL to identify different .Net Core processes, which are typically launched using the same .Net Core runtime. *Include the dll extension*.

Only valid in conjunction with -program and -programToMonitorEXE.

Examples:

-programToMonitorDLL c:\test\dotNetCoreApp.dll

-program c:\testbed.exe **-programToMonitorEXE "**c:\program files\dotnet\dotnet.exe" **-programToMonitorDLL** c:\test\dotNetCoreApp.dll

In this last example `c:\testbed.exe` is launched but not monitored. Only when `testbed.exe` launches a child process `c:\program files\dotnet\dotnet.exe` to run the application `c:\test\dotNetCoreApp.dll` is that child process monitored.

-programToMonitorLaunchCount

Specify the n^{th} invocation of the program specified by -programToMonitor which is to have its data collected.

Any value which is invalid (including anything less than 1) will default to 1.

Only valid in conjunction with -programToMonitor and consequently also -program.

Examples:

-programToMonitorLaunchCount 1
-programToMonitorLaunchCount 34

-program c:\testbed.exe **-programToMonitor** c:\testbed-child-process.exe **-programToMonitorLaunchCount 1**

In the above example `c:\testbed.exe` is launched but not monitored. As soon as `testbed.exe` launches a child process `c:\testbed-child-process.exe` then that child process monitored.

If the value 1 was changed to a 2, then only the *second* invocation of `c:\testbed-child-process.exe` would get monitored, with the first invocation being ignored.

-arg

Passes command line arguments to the target program, and can be used multiple times.

 To pass quotes along with the string, escape a pair of inner quotes like the example below

Only valid with: -program

-arg myProgram.exe
-arg "c:\Program Files\myApp\myProgram.exe"
-arg "-in infile -out outfile"
-arg "\"a quoted string\""

-allArgs

Passes the remainder of the command line (after -allArgs) to the program being launched.

Unlike -arg above, there is no need to escape the quotes as the content is passed verbatim.

Only valid with: -program

Example:

-allArgs anything put here is passed to the target program "even stuff in quotes" is passed

-directory

Sets the working directory in which the program is executed. If **-directory** is not specified the program is run in its current directory.

Only valid with: -program

Examples:

-directory c:\development\
-directory "c:\research and development\"

-environment

Environment variables for program, as a series of name/value pairs. Not to be confused with -setenvironment.

Use this option once for each environment variable you wish to set.

 To pass quotes along with the string, escape a pair of inner quotes like the example below

Only valid with: -program

Examples:

-environment APP_FLAG=ON;

-environment "APP_FAG=ON;"
-environment "APP_COMMS=ON; APP_DEBUG=OFF;"
-environment "APP_MSG=\"A quoted string with spaces\";"

-dataCollectType

Specifies the type of data collection that you want. Native, .Net or mixed mode (both native and .Net).

Valid with -program and -monitorAService.

Examples:

-dataCollectType native
-dataCollectType dotNet
-dataCollectType mixedMode

-stdin

Specifies a file to be read and piped to the standard input of the application being tested.

If the filename contains spaces, the filename should be quoted.

An error occurs if the file does not exist. See **-ignoreMissingStdin** to avoid this error.

Examples:

-stdin c:\settings\input.txt
-stdin "c:\reg tests settings\input.txt"

-stdout

Specifies a file to be written with data piped from the standard output of the application being tested.

If the filename contains spaces, the filename should be quoted.

An error occurs if the file does not exist. See **-ignoreMissingStdout** to avoid this error.

Examples:

-stdout c:\settings\output.txt
-stdout "c:\reg tests results\output.txt"

-ignoreMissingStdin

If this flag is specified and the file specified by **-stdin** does not exist, no error is reported.

-ignoreMissingStdout

If this flag is specified and the file specified by **-stdout** does not exist, no error is reported.

Target program startup modes

-createProcessStartupThread
-normalStartupThread
-idleStartupThread
-suspendStartupThread
-pauseStartupThread
-noSuspendInStubDuringAttach

All these options are obsolete and will be ignored if present on command lines or in command files.

Injecting into a program

-injectName

Sets the name of the process for Memory Validator to attach to.

Not compatible with -program, -injectID, -waitName or -monitorAService.

Examples:

-injectName c:\testbed.exe
-injectName "c:\new compiler\version2\testbed.exe"

-injectID

Sets the numeric id of a process for Memory Validator to attach to.

Not compatible with -program, -injectName, -waitName or -monitorAService.

Example:

-injectID 1032

Waiting for a program

-waitNameEXE
-waitName

-waitName has been replaced by **-waitNameEXE**. **-waitName** will be honoured to provided backwards compatibility.

Sets the name of a process that Memory Validator will wait for.

When the named process starts Memory Validator will attach to the process.

Not compatible with -program, -injectName, -injectID or -monitorAService.

Examples:

-waitNameEXE c:\testbed.exe
-waitNameEXE "c:\new compiler\version2\testbed.exe"

-waitNameDLL

Sets the name of a process DLL that Memory Validator will wait for.

When the named process starts Memory Validator will attach to the process.

Examples:

-waitNameDLL c:\dotNetApp.dll
-waitNameDLL "c:\new compiler\version2\dotNetApp.dll"

For use with -waitNameEXE when you want to wait for .Net Core applications.

-waitNameEXE "c:\program files\dotnet\dotnet.exe" **-waitNameDLL** "c:\testApps\dotNetCoreApp\release\dotNetCoreApp.dll"

Monitoring a service

-monitorAService

Sets the full file system path of a service *including any extension*.

Memory Validator will wait for the service to start and attach to it.

Not compatible with -program, -injectName, -injectID or -waitName.

Examples:

-monitorAService c:\service.exe
-monitorAService "c:\new compiler\version2\service.exe"

.Net Core specific arguments

-dotNetCoreArg

Specifies and argument to pass to the .Net Core runtime. You can specify -dotNetCoreArg as many times as you need to pass as many arguments as you need.

See this Microsoft document for the list of .Net Core runtime configuration options https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet#runtime-options.

Use this argument with -program.

Examples:

-dotNetCoreArg "--roll-forward LatestPatch"
-dotNetCoreArg "--runtimeconfig ./configUnitTest.json"

-dotNetCoreLaunchType

Specifies the type of program being launched by the .Net Core runtime. You can specify -dotNetCoreLaunchType once. If specified more than once, the last definition is used.

Use this argument with -program.

Examples:

-dotNetCoreLaunchType SelfContained
-dotNetCoreLaunchType FrameworkDependent

Data Collection

-collectData

Enables or disables the collection of flow tracing data

Examples:

-collectData:On
-collectData:Off

-collectStdout

Enables or disables the collection of standard output (stdout)

Examples:

-collectStdout:On
-collectStdout:Off

5.2.5 User interface visibility

User interface visibility

You can choose to hide or show Memory Validator during the test, as well as the window of the target application.

-displayUI

Forces the Memory Validator user interface to be displayed during the test.

This is useful for debugging a command line session that is not working, for example inspecting the Diagnostic tab for messages related to the test.

You wouldn't normally use this option when running unattended regression tests.

-doNotInteractWithUser

This causes Memory Validator to never display dialog boxes in the target application that is being profiled.

 This applies even for warning and error dialog boxes.

The intended use for this option is for when you are running command line sessions on unattended computers and you have automated processes that may kill the Memory Validator user interface if something goes wrong. Actions such as this then cause the stub to recognise the user interface has gone away and display an error warning.

-hideUI

Hides the Memory Validator user interface during the test.

-launchAppHidden

Hides the target application during the test.

 Depending on your application, this may not work and may not even be suitable.

This is equivalent to setting the `wShowWindow` member of the `STARTUPINFO` struct to `SW_HIDE` when using the Win32 `CreateProcess()` function.

It's useful if you're testing console applications that have no user interaction, as it prevents the console/command prompt from being displayed.

For GUI applications this option very much depends on how your application works.

For interactive applications, it's clearly has no use, but for some, hiding the GUI may help prevent various windows messages from being processed.

Typically, for complex applications, it's better to design this capability into your application and control it via a command line, which can be passed in from Memory Validator via the **-arg** option.

5.2.6 Session Management

Session management

The following options let you control the sessions during testing

-baseline

Loads a previous session as the baseline session against which the recorded session is compared for regressions or improvements.

Examples:

-baseline c:\baseline\testMacro1.mvm
-baseline "c:\base line\testMacro1.mvm"

 Ensure your session manager is configured to hold at least 2 sessions or use **-numSessions** to specify how many sessions to use.

-numSessions

Sets the number of sessions that can be loaded at once.

This is equivalent to the same setting in the session manage and can't be less than 1.

Example:

-numSessions 2

-saveSession

Saves the session data when all data has finished being collecting from the target program.

Examples:

-saveSession c:\results\testMacro1.mvm
-saveSession "c:\test results\testMacro1.mvm"

-sessionLoad

Loads a previously created session to act as the comparison session and which will be compared against a baseline session, for regression testing.

This might be used when a series of tests have been performed and the sessions saved.

The regression test results can then be generated by using **-baseline** and **-sessionLoad** to load two sessions and compare them.

Use in conjunction with: **-baseline,**

Not compatible with: **-program**, **-injectName**, **-injectID**, **-waitName**, as regression tests will be performed using the wrong session data (the session from **-sessionLoad** rather than the session from **-baseline**).

Examples:

-sessionLoad c:\results\testMacro1.mvm
-sessionLoad "c:\test results\testMacro1.mvm"

Ensure your session manager is configured to hold at least 2 sessions or use **-numSessions** to specify how many sessions to use.

-sessionCompareHTML
-sessionCompareXML

Compare two sessions, producing an HTML or XML report detailing any regression and improvements.

A description of the XML tags used to produce the report is here.

The two sessions can be loaded using one of these options:

-baseline and **-sessionLoad**

-baseline and running a program using one of **-program**, **-injectName**, **-injectID**, or **-waitName**

Examples:

-sessionCompareXML c:\regtests\testMacro1.xml
-sessionCompareHTML "c:\reg tests\testMacro1.html"

Ensure your session manager is configured to hold at least 2 sessions or use **-numSessions** to specify how many sessions to use.

-flatXMLSessionCompareExport

Use the old style XML report without the <EVENT> tag when exporting the results of a session comparison.

A similar option **-flatXMLSessionExport** is available for exporting the session data.

See also the description of the XML export tags.

5.2.7 Session Export Options

Session export format - HTML or XML

-exportAsHTML
-exportAsXML

Export the session data as an HTML or XML file when Memory Validator has finished collecting data from the target program.

Example:

-exportAsHTML c:\results\html\testMacro1.html

-exportAsXML "c:\test results\xml\testMacro1.html"

Example fragment of a detailed HTML export

```
<TR BGCOLOR="#008080"><TD>Thread:0x000000d0</TD>
<TD>Time:7/14 13:32:03 651ms</TD>
<TD>File:E:\OM\C\memory32\examples\nativeExample\nativeExample.CPP</TD>
<TD>Line:161</TD>
<TD>Address:0x00375668</TD>
<TD>Size:0x00000004</TD>
<TD>Type:Allocation</TD>
<TD>Leaked:TRUE</TD>
<TD>Uninitialised:FALSE</TD>
<TD>Damaged:FALSE</TD>
<TD>Unused:FALSE</TD>
<TD>SizeError:FALSE</TD>
<TD>IncorrectUsage:FALSE</TD>
<TD>Type:Unknown </TD>
<TD>Allocation ID:157</TD>
</TR>
 <TR><TD COLSPAN=10><TABLE>
  <TR><TD>0x00401905 nativeExample.exe CTeststakApp::CTeststakApp : [E:\OM\C\mem
  <TR><TD>0x00401e58 nativeExample.exe $E320 : [E:\OM\C\memory32\examples\native
  <TR><TD>0x00401e33 nativeExample.exe $E323<BR></TD></TR>
  <TR><TD>0x1020ad33 MSVCRTD.dll _initterm : [crt0dat.c Line 524]<BR></TD></TR>
  <TR><TD>0x00412bfb nativeExample.exe wWinMainCRTStartup : [crtexe.c Line 274]<
  <TR><TD>0x7c816d4a KERNEL32.dll RegisterWaitForInputIdle<BR></TD></TR>
 </TABLE></TD>
</TR>
```

See also the section on XML Export Tags.

Session export encoding - HTML or XML

These options allow you to export the session data as UTF-16, UTF8 or ASCII. UTF-16 and UTF-8 will add a byte order mark (BOM) to the start of the exported file.

-exportAsHTML_BOM

The exported HTML will be exported with the appropriate format.

-exportAsHTML_BOM ASCII
-exportAsHTML_BOM UTF8
-exportAsHTML_BOM UTF16

-exportAsXML_BOM

The exported XML will be exported with the appropriate format.

-exportAsXML_BOM ASCII
-exportAsXML_BOM UTF8
-exportAsXML_BOM UTF16

HTML and XML export options

The following options control settings for HTML and XML export of the recorded session results

All the export settings take the form option:On or Option:Off

Example:

-htmlAllocMemory:On
-xmlReAllocMemory:Off

All these options are equivalent to the settings used in the Export Session dialog.

-htmlAllocMemory **-xmlAllocMemory**	Include allocated memory data in the export
-htmlReAllocMemory **-xmlReAllocMemory**	Include reallocated memory data in the export
-htmlFreeMemory **-xmlFreeMemory**	Include free memory data in the export
-htmlLeaksOnly **-xmlLeaksOnly**	Only include information about *leaked* data
-htmlMemory **-xmlMemory**	Include data describing memory allocations
-htmlMemoryStackTrace **-xmlMemoryStackTrace**	Include stack traces for memory allocations
-htmlMemoryFilter **-xmlMemoryFilter**	Apply filters (session or global) to data memory allocations
-htmlHandles **-xmlHandles**	Include data describing handle allocations
-htmlHandleStackTrace **-xmlHandleStackTrace**	Include stack traces for handle allocations
-htmlHandleFilter **-xmlHandleFilter**	Apply filters (session or global) to data for handle allocations
-htmlErrors **-xmlErrors**	Include data describing error conditions
-htmlErrorStackTrace **-xmlErrorStackTrace**	Include stack traces for error conditions
-htmlErrorFilter **-xmlErrorFilter**	Apply filters (session or global) to data for error conditions
-htmlTrace **-xmlTrace**	Include data describing TRACE and OutputDebugString information

-htmlTraceStackTrace **-xmlTraceStackTrace**	Include stack traces for trace information
-htmlTraceFilter **-xmlTraceFilter**	Apply filters (session or global) to data for trace information
-htmlDetailedReport **-xmlDetailedReport**	Include Thread ID and Timestamp information as additional data
-htmlDoColourCode	Make the HTML export colour coded (not relevant for XML export)
-flatXMLSessionExport	Use the old style (flat) XML report *without* the <EVENT> tag. See XML Export Tags.

Exporting objects

Exporting data about objects doesn't correspond to any corresponding setting in the user interface.

When running automated regression tests in unattended mode, you won't be able to manually inspect the Types tab, so exporting this data can be very useful.

You can choose to export all the Objects data or just the data for leaked objects.

-exportSessionObjectsHTML
-exportSessionObjectsXML

Exports the data on the Types tab (sub tab "DLL", for all threads) as an HTML or XML file when Memory Validator has finished collecting data from the target program.

Example:

-exportSessionObjectsHTML c:\results\html\testMacro1Objects.html
-exportSessionObjectsHTML "c:\test results\xml\testMacro1Objects.html"

-exportSessionLeakedObjectsHTML
-exportSessionLeakedObjectsXML

As above, but only data for objects with a non-zero Count value is exported.

Since the Count column indicates *live* allocations a non-zero count indicates leaked objects once the target program has exited.

➡ See also -filterSessionLeakedObjectsHTML / XML for applying global filters to exported session objects.

⊟ Example of an Objects XML export

```
<XML>
 <VALIDATORVERSION>1.0</VALIDATORVERSION>
 <VALIDATORDATE>Apr 14 2014</VALIDATORDATE>
 <VALIDATORTIME>11:48:30</VALIDATORTIME>
 <TITLE>nativeExample.exe [All Objects]</TITLE>
  <OBJECTS>
   <OBJECT>
    <NAME>CtestParsing_c</NAME>
    <NUMBER>3</NUMBER>
    <MAX>3</MAX>
    <TOTAL>3</TOTAL>
    <SIZE>4</SIZE>
    <CUMUMLATIVE_SIZE>12</CUMUMLATIVE_SIZE>
   </OBJECT>
  <OBJECT>
   <NAME>char*</NAME>
   <NUMBER>0</NUMBER>
   <MAX>2</MAX>
   <TOTAL>2</TOTAL>
   <AVERAGE_SIZE>672.50</AVERAGE_SIZE>
   <CUMUMLATIVE_SIZE>1345</CUMUMLATIVE_SIZE>
  </OBJECT>
 </OBJECTS>
</XML>
```

5.2.8 Filter options

Filter options

Filters can be used during session comparison to exclude data from the test results.

Filters are usually stored in the default filter file.

To save your own set of filters for use in a regression test, you can save the filters to a file from the Filter Manager dialog.

-sessionFilters
-globalFilters

Specify filters to be applied to the sessions loaded for the regression test.

For most use cases you'll want to use **-globalFilters** so that the same filters are applied to the baseline session *and* the recorded session.

Examples:

-sessionFilters c:\filters\testMacro1.mvf
-globalFilters "c:\reg tests filters\testMacro1.mvf"

-compareUseBaselineFilters

Apply the filters loaded using **-sessionFilters** to the baseline session.

-compareUseGlobalFilters

Apply the filters loaded using **-globalFilters** to the baseline session and the comparison session.

-filterSessionLeakedObjectsHTML
-filterSessionLeakedObjectsXML

When the session objects are exported to an HTML or XML file, the output is filtered using the filters loaded using **-globalFilters**.

5.2.9 File Locations

File Locations

When using the command line it's convenient to store settings and options in files that can be easily referenced.

Those files include:

- Global settings files
- File locations for source, third party source, PDB or MAP files
- DLL hook files

Each of these file types can be saved or exported from Memory Validator.

Loading global settings from a file

Global settings are usually stored in the registry, but you can save a specific set of settings for use in regression tests:

- **Settings** menu › **Save settings...**

-settings

Points to a previously saved settings file to be used for the test.

Examples:

-settings c:\settings\testMacro1.mvs
-settings "c:\reg tests settings\testMacro1.mvs"

File locations for source, PDB or MAP files

File location files can be easily generated by exporting file locations from the File Locations page of the settings dialog.

-fileLocations

Specify a plain text file listing file locations to be used during testing. See the format of the file below.

Each set of file types (one per line) is preceded by a header line in the file.

- **[Files]** › Source files
- **[Third]** › Third party source files
- **[PDB]** › PDB files
- **[MAP]** › MAP files

Example:

-fileLocations c:\regressionsTests\testFileLocations1.mvxfl

Example file:

```
[Files]
c:\work\project1\
[Third]
d:\VisualStudio\VC98\Include
[PDB]
c:\work\project3\debug
c:\work\project3\release
[MAP]
c:\work\project3\debug
c:\work\project3\release
```

Files listing DLLs to hook

DLL hook files can be easily generated by exporting DLL hooks from the Hooked DLLs page in the Filters section of the settings dialog.

-dllHookFile

Points to a file listing the DLLs to be hooked for the test.

Examples:

-dllHookFile c:\settings\testMacroDLLs.mvx
-dllHookFile "c:\reg tests settings\testMacroDLLs.mvx"

The first line of text in the DLL hooks file is one of the following:

- Rule: DoNot Hook › DLLs marked as enabled *will not* be hooked. All other DLLs will be hooked
- Rule: DoHook › DLLs marked as enabled *will* be hooked. All other DLLs will not be hooked

- Rule: HookAll > All DLLs will be hooked regardless of the settings in the list

Capitalization is important.

The remaining lines list one DLL filename or folder path and an enabled state on each line.

Example:

```
Rule:DoNotHook
nativeExample.exe enable=FALSE
MFC42D.DLL enable=TRUE
MSVCRTD.dll enable=TRUE
KERNEL32.dll enable=TRUE
ole32.dll enable=TRUE
```

Example:

```
Rule:DoHook
E:\OM\C\memoryValidator\examples\nativeExample\DebugNonLink
enable=TRUE
```

Example:

```
Rule:DoHook
E:\OM\C\memoryValidator\examples\nativeExample with
spaces\DebugNonLink enable=TRUE
```

Example:

```
Rule:DoHook
%ENV_VAR%\DebugNonLink enable=TRUE
```

Here, the environment variable `ENV_VAR` is used to replace the text `%ENV_VAR%` in the path definition.

The file can be ANSI or UNICODE text and paths with spaces do not need quotes.

5.2.10 Command Files

Using a command file

If your command line is very long, consider using **-commandFile** to specify a command file for your arguments.

-commandFile

Specify a file from which to read the command line arguments.

Useful when command lines become unwieldy or longer than the windows command shell limits.

Use -- to insert comments into the file, including when commenting out option.

Examples:

-commandFile c:\regtests\testMacro1.cf
-commandFile "c:\reg tests\testMacro1.cf"

Example command file

```
-hideUI
-program c:\testbed\testApp.exe

-- arguments for application
-arg argumentOne
-arg argumentTwo
-arg "-s wobble"
-directory c:\testbed\test1
-settings c:\testbed\settings_test1.mvs

-- do export and save of the results
-exportAsHTML c:\testbed\results\test1.html
-saveSession c:\testbed\results\test1.mvm
```

For any argument that can be supplied to a command in a command file, you can also specify an environment variable substitution.

```
-directory %DIR%
-program %DIR%\testProgram.exe
```

The environment variables must have been set prior to starting Memory Validator.

You cannot specify a command with an environment variable substitution.

5.2.11 Help, Errors & Return Codes

The following options may help with using and debugging the command line driven automated regression testing.

Command line help

-help
-?

Command line help is printed on the standard output.

Debugging command driven testing

If you're having problems with using the command line, check the following, try displaying error messages using the option below, and look at the exit return codes.

.

- separate command line arguments with spaces

- all command line options that include spaces need to have quotes around them
- some arguments are only useful in conjunction with others - check notes against each option
- some arguments are incompatible with others - check notes against each option

-showErrorsWithMessageBox

Forces errors to be displayed using a message box when running from the command line.

This can be very useful when debugging a command line that does not appear to work correctly.

Exit return codes

Memory Validator returns the following status codes when running from the command line.

- **0** › All ok
- **-1** › Unknown error. An unexpected error occurred starting the runtime
- **-2** › Application started ok. You should not see this code returned
- **-3** › Application failed to start. E.g. runtime not present, not an executable or injection dll not present,
- **-4** › Target application is not an application
- **-5** › Don't know what format the executable is, cannot process it
- **-6** › Not a 32 bit application
- **-7** › Not a 64 bit application
- **-8** › Using incorrect MSVCR(8|9).DLL that links to CoreDLL.dll (incorrect DLL is from WinCE)
- **-9** › Win16 app cannot start these because we can't inject into them
- **-10** › Win32 app - not used
- **-11** › Win64 app - not used
- **-12** › .Net application
- **-13** › User bailed out because app not linked to MSVCRT dynamically
- **-14** › Not found in launch history
- **-15** › DLL to inject was not found
- **-16** › Startup directory does not exist
- **-17** › Symbol server directory does not exist
- **-18** › Could not build a command line
- **-19** › No runtime specified, cannot execute script (or Java) (obsolete)
- **-20** › Java arguments are OK - not an error (obsolete)
- **-21** › Java agentlib supplied that is not allowed because Java Memory Validator uses it (obsolete)
- **-22** › Java xrun supplied that is not allowed because Java Memory Validator uses it (obsolete)
- **-23** › Java cp supplied that is not allowed because Java Memory Validator uses it (obsolete)
- **-24** › Java classpath supplied that is not allowed because Java Memory Validator uses it (obsolete)
- **-25** › Firefox is already running, please close it (obsolete)
- **-26** › Lua runtime DLL version is not known (obsolete)
- **-27** › Not compatible software
- **-28** › InjectUsingCreateProcess, no DLL name supplied
- **-29** › InjectUsingCreateProcess, Unable to open PE File when inspecting DLL
- **-30** › InjectUsingCreateProcess, Invalid PE File when inspecting DLL
- **-31** › InjectUsingCreateProcess, No Kernel32 DLL
- **-32** › InjectUsingCreateProcess, NULL VirtualFree() from GetProcAddress
- **-33** › InjectUsingCreateProcess, NULL GetModuleHandleW() from GetModuleHandleW
- **-34** › InjectUsingCreateProcess, NULL LoadLibraryW() from LoadLibraryW

- **-35** › InjectUsingCreateProcess, NULL FreeLibrary() from FreeLibrary
- **-36** › InjectUsingCreateProcess, NULL VirtualProtect() from GetProcAddress
- **-37** › InjectUsingCreateProcess, NULL VirtualFree() from GetProcAddress
- **-38** › InjectUsingCreateProcess, unable to find DLL load address
- **-39** › InjectUsingCreateProcess, unable to write to remote process's memory
- **-40** › InjectUsingCreateProcess, unable to read remote process's memory
- **-41** › InjectUsingCreateProcess, unable to resume a thread
- **-42** › UPX compressed - cannot process such executables
- **-43** › Java class not found in CLASSPATH
- **-44** › Failed to launch the 32 bit svlGetProcAddressHelperUtil.exe
- **-45** › Uknown error with svlGetProcAddressHelperUtil.exe
- **-46** › Couldn't load specified DLL into svlGetProcAddressHelperUtil.exe
- **-47** › Couldn't find function in the DLL svlGetProcAddressHelperUtil.exe
- **-48** › Missing DLL argument svlGetProcAddressHelperUtil.exe
- **-49** › Missing function argument svlGetProcAddressHelperUtil.exe
- **-50** › Missing svlGetProcAddressHelperUtil.exe
- **-51** › Target process has a manifest that requires elevation
- **-52** › svlInjectIntoProcessHelper_x64.exe not found
- **-53** › svlInjectIntoProcessHelper_x64.exe failed to start
- **-54** › svlInjectIntoProcessHelper_x64.exe failed to return error code
- **-55** › getImageBase() worked ok
- **-56** › ReadFile() failed in getImageBase()
- **-57** › NULL pointer when trying to allocate memory
- **-58** › CreateFile() failed in getImageBase()
- **-59** › ReadProcessMemory() failed in getImageBase()
- **-60** › VirtualQueryEx() failed in getImageBase()
- **-61** › Bad /appName argument in svlInjectIntoProcessHelper_x64.exe
- **-62** › Bad /dllName argument in svlInjectIntoProcessHelper_x64.exe
- **-63** › Bad /procId argument in svlInjectIntoProcessHelper_x64.exe
- **-64** › Failed to OpenProcess in svlInjectIntoProcessHelper_x64.exe
- **-65** › A DLL that the .exe depends upon cannot be found
- **-66** › A stdin file was specified, but Validator could not open it
- **-67** › A stdout file was specified, but Validator could not open it
- **-68** › Failed to create the child output pipe
- **-69** › Failed to create a duplicate of the output write handle for the std error write handle. This is necessary in case the child application closes one of its std output handles
- **-70** › Failed to create the child input pipe
- **-71** › Failed to create a duplicate output read temporary file
- **-72** › Failed to create a duplicate input write temporary file
- **-73** › User was trying to launch a service as an application that was linked to MV APIs. User cancelled when informed of this fact
- **-74** › Returned by Memory Validator if user performs a baseline comparison and memory leaks are detected
- **-75** › Shutdown and restart as 32 bit Memory Validator
- **-76** › Shutdown and restart as 64 bit Memory Validator
- **-77** › Entry point in executable is NULL.
- **-78** › Application is .Net Core.
- **-79** › Entry point is for a .Net application.
- **-80** › VirtualAllocEx() returned NULL
- **-81** › InjectUsingCreateProcess, NULL GetLastError() from GetProcAddress

Updating data after test completion

You can run automated tests that leave the user interface open after completion,

The following options are used to automatically refresh the main data tabs in Memory Validator once a test is complete.

-refreshMemory
-refreshObjects
-refreshHotspot
-refreshAnalysis
-refreshPages

5.2.12 Command Line Reference

Command line reference

The following alphabetical list provides a convenient look-up for all the command line arguments used in automated regression testing.

Option	Description
-?	Print command line help on the standard output.
-allArgs	Pass the remainder of the command line (after -allArgs) to the program being launched.
-arg	Pass command line arguments to the target program. Can be used multiple times.
-baseline	Load a previous session as the baseline session against which the recorded session is compared for regressions or improvements.
-collectData	Turn data collection on or off
-collectStdout	Turn collection of stdout on or off
-commandFile	Specify a file from which to read the command line arguments.
-compareUseBaselineFilters	Apply the filters loaded using -sessionFilters to the baseline session.
-compareUseGlobalFilters	Apply the filters loaded using -globalFilters to the baseline session and the comparison session.
-createProcessStartupThread	Obsolete and ignored if present.
-devIDE	Specify the development environment used to be the target program.
-devVisualStudioVersion	Specify which version of Visual Studio by year.
-devVisualStudioYear	Specify which version of Visual Studio by version number.
-directory	Set the working directory in which the program is executed.

-displayUI	Force the Memory Validator user interface to be displayed during the test.
-dllHookFile	Point to a file listing the DLLs to be hooked for the test.
-doNotInteractWithUser	Never display dialog boxes (including error boxes) in the target application that is being profiled
-dotNetCoreArg	Specify a runtime configuration option to the .Net runtime.
-dotNetCoreLaunchType	Specify if you are launching a self contained or framework dependent .Net Core application.
-environment	Environment variables for program, as a series of name/value pairs
-exportAsHTML -exportAsXML	Export the session data as an HTML or XML file when Memory Validator has finished collecting data from the target program.
-exportAsHTML_BOM -exportAsXML_BOM	Specify the file encoding for the exported file
-exportSessionLeakedObjectsHTML -exportSessionLeakedObjectsXML	Export the data on the Types tab as an HTML or XML file when Memory Validator has finished collecting data from the target program. Only data for objects with a non-zero Count value is exported.
-exportSessionObjectsHTML -exportSessionObjectsXML	Exports the data on the Types tab as an HTML or XML file when Memory Validator has finished collecting data from the target program.
-fileLocations	Specify a plain text file listing file locations to be used during testing.
-exportSessionObjectsHTML -exportSessionObjectsXML	When the session objects are exported to an HTML or XML file, output is filtered using the filters loaded using -globalFilters.
-flatXMLSessionCompareExport	Use the old style XML report without the <EVENT> tag when exporting the results of a session comparison.
-flatXMLSessionExport	Use the old style (flat) XML report *without* the <EVENT> tag.
-globalFilters	Specify filters to be applied to the sessions loaded for the regression test.
-help	Print command line help on the standard output.
-hideUI	Hide the Memory Validator user interface during the test.
-htmlAllocMemory	Include allocated memory data in the export.
-htmlDetailedReport	Include Thread ID and Timestamp information as additional data.
-htmlDoColourCode	Make the HTML export colour coded.
-htmlErrorFilter	Apply filters (session or global) to data for error conditions.
-htmlErrors	Include data describing error conditions.
-htmlErrorStackTrace	Include stack traces for error conditions.
-htmlFreeMemory	Include free memory data in the export

-htmlHandleFilter	Apply filters (session or global) to data for handle allocations.
-htmlHandles	Include data describing handle allocations.
-htmlHandleStackTrace	Include stack traces for handle allocations.
-htmlLeaksOnly	Only include information about *leaked* data.
-htmlMemory	Include data describing memory allocations.
-htmlMemoryFilter	Apply filters (session or global) to data memory allocations.
-htmlMemoryStackTrace	Include stack traces for memory allocations.
-htmlReAllocMemory	Include reallocated memory data in the export.
-htmlTrace	Include data describing TRACE and OutputDebugString information.
-htmlTraceFilter	Apply filters (session or global) to data for trace information.
-htmlTraceStackTrace	Include stack traces for trace information.
-idleStartupThread	Obsolete and ignored if present.
-injectID	Set the numeric id of a process for Memory Validator to attach to.
-injectName	Set the name of the process for Memory Validator to attach to.
-launchAppHidden	Hide the target application during the test.
-monitorAService	Specify the full file system path to the service to monitor with Memory Validator, including any extension. The service is not started by Memory Validator but my an external means.
-normalStartupThread	Obsolete and ignored if present.
-noSuspendInStubDuringAttach	Obsolete and ignored if present.
-numSessions	Set the number of sessions that can be loaded at once.
-pauseStartupThread	Obsolete and ignored if present.
-program	Specify the full file system path of the executable target program to be started by Memory Validator, including any extension.
-programToMonitorDLL	Specify the .Net Core DLL that identifies the program being monitored. Use in conjunction with -programToMonitorEXE.
-programToMonitorEXE -programToMonitor	Changes which program the data is collected from but does not change which process Memory Validator initially launches.
-programToMonitorLaunchCount	Specify the nth invocation of the programToMonitor which is to have its data collected.
-refreshAnalysis	Automatically refresh the Analysis tab in Memory Validator once a test is complete.
-refreshHotspot	Automatically refresh the Hotspots tab in Memory Validator once a test is complete.

-refreshMemory	Automatically refresh the Memory tab in Memory Validator once a test is complete.
-refreshObjects	Automatically refresh the Types tab in Memory Validator once a test is complete
-refreshPages	Automatically refresh the Pages tab in Memory Validator once a test is complete.
-saveSession	Save the session data when all data has finished being collecting from the target program.
-sessionCompareHTML -sessionCompareXML	Compare two sessions, producing an HTML or XML report detailing any regression and improvements.
-sessionFilters	Specify filters to be applied to the sessions loaded for the regression test.
-sessionLoad	Load a previously created session to act as the comparison session and which will be compared against a baseline session, for regression testing.
-showErrorsWithMessageBox	Force errors to be displayed using a message box when running from the command line.
-setenvironment	Environment variables for Memory Validator, as a series of name/value pairs
-settings	Point to a previously saved settings file to be used for the test.
-suspendStartupThread	Obsolete and ignored if present.
-waitNameDLL	Name a .Net Core dll that identifies the process to wait for. Use in conjunction with -waitNameEXE.
-waitNameEXE -waitName	Set the name of a process that Memory Validator will wait for.
-xmlAllocMemory	Include allocated memory data in the export.
-xmlDetailedReport	Include Thread ID and Timestamp information as additional data.
-xmlErrorFilter	Apply filters (session or global) to data for error conditions.
-xmlErrors	Include data describing error conditions.
-xmlErrorStackTrace	Include stack traces for error conditions.
-xmlFreeMemory	Include free memory data in the export.
-xmlHandleFilter	Apply filters (session or global) to data for handle allocations.
-xmlHandles	Include data describing handle allocations.
-xmlHandleStackTrace	Include stack traces for handle allocations.
-xmlLeaksOnly	Only include information about *leaked* data.
-xmlMemory	Include data describing memory allocations.
-xmlMemoryFilter	Apply filters (session or global) to data memory allocations.
-xmlMemoryStackTrace	Include stack traces for memory allocations.
-xmlReAllocMemory	Include reallocated memory data in the export.

-xmlTrace	Include data describing TRACE and OutputDebugString informationApply.
-xmlTraceFilter	Apply filters (session or global) to data for trace information.
-xmlTraceStackTrace	Include stack traces for trace information.

To run 32 bit memory validator run `C:\Program Files (x86)\Software Verify\Memory Validator x86\memoryValidator.exe`

To run 64 bit memory validator run `C:\Program Files (x86)\Software Verify\Memory Validator x64\memoryValidator_x64.exe`

5.2.13 Troubleshooting

Running from the command line can cause some problems, often because you can't be sure that what you put on the command line did what you thought would do.

Ensure the arguments supplied are what you expected.

-echoArgsToUser

If you are testing a console application, make sure you can see it.

-showCommandPrompt

If an errors occur when processing the command line, make sure you can see those.

-showErrorsOnCommandPrompt

-showErrorsWithMessageBox

Look on the diagnostic tab to ensure the diagnostic data collected makes sense.

If you've got -hideUI in your command line, comment it out temporarily (make it -xhideUI so that it's not recognised).

What if the tool hangs?

If you're running from the command line, most likely you'll be running from a cmd prompt, or possibly powershell.

We've only ever had one customer report a hang with any of our tools when running from the command line.

We eventually found the problem, and it wasn't with the software tool.

The problem was that they were running the tool in hidden mode (-hideUI) from a command prompt and for unknown reasons the tool would never exit.

When they added a simple change to their command the problem went away.

They added **cmd /c** to the start of their command line. This opens a new command prompt and instructs it to launch the command line and wait for it to exit.

Problem command line:

```
"c:\program files (x86)\Software Verify\Memory Validator x64\memoryValidator_x64.exe" -program c:\
```

Working command line:

```
cmd /c "c:\program files (x86)\Software Verify\Memory Validator x64\memoryValidator_x64.exe" -prog
```

Part VI

6 Native API

The Memory Validator API

There are some features of Memory Validator that are useful to call directly from your program, including tracking of memory in custom heap managers.

Memory Validator has an API that makes this possible; just include `svlMPAPI.c and svlMPAPI.h` to your codebase. There is no library to link to, dlls to copy.

Source files

The source files can be found in the `API` directory in the Memory Validator install directory.

```
svlMVAPI.h
svlMVAPI.c
```

Just add these files to your project and build.

If you are using precompiled headers you will need to disable them for `svlMVAPI.c`.

Working with services?

If you are working with services you to attach Memory Validator to a service and to start Memory Validator, you should use the NT Service API, not the functions in this API.

All the other functions in this API can be used with applications and with services.

Unicode or ANSI?

All the API functions are provided in Unicode and ANSI variants where strings are used. We've also provided a character width neutral #define in the same fashion that the Windows.h header files do.

For example the function for naming a heap is provided as `mvSetHeapNameA()`, `mvSetHeapNameW()` with the character width neutral `mvSetHeapName()` mapping to `mvSetHeapNameW()` for unicode and `mvSetHeapNameA()` for ANSI.

In this document we're going to use `TCHAR` like the Window.h header files do.

Deploying on a customer machine

You can use the API without incurring any dependency on Memory Validator.

If Memory Validator is not installed on the machine the software runs on, nothing will happen.

This allows you to add the Memory Validator API to your software without need to have a separate build for use with Memory Validator.

Loading the Profiler

For most use cases won't need to load the profiler, as the profiler will have been loaded when your launched your program from Memory Validator, or when you injected into your program using Inject or Wait For Application.

However if you're running your program from outside of Memory Validator and want to load the profiler from inside your program you can use `mvLoadProfiler()` to do that. You'll then need to call `mvStartProfiler()` to start it.

Starting the Profiler

To start the profiler from your API code you need to call the function mvStartProfiler() from your code before you call any API functions. Ideally you should call this function as early in your program as possible.

If you prefer to start the profiler from the user interface or command line you can omit the mvStartProfiler() call. You can leave it present if you wish to start Memory Validator from your program.

Naming threads

You can name threads using the `mvSetThreadName()` function.

Naming heaps

You can name threads using the `mvSetHeapName()` function.

Turning data collection on and off

You can turn data collection on and off using the `mvSetCollect()` function.

You can use `mvGetCollect()` to inspect the data collection status.

Tagging many allocations with the same identifier

You can group many related allocations together by assigning them a tag tracker.

You do this by pushing a tag tracker on to the tag tracker stack.

Any allocations that happen when there are tag trackers on the stack are assigned to that tag tracker.

When you have finished with that tag tracker you can pop it off the stack.

This can be useful for clustering related database allocations, or related game allocations.

For example:

```
mvPushTracker(_T("Flowers"));

f = new Daffodil();
flowers.push_back(f);

f = new Rose();
flowers.push_back(f);

f = new Geranium();
flowers.push_back(f);

mvPopTracker();
```

Having configured "Flowers" as above, in the user interface in the various filtering options you can select Flowers in the tag tracker (or in the Statistics view) to see which Flowers are still allocated without seeing data for any other allocation.

You can also do this with a helpful class: **svlDataTracker**. When the class does out of scope the top of the tag tracker stack is popped.

You can push more than one tag tracker at a time.

```
{
   svlDataTracker(_T("Plants"));

   {
      svlDataTracker(_T("Trees"));

      t = new Oak();                                          // Trees tag tracker
      trees.push_back(t);

      t = new Sycamore();                                     // Trees tag tracker
      trees.push_back(t);

      t = new Cyrpress();                                     // Trees tag tracker
      trees.push_back(t);
   }

   {
      svlDataTracker(_T("Flowers"));

      f = new Daffodil();                                     // Flowers tag tracker
      flowers.push_back(f);

      f = new Rose();                                         // Flowers tag tracker
      flowers.push_back(f);

      f = new Geranium();                                     // Flowers tag tracker
      flowers.push_back(f);
   }
```

```
   g = new Grass();                                          // Plants tag tracker
}

str = new string("weebles wobble but they don't fall down");      // no tag tracker
```

Placing watermarks to identify the start and end of behaviours in the target application

You can place watermarks to identify locations in the stream of memory allocations being monitored.

Done carefully this can allow you to provide easy ways to check that particular parts of your application are behaving as expected.

For example you could place a watermark before a database transaction, and a watermark after a database transaction. Then either during a session, or at the end of the session (when you have the leak report) you can display all allocations between the two watermarks.

You can also place watermarks interactively, but automated watermark placement is much faster, and easier to place them in the correct place.

```
mvSetWatermarkEx(_T("Start purchase"));

if (processPurchase())
{
   commitTransaction();
}
else
{
   rollbackTransaction();
}

mvSetWatermarkEx(_T("End purchase"));
```

Having created two watermarks you can now configure various filtering options on many of Memory Validator's user interfaces to show you data before a watermark, after a watermark and between two watermarks.

If there is nothing shown between two watermarks it's because anything that was allocated between the creation of those two watermarks has been deallocated.

6.1 Native API Reference

Unicode or ANSI?

All the API functions are provided in Unicode and ANSI variants where strings are used. We've also provided a character width neutral #define in the same fashion that the Windows.h header files do.

For example the function for naming a heap is provided as `mvSetThreadNameA()`, `mvSetThreadNameW()` with the character width neutral `mvSetThreadName()` mapping to `mvSetThreadNameW()` for unicode and `mvSetThreadNameA()` for ANSI.

In this document we're going to use `TCHAR` like the Window.h header files do.

All the API functions are declared as `extern "C"`, so they can be used by C users and C++ users.

To use these functions #include `svlMVAPI.h` into your code.

Documentation, A or W?

Where there are functions that take strings we document the function as mvFunction(), but they will be implemented as mvFunctionA() for ANSI or mvFunctionW() for Unicode, using the same convention that Microsoft uses to document their functions.

6.1.1 Loading and Starting the Profiler

mvLoadProfiler

Loads the profiler DLL into memory, but does not start the profiler.

Use this for:
32 bit applications with a 32 bit Memory Validator GUI
64 bit applications with a 64 bit Memory Validator GUI

For most use cases won't need to load the profiler, as the profiler will have been loaded when your launched your program from Memory Validator, or when you injected into your program using Inject or Wait For Application.

However if you're running your program from outside of Memory Validator and want to load the profiler from inside your program you can use `mvLoadProfiler()` to do that. You'll then need to call `mvStartProfiler()` to start it.

```
extern "C"
int mvLoadProfiler();

Returns:
TRUE   Successfully loaded MV DLL into target application.
FALSE  Failed to load the MV DLL into target application.
       Check that the PATH environment variable points to the Memory Validator
install directory contains svlMemoryValidatorStub*.dll.
```

Do not use this function if you are working with services, use the ***NT Service API***.

mvLoadProfiler6432

Loads the profiler DLL into memory, but does not start the profiler.

Use this for:
32 bit applications with a 64 bit Memory Validator GUI

For most use cases won't need to load the profiler, as the profiler will have been loaded when your launched your program from Memory Validator, or when you injected into your program using Inject or Wait For Application.

However if you're running your program from outside of Memory Validator and want to load the profiler from inside your program you can use `mvLoadProfiler6432()` to do that. You'll then need to call `mvStartProfiler()` to start it.

```
extern "C"
int mvLoadProfiler6432();

Returns:
TRUE   Successfully loaded MV DLL into target application.
FALSE  Failed to load the MV DLL into target application.
       Check that the PATH environment variable points to the Memory Validator
install directory contains svlMemoryValidatorStub*.dll.
```

Do not use this function if you are working with services, use the ***NT Service API***.

mvStartProfiler

To start the profiler from your API code you need to call the function `mvStartProfiler()` from your code before you call any API functions. Ideally you should call this function as early in your program as possible.

```
extern "C"
int mvStartProfiler();

Returns:
TRUE   Successfully started MV profiler.
FALSE  Failed to start the MV profiler.
```

If you prefer to start the profiler from the user interface or command line you can omit the `mvStartProfiler()` call. You can leave it present if you wish to start Memory Validator from your program.

Do not use this function if you are working with services, use the ***NT Service API***.

mvIsValidatorPresent

To test if Memory Validator has loaded into this application call `mvIsValidatorPresent()`.

```
extern "C"
int mvIsValidatorPresent();

Returns:
TRUE   Memory Validator is loaded into this process.
FALSE  Memory Validator is not loaded into this process.
```

6.1.2 Custom Heap Tracking

API functions for tracking allocations in custom heaps

The following group of API functions notify Memory Validator of allocation behaviour in a custom memory manager.

The functions cover:

- allocation
- reallocation
- free
- object reference count changes (increment and decrement)

The **dllId** parameter is the index of the Memory Validator extension DLL (if any) to use.

You can pass two items of user defined data.

The `Ex` functions take an extra **tagTracker** parameter - an id returned from a call to `mvAddTracker()`, or 0 if you don't have one.

mvUserCustomAlloc()
mvUserCustomAllocEx()

```
extern "C"
void mvUserCustomAlloc(void*     address,        // address of allocation
                       SIZE_T    size,           // size of allocation (bytes)
                       DWORD_PTR userData1,
                       DWORD_PTR userData2,
                       DWORD     dllId = -1);    // extDll id, -1 if none

extern "C"
void mvUserCustomAllocEx(void*     address,      // address of allocation
                         SIZE_T    size,         // size of allocation (bytes)
                         DWORD_PTR userData1,
                         DWORD_PTR userData2,
                         DWORD     dllId = -1,   // extDll id, -1 if none
                         DWORD     tagTracker);  // see mvAddTracker() to get tracker id
```

mvUserCustomReAlloc()
mvUserCustomReAllocEx()

```
extern "C"
void mvUserCustomReAlloc(void*  oldAddress,  // address of allocation
                         void*  newAddress,  // address of allocation
                         SIZE_T newSize,     // size of allocation (bytes)
                         DWORD  userData1,
                         DWORD  userData2,
                         DWORD  dllId = -1); // extDll id, -1 if none
```

```
extern "C"
void mvUserCustomReAllocEx(void*     oldAddress,  // address of allocation
                           void*     newAddress,  // address of allocation
                           SIZE_T    newSize,     // size of allocation (bytes)
                           DWORD_PTR userData1,
                           DWORD_PTR userData2,
                           DWORD     dllId = -1,  // extDll id, -1 if none
                           DWORD     tagTracker); // see mvAddTracker() to get tracker id
```

mvUserCustomFree()
mvUserCustomFreeEx()

```
extern "C"
void mvUserCustomFree(void* address,     // address of free
                      DWORD userData1,
                      DWORD userData2,
                      DWORD dllId = -1); // extDll id, -1 if none

extern "C"
void mvUserCustomFreeEx(void*     address,     // address of free
                        DWORD_PTR userData1,
                        DWORD_PTR userData2,
                        DWORD     dllId = -1,  // extDll id, -1 if none
                        DWORD     tagTracker); // see mvAddTracker() to get tracker id
```

mvUserCustomRefCountDecrement()
mvUserCustomRefCountDecrementEx()

Notifies Memory Validator that a specific object has had its reference count *decremented*.

This is to provide support for those software using reference counting, but have no way of easily tracking where and when reference counts change.

```
extern "C"
void mvUserCustomRefCountDecrement(DWORD data,        // address of allocation
                                   DWORD userData,
                                   DWORD userData2,
                                   DWORD dllId = -1); // extDll id, -1 if none

extern "C"
void mvUserCustomRefCountDecrementEx(DWORD     data,        // address of allocation
                                     DWORD_PTR userData,
                                     DWORD_PTR userData2,
                                     DWORD     dllId = -1,  // extDll id, -1 if none
                                     DWORD     tagTracker); // see mvAddTracker() to get track
```

mvUserCustomRefCountIncrement()
mvUserCustomRefCountIncrementEx()

Notifies Memory Validator that a specific object has had its reference count *incremented*.

```
extern "C"
void mvUserCustomRefCountIncrement(DWORD data,        // address of allocation
                                   DWORD userData1,
                                   DWORD userData2,
                                   DWORD dllId = -1); // extDll id, -1 if none

extern "C"
void mvUserCustomRefCountIncrementEx(DWORD     data,        // address of allocation
                                     DWORD_PTR userData1,
                                     DWORD_PTR userData2,
                                     DWORD     dllId = -1,  // extDll id, -1 if none
                                     DWORD     tagTracker); // see mvAddTracker() to get track
```

6.1.3 Naming Heaps

API Function for naming heaps

Named heaps are very useful if you want to use the filters to display data.

Filtering heaps with real names make it much easier to watch events than using a thread id, especially as thread ids are quite likely be different between sessions.

mvSetHeapName()

Tells Memory Validator to name a specific heap.

```
extern "C"
void mvSetHeapName(HANDLE       hHeap,    // must be a HeapCreate() handle
                   const TCHAR* name);
```

6.1.4 Naming Threads

API Function for naming heaps

Named threads are very useful if you want to use the thread filter to display data.

Filtering threads with real names make it much easier to watch events than using a thread id, especially as thread ids are quite likely be different between sessions.

mvSetThreadName()

Tells Memory Validator to name a specific thread.

```
extern "C"
void mvSetThreadName(DWORD           threadID,    // id identifying a thread (CreateThread, Cre
                     const TCHAR*    name);
```

6.1.5 Setting Watermarks & Bookmarks

API Functions setting watermarks and bookmarks

mvSetWatermark()

Creates a named watermark in Memory Validator. This watermark applies to data regardless of which thread the data is collected on.

You can use this watermark in the main tab views when filtering results and detecting memory leaks.

The watermarks can also be used in the API to notify your target program (via a callback) of leaks detected after or between named watermarks.

The watermark created will reference the most recent data trace recorded at the time of calling.

```
extern "C"
void mvSetWatermark(const TCHAR   *name);
```

mvSetWatermarkEx()

Creates a named watermark in Memory Validator. This watermark applies to data collected on this thread only.

This is unlike mvSetWatermark which applies to data regardless of which thread the data is collected on.

You can use this watermark in the main tab views when filtering results and detecting memory leaks.

The watermarks can also be used in the API to notify your target program (via a callback) of leaks detected after or between named watermarks.

The watermark created will reference the most recent data trace recorded at the time of calling.

```
extern "C"
void mvSetWatermarkEx(const TCHAR   *name);
```

mvSetBookmark()

Creates a named bookmark, referencing the most recent data trace recorded at the time of calling.

This is useful to automatically generate a *point of interest* in the event history that you might want to locate easily in the main tab views.

```
extern "C"
void mvSetBookmark(const TCHAR   *name);
```

6.1.6 Callbacks for Leaks & Uninitialized Data

API Function callbacks for leaks and uninitialized data

The API leak detection callback facility can call a function in your application if one of the following happens:

- each time a leak is found before the program ends.
- a leak is detected for an allocation made after or between specified watermarks
- uninitialized memory is detected in the C runtime heap

You can setup each callback with an item of user data which will be passed back into the callback.

mvLeakDetect()

A specified callback in your program is called *for each item* of leaked memory.

Usually leaks are detected by monitoring the entire program execution, and are viewed once the target program ends.

Data not deallocated by the time the program ends is deemed to have been leaked, unless it is pointed to by a pointer in static memory, when it is regarded as 'potentially in use'.

 Use with caution. This function is not very efficient and can take a long time to execute if there are lots of allocations. However, it's useful if you really need to know about leaked items before the program has ended.

```
extern "C"
int mvLeakDetect(API_LEAK_CALLBACK callback,  // The callback to call when leaked memory is fo
                 void*             userData,  // User data you want to pass back into the call
                 DWORD             flags);    // Flags used to control what memory is searched
```

The callback needs to have the form:

```
void leakCallback(API_MEMORY_INFO *data,      // Information about the leaked memory. If NULL,
                  void            *userData); // The value you supplied to mvLeakDetect()
```

The following values are available as flags that can be passed to `mvLeakDetect()`. These flags can be combined using the OR operator. These flags are defined in the `allEnum.h` header file.

```
LEAK_DETECT_NONE         0x00000000   // No leak detects are performed

LEAK_DETECT_CPP          0x00000001   // Scan CRT debug heap
LEAK_DETECT_C            0x00000002   // Scan C heap (see heapwalk())
LEAK_DETECT_WIN32_HEAPS  0x00000004   // Scan Win32 heaps (see HeapWalk())
LEAK_DETECT_GLOBAL       0x00000008   // Scan non-dynamic memory in DLLs (variables in non-sta
LEAK_DETECT_STACK        0x00000010   // Scan the stack space of the current thread
LEAK_DETECT_ALL_STACKS   0x00000020   // Not implemented
LEAK_DETECT_ALL_TLS      0x00000040   // Not implemented

LEAK_DETECT_ALL_HEAPS           (LEAK_DETECT_CPP | LEAK_DETECT_C | LEAK_DETECT_WIN32_HEAPS)
LEAK_DETECT_ALL_EXCEPT_STACK    (LEAK_DETECT_CPP | LEAK_DETECT_C | LEAK_DETECT_WIN32_HEAPS | LE
LEAK_DETECT_ALL                 (LEAK_DETECT_CPP | LEAK_DETECT_C | LEAK_DETECT_WIN32_HEAPS | LE
```

We recommend using the flag value `LEAK_DETECT_ALL_EXCEPT_STACK`.

We don't recommend scanning the stack space of the current thread for references to pointers in local variables.

The reasons we don't recommend `LEAK_DETECT_STACK` are:

- Identifying just the stack space used for local variables is slow. This is a serious performance overhead.
- We can't identify the stack space used for local variables for any functions that don't have full debug information. This means any Microsoft code and any third party code (shell extensions etc).
- Scanning all the stack space means we're scanning workspace that was previous used but is now just workspace for any functions on the stack. This space can contain old values from previous function calls leading to FALSE positive identifications of pointers which would be incorrectly reported as found (not leaked).

mvLeakDetectFromWatermark()

Here you specify a watermark, for which a callback is only called for leaks detected for memory allocated after the watermark was created.

The watermark can be created via the API using `mvSetWatermark` or interactively by the user.

```
extern "C"
int mvLeakDetectFromWatermark(const TCHAR*           watermarkName, // The name of the waterma
                              API_WATERMARK_CALLBACK callback,      // The callback to call wh
                              void*                  userData);     // User data you want to p
```

The callback has the form:

```
void leakCallback(DWORD address,    // Address of the leaked memory
                  void  *userData); // The value you supplied to mvLeakDetectFromWatermark()
```

mvLeakDetectBetweenWatermarks()

Similar to `mvLeakDetectFromWatermark`, but the callback is only made for leaks where the memory allocation is between two named watermarks.

```
extern "C"
int mvLeakDetectBetweenWatermarks(const TCHAR*           firstWatermarkName, // The name of th
                                  const TCHAR*           lastWatermarkName,  // The second wat
                                  API_WATERMARK_CALLBACK callback,           // The callback t
                                  void*                  userData);          // User data you
```

The callback has the same form as `mvLeakDetectFromWatermark` above.

mvDetectUninitialised()

Sets a callback to be called once per item of data if uninitialized data is detected in the C runtime heap.

```
extern "C"
int mvDetectUninitialised(API_UNINITIALISED_CALLBACK callback,  // The callback to call when u
                          void*                      userData); // User data you want to pass
```

The callback needs to have the same form as used with `mvLeakDetect`:

```
void leakCallback(API_MEMORY_INFO *data,      // Information about the uninitialized memory. I
                  void            *userData); // The value you supplied to mvLeakDetect()
```

6.1.7 Tag Tracking

API functions for tracking tags

The following group of API functions help your program associate trackable data tags with your memory allocations.

The tags can then be tracked and used as filters to display events in the main display tabs.

The API functions let you:

- create a tag, and get an id for it
- manage a 'stack' of tags, pushing and popping a tag to change the 'current' tag.
- pass a tag id when notifying Memory Validator (via the API) of allocations on a custom heap

mvAddTracker()

Add a tag tracker name. Unlike **mvPushTracker** this does not make it the current tag tracker.

The id for the named tracker is returned and can be used with the API functions when making allocations on a custom heap

```
extern "C"
DWORD mvAddTracker(const TCHAR *trackerName);
```

mvPushTracker()

Push a tag tracker on to the tag stack, making it the 'current' tag tracker associated with allocations.

```
extern "C"
int mvPushTracker(const TCHAR *trackerName);
```

mvPopTracker()

Pop the current tag tracker from the top of the tag stack.

If any more tags are on the stack, the top one becomes current.

```
extern "C"
int mvPopTracker();
```

6.1.8 Data Collection

mvSetCollect()

Enables or disables data collection - i.e. whether data is sent to Memory Validator from your target application.

```
extern "C"
void mvSetCollect(int enable); // TRUE to enable, FALSE to disable
```

mvGetCollect()

Returns whether data collection is on.

```
extern "C"
int mvGetCollect(); // Returns TRUE or FALSE
```

mvIgnoreThisThread()

Use this function to tell Memory Validator to ignore this thread or pay attention to this thread.

```
extern "C"
int mvIgnoreThisThread(); // Returns TRUE or FALSE
```

The return code indicates Memory Validator previous ignore state. You can use this value in a subsequent call to `mvIgnoreThisThread()`.

Example:

```
int prevValue;

prevValue = mvIgnoreThisThread(TRUE); // tell MV to ignore this thread and record the previous

doBakeCake();                         // these operations on this thread are ignored

mvIgnoreThisThread(prevValue);        // tell MV to restore the previous ignore this thr
```

6.1.9 Lifetime Allocations

Some application designs make use of allocations that are intended to last the lifetime of the application.

Typically such allocations will be assigned to static variables and they will never be deallocated.

It is useful if these allocations can be marked as "lifetime allocations" that are intended to leak.

The Memory Validator user interface will not identify lifetime allocations as leaks, thus removing clutter from your memory leak reports.

mvMarkMemoryAsLifetimeAllocation()

Mark memory at this address as a lifetime allocation.

```
extern "C"
void mvMarkMemoryAsLifetimeAllocation(void  *address);
```

mvMarkHandleAsLifetimeAllocation()

Mark this handle as a lifetime allocation.

```
extern "C"
void mvMarkHandleAsLifetimeAllocation(HANDLE handle);
```

6.1.10 Playing Sounds

You can use the API to play sounds from the Memory Validator GUI.

mvPlaySound()

Plays a sound.

The sound is specified by a full path to a Windows .wav file.

```
extern "C"
void mvPlaySound(const TCHAR *fullPathToWavFile);
```

example:

```
mvPlaySound(_T("c:\\windows\\media\\Windows Notify Email.wav"));
```

The request to play the sound is added to the data stream sent to the GUI. The sound is played when the Memory Validator user interface reads this event data.

This allows you to add mvPlaySound() calls alongside other API calls and have then effectively sound at the same time the other events arrive at the GUI.

For example you might want to play a sound to announce that a particular event has happened in the program, and also associate a watermark with that event. You can do that with two API calls.

```
mvSetWatermark(_T("Data Processing Queue Start"));
mvPlaySound(_T("c:\\windows\\media\\Windows Notify Email.wav"));
```

6.1.11 Utility Functions

API utility functions

The API utility functions provide access to a few tools such as

- Dumping leaked objects to a callback or a data file
- Force a garbage collection on the C runtime heap
- Running an integrity check on the C runtime heap
- Shutting down Memory Validator

mvUserDumpLeaks()

Dumps any leaked memory objects to the specified callback, or to a data file, depending on the `dumpMethod` parameter.

```
extern "C"
DUMP_RESULT mvUserDumpLeaks(DUMP_METHOD        dumpMethod, // how to do the dump DUMP_TO_FILE_
                            const TCHAR*       fileName,   // NULL if no file,
                            USER_DUMP_CALLBACK callback,   // NULL if not to callback
                            DWORD              userData);  // userdata to pass to callback
```

mvGarbageCollect()

Forces garbage collection to be performed on the C runtime heap.

This facility is useful to allow cleanup after a function has leaked and would like to start from a known state of 'not leaked'.

 Use with caution. Memory Validator is not designed to be used as a garbage collector, and this function is not very efficient. It can take a long time to execute if there are lots of allocations.

```
extern "C"
int mvGarbageCollect();
```

mvIntegrityCheck()

Perform an integrity check on the C runtime heap.

```
extern "C"
int mvIntegrityCheck();  // Returns TRUE or FALSE depending on whether the check passed
```

mvShutdownMemoryValidator()

Call from your target application to turn off Memory Validator functionality.

```
extern "C"
int mvShutdownMemoryValidator();
```

6.1.12 Example code

Example : apiExample

Please see the **apiExample** application for a simple application that demonstrates the usage of the API.

You can find this in the **examples** folder in the Memory Validator installation directory.

Example : Load and start profiler then set watermarks during execution

This simple code snippet shows how you might use the API in your application.

This example is for a simplistic cooking machinery that mixes ingredients and bakes them into a cake.

Note that when run on a computer that doesn't have Memory Validator installed these API calls will have no effect, and there are no dependencies upon Memory Validator - you can ship your product with these API calls in it.

```
#include "svlMVAPI.h"

int main(int  argc,
         char *argv[])
{
    if (mvLoadProfiler())
    {
        // profiler loaded OK

        if (mvStartProfiler())
        {
            // profiler started OK
            // ensure we have data collection turned on (don't call this if you wish to honour the
```

```
            mvSetCollect(TRUE);
        }
        else
        {
            // failed to start profiler
        }
    }
    else
    {
        // failed to load profiler (is Memory Validator installed on the machine?)
    }

    // place watermarks before and after all major events
    // we can then use the watermarks in the GUI to see which parts of the application leak memory

    mvSetWatermark(_T("Start"));

    doInit();

    mvSetWatermark(_T("AddIngredients"));

    doAddIngredients();

    mvSetWatermark(_T("MixIngredients"));

    doMixIngredients();

    mvSetWatermark(_T("Bake"));

    doBake();

    mvSetWatermark(_T("Cool"));

    doCooldown();

    mvSetWatermark(_T("Shutdown"));

    doShutdown();

    mvSetWatermark(_T("End"));
}
```

6.2 C# API

The C# API is a wrapper around the native API.

For all of these APIs see the native API for more details.

Adding the API to your application

The C# API is provided as a source code `svlMVAPI.cs` file that you add to your application. The source file is in the API directory in the Memory Validator install directory.

The C# API does not add any dependencies to your application - if Memory Validator is present the API functions work, if Memory Validator is not present the API functions do nothing.

The C# API

The C# API is implemented by the `MemoryValidator` class in the `SoftwareVerify` namespace.

6.2.1 Snapshots

makeSnapshot()

Create a snapshot of .Net objects with the name snapshotName.

```
public static void makeSnapshot(string snapshotName);
```

makeSnapshotComparison()

Create a comparison of the most recent 2 snapshots with the name comparisonName.

```
public static void makeSnapshotComparison(string comparisonName);
```

6.2.2 Object Inactivity

setStaleStartupThreshold()

Mark this location in program execution as the the stale startup threshold location.

```
public static void setStaleStartupThreshold();
```

setStaleIgnoreThreshold()

Set the number of garbage collections to use as the stale object ignore threshold.

```
public static void setStaleIgnoreThreshold(UInt32 threshold);
```

enableObjectActivityDataCollection()

Enable object activity data collection.

```
public static void enableObjectActivityDataCollection(bool enable);
```

setStaleInactivityStart()

Set the stale inactivity start location.

```
public static void setStaleInactivityStart();
```

setStaleInactivityEnd()

Set the stale inactivity end location.

```
public static void setStaleInactivityEnd();
```

setStaleInactivityIgnore()

Set the stale inactivity ignore threshold. Specify the number of garbage collections to ignore.

```
public static void setStaleInactivityIgnore(UInt32 ignore);
```

6.2.3 Watermarks & Bookmarks

setWatermark()

Set a watermark at the current location.

```
public static void setWatermark(string name);
```

setWatermarkEx()

Set a watermark at the current location.

```
public static void setWatermarkEx(string name);
```

watermarkRangeStart()

Mark the start of a watermark range.

```
public static void watermarkRangeStart(string name);
```

watermarkRangeEnd()

Mark the end of a watermark range. The specified name must match the name passed to previous call to `watermarkRangeStart`.

```
public static void watermarkRangeEnd(string name);
```

setBookmark()

Set a bookmark at the current location.

```
public static void setBookmark(string name);
```

6.2.4 Tag Tracking

addTracker()

Add a tag tracker. The tag tracker is not made the current tracker. It's ID is returned from the function for future use with tag tracking functions.

```
public static UInt32 addTracker(string trackerName);
```

pushTracker()

Push a tag tracker onto the tag tracker stack. The tag tracker at the top of the stack (if there is one) will be used to tag memory and handle allocations.

```
public static void pushTracker(string trackerName);
```

popTracker()

Pop a tag tracker from the tag tracker stack. The tag tracker at the top of the stack (if there is one) will be used to tag memory and handle allocations.

```
public static void popTracker();
```

6.2.5 Data Collection

collectOn()

Turn data collection on.

```
public static void collectOn();
```

collectOff()

Turn data collection off.

```
public static void collectOff();
```

setCollect()

Turn data collection on or off.

```
public static void setCollect(bool enable);
```

getCollect()

Determine if data collection is turned on or off.

```
public static bool getCollect();
```

6.2.6 Utility Functions

isPresent()

Returns true if Memory Validator is loaded into the process.

```
public static bool isPresent();
```

shutdownMemoryValidator()

Shutdown Memory Validator. After this call completes Memory Validator's stub will do no work.

```
public static void shutdownMemoryValidator();
```

enableStubSymbols()

Enable symbols in the stub.

```
public static void enableStubSymbols();
```

6.3 Calling the API via GetProcAddress

Calling API functions using GetProcAddress

If you don't want to use the svlMVAPI.c/h files you can use `GetProcAddress()` to find the interface functions in the Memory Validator DLL.

The interface functions have different names and do not use C++ name mangling, but have identical parameters to the API functions.

To determine the function name take any native API name, replace the leading mv with **api**. For example `mvSetWatermarkEx()` becomes `apiSetWatermarkEx();`

Example usage

```
typedef void (*mvSetWatermark_FUNC)(const TCHAR   *name);

HMODULE getValidatorModule()
{
   HMODULE   hModule;

   hModule = GetModuleHandle(_T("svlMemoryValidatorStub6432.dll"));            // 32 bit DLL wi
   if (hModule == NULL)
      hModule = GetModuleHandle(_T("svlMemoryValidatorStub_x64.dll"));  // 64 bit DLL with 64 b
   if (hModule == NULL)
      hModule = GetModuleHandle(_T("svlMemoryValidatorStub.dll"));             // 32 bit DLL wi

   return hModule;
}
```

```
HMODULE   hMod;

// get module, will only succeed if Memory Validator launched this app or is injected into thi

hMod = getValidatorModule();
if (hMod != NULL)
{
    // MV is present, lookup the function and call it to set a watermark for this location in

    mvSetWatermark_FUNC   pFunc;

    // "apiSetWatermark" is equivalent to linking against "mvSetWatermark"

    pFunc = (mvSetWatermark_FUNC)GetProcAddress(hMod, "apiSetWatermark");
    if (pFunc != NULL)
    {
        (*pFunc)(watermarkName);
    }
}
```

API functions and their GetProcAddress names

For any API functions not listed, try looking up the name in `svlMemoryValidatorStub.dll` using depends.exe or PE File Browser.

⊟ Show API functions and GetProcAddress names

API Name	GetProcAddress() Name
mvAddTrackerA	apiAddTrackerA
mvAddTrackerW	apiAddTrackerW
mvDetectUninitialise	apiDetectUninitialised
mvGarbageCollect	apiGarbageCollect
mvGetCollect	apiGetCollect
mvIntegrityCheck	apiIntegrityCheck
mvLeakDetect	apiLeakDetect
mvLeakDetectBetweenW	apiLeakDetectBetweenWatermarksA
mvLeakDetectBetweenW	apiLeakDetectBetweenWatermarksW
mvLeakDetectFromWate	apiLeakDetectFromWatermarkA
mvLeakDetectFromWate	apiLeakDetectFromWatermarkW
mvPushTrackerA	apiPushTrackerA
mvPushTrackerW	apiPushTrackerW
mvPopTracker	apiPopTracker
mvSetBookmarkA	apiSetBookmarkA
mvSetBookmarkW	apiSetBookmarkW
mvSetCollect	apiSetCollect
mvSetHeapNameA	apiSetHeapNameA
mvSetHeapNameW	apiSetHeapNameW
mvSetThreadNameA	apiSetThreadNameA
mvSetThreadNameW	apiSetThreadNameW
mvSetWatermarkA	apiSetWatermarkA
mvSetWatermarkW	apiSetWatermarkW
mvShutdownMemoryVal	apiShutdownMemoryValidator
mvUserCustomAlloc	informAlloc
mvUserCustomReAlloc	informReAlloc
mvUserCustomFree	informFree

```
mvUserCustomRefCountInformRefCountDecrement
mvUserCustomRefCountInformRefCountIncrement
mvUserDumpLeaks     informDumpLeaks
```

Other exported functions

You may see some other functions exported from `svlMemoryValidatorStub.dll(_x64).dll`.

 These other functions are for Memory Validator's use. Using them may damage memory locations and/or crash your code. Best not to use them!

6.4 Convenience functions

Convenience functions

One convenience function is provided that will start the Memory Validator GUI (if it is not already running), then load the Memory Validator leak tracker into your process and start monitor memory usage.

```
extern "C"
int loadValidatorIntoApplication();

Returns:
TRUE   Successfully loaded MV DLL into target application and successfully
started the profiler.
FALSE   Failed to load the MV DLL or failed to start the profiler.
```

To use this function #include `loadValidatorIntoApplication.h` into your code.

The source files can be found in the `API` directory in the Memory Validator install directory.

```
loadValidatorIntoApplication.h
loadValidatorIntoApplication.c
```

Just add these files to your project and build.

Part VII

7 Working with IIS and Services

When working with NT services your account must have the appropriate privileges described in the User Permissions topic.

Attaching to your service

To use Memory Validator with NT Services you need to link a small library to your application and call two functions in the library.

The NT Service API

The NT Service API is provided to enable Memory Validator to work with services.

The API works just as well with normal applications, and the same considerations outlined here also apply generally.

When the NT Service API is used, source code symbols are acquired in the stub and sent to the Memory Validator user interface.

Monitoring the service

When working with Memory Validator and services using the NT Service API you don't start the service using Memory Validator.

Instead, you start the service the way you normally start the service - e.g. with the service control manager.

The code that you have embedded into your service then contacts Memory Validator, which you should have running *before* starting the service.

Once you've exercised your service and stopped it, Memory Validator will show the usual leak information

See the section on monitoring a service for details.

Examples and help

We provide some Example Service Source Code to demonstrate how to embed the service code into your service.

If you have problems using Memory Validator with services, please contact us at support@softwareverify.com.

7.1 NT Service API

The Memory Validator stub service libraries

The NT Service API is very simple, consisting of functions to load, start and unload the Memory Validator DLL.

We have also provided some debugging functions to help you debug the implementation of the NT Service API because getting data into and out of services is not always straightforward.

The stub service libraries used for this are shown in the following table:

	32 bit Memory Validator	64 bit Memory Validator

32 bit service	`svlMVStubService.lib`
	`svlMVStubServiceMD.lib`
	`svlMVStubServiceMT.lib`
		`svlMVStubService6432.lib`

64 bit service	N/A	`svlMVStubService_x64.lib`
		`svlMVStubServiceMD_x64.lib`
		`svlMVStubServiceMT_x64.lib`

All the functions exported from these libraries are exported as `extern "C"` so that C and C++ users can use them.

Library name suffixes

The MD suffix indicates the library was built with the /MD compiler switch.
The MT suffix indicates the library was built with the /MT compiler switch.

Directory Name: 2010 or 2012?

Visual Studio 6 to Visual Studio 2010
If you are using Visual Studio 2010 or earlier, use libraries from a directory with 2010 in the directory name.

Visual Studio 2010 to Visual Studio 2022
If you are using Visual Studio 2012 or later, use libraries from a directory with 2012 in the directory name.

Header files

The header files can be found in the `svlMVStubService` directory in the Memory Validator install directory.

The headers file provide an error enumeration and the NT Service API functions.

```
svlMVStubService.h
svlServiceError.h
```

Linker Problems

Some linkers cannot link the stub service library file. If you have this problem see What do I do if I cannot use svlMVStubService.lib?

Loading the Memory Validator DLL into your service

To load the Memory Validator stub dll `svlMemoryValidatorStub(_x64).dll` into your service, call `svlMVStub_LoadMemoryValidator()`, *not* `LoadLibrary()`.

If you are monitoring a 32 bit service with the 64 bit Memory Validator user interface you should use `svlMVStub_LoadMemoryValidator6432()`.

Shutting down the Memory Validator DLL from your service.

To shutdown Memory Validator's monitoring of the service , call `svlMVStub_ShutdownMemoryValidator()`.

This sends the shutting down notification and removes any hooks for your process.

Calling this function is optional. You can stop your service without calling this function.

Unloading the Memory Validator DLL from your service.

To unload the Memory Validator stub dll, call `svlMVStub_UnloadMemoryValidator()`, not `FreeLibrary()`.

Calling this function is optional. You can stop your service without calling this function.

Setting a service notification callback

Once you have successfully loaded the Memory Validator DLL you can setup a service callback so that the service control manager can be kept updated during the process of starting the validator.

When a service is starting, Windows requires the service to inform the Service Control Manager (SCM) that is starting at least every ten seconds.

Failure to do so results in Windows concluding that the service has failed to start, and the service is terminated.

Instrumenting your service may well take *more* than 10 seconds, depending on the complexity and size of your service.

The solution is for *Memory Validator* to periodically call a user supplied callback from which you can regularly inform the SCM of the appropriate status.

You can set the service callback with `svlMVStub_SetServiceCallback(callback, userParam)`.

Usage

Here is an example callback which ignores the userParam.

```
void serviceCallback(void   *userParam)
{
    static DWORD dwCheckPoint = 1;

    ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    ssStatus.dwServiceSpecificExitCode = 0;

    ssStatus.dwControlsAccepted = 0;

    ssStatus.dwCurrentState = dwCurrentState;
    ssStatus.dwWin32ExitCode = dwWin32ExitCode;
    ssStatus.dwWaitHint = dwWaitHint;

    ssStatus.dwCheckPoint = dwCheckPoint++;

    // Report the status of the service to the service control manager.

    return SetServiceStatus(sshStatusHandle, &ssStatus);
}
```

Starting Memory Validator DLL in your service

To start Memory Validator detecting memory leaks in your service call svlMVStub_StartMemoryValidator.

Starting Memory Validator DLL in IIS

To start Memory Validator profiling IIS call svlMVStub_StartMemoryValidatorForIIS().

Setting a filename for all logging data to be written to

To set the filename for all debugging/logging information to be written to call svlMVStub_setLogFileName().

Deleting the logfile

To delete the log file call svlMVStub_deleteLogFile().

Writing text to the logfile

To write a standard ANSI character string to the log file call svlMVStub_writeToLogFileA(text). The ANSI string will be converted to Unicode prior to writing to the log file.

To write a Unicode character string to the log file call svlMVStub_writeToLogFileW(text).

Writing error code descriptions to the logfile

To write a human readable description of the SVL_SERVICE_ERROR error code to the log file call `svlMVStub_writeToLogFile(errCode)`.

Writing LastError code descriptions to the logfile

To write a human readable description of the Windows error code to the log file call `svlMVStub_writeToLogFileLastError(errCode)`.

Dumping the PATH environment to the logfile

To write the contents of the PATH environment variable to the log file call `svlMVStub_dumpPathToLogFile()`.

This can be useful if you want to know what the search path is when trying to debug why a DLL wasn't found during an attempt to load the Validator DLL.

7.1.1 Changes to the NT Service API

API changes - February 2018

To make the API easier to use with services we made the following changes:

- Changed the API by adding many debugging functions to allow you to easily log information.
- We also extended the error enumeration to provide additional error status values.
- We also split the function of loading and starting Memory Validator into two functions - a *load* function and a *start* function.
- We split the functionality so that you could setup a service callback prior to calling the *start* function.

  The service callback allows the service control manager to be informed that the service is still active during time consuming operations, such as starting the Memory Validator when the service is non-trivial in scope.

  Failure to inform the service control manager results in the service being killed by the service control manager because it thinks the service has hung.

  This change in the API is to ensure you get better results from using our software.

What do you need to do to move from the old API to the new API?

Change all SVL_ERROR declarations to SVL_SERVICE_ERROR.

Your previous startup code probably looked like this:

```
SVL_ERROR errCode;

errCode = svlMVStub_LoadMemoryValidator();
```

Change it to this:

```
SVL_SERVICE_ERROR errCode;

errCode = svlMVStub_LoadMemoryValidator();

errCode = svlMVStub_SetServiceCallback(serviceCallback, NULL);

errCode = svlMVStub_StartMemoryValidator();
```

The serviceCallback would look something like this:

```
void serviceCallback(void   *userParam)
{
    static DWORD dwCheckPoint = 1;

    ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    ssStatus.dwServiceSpecificExitCode = 0;

    ssStatus.dwControlsAccepted = 0;

    ssStatus.dwCurrentState = dwCurrentState;
    ssStatus.dwWin32ExitCode = dwWin32ExitCode;
    ssStatus.dwWaitHint = dwWaitHint;
    ssStatus.dwCheckPoint = dwCheckPoint++;

    // Report the status of the service to the service control manager.

    return SetServiceStatus(sshStatusHandle, &ssStatus);
}
```

In the code above we have omitted error handling. To see how to use the new logging function with error handling pleas

Important.

Once your service is running (rather than starting) your service callback should set the appropriate running status `SERVICE_RUNNING` rather than `SERVICE_START_PENDING`.

```
if (!ReportStatusToSCMgr(SERVICE_RUNNING,       // service state
                         NO_ERROR,              // exit code
                         0))                    // wait hint
{
   dwErr = GetLastError();
   if (bLogging)
      svlMVStub_writeToLogFileW(L"ReportStatusToSCMgr:5\r\n");
   goto cleanup;
}
```

An alternative solution is to prevent the service callback from being called once the service status has been set to running.

```
svlMVStub_SetServiceCallback(NULL, NULL);.
```

7.1.2 NT Service API Reference

The API consists of the following functions.

SVL_SERVICE_ERROR Enumeration

```
typedef enum _svlServiceError
{
   SVL_OK,                                                  // Normal behaviour
   SVL_ALREADY_LOADED,                                      // Stub DLL already loaded into serv
   SVL_LOAD_FAILED,                                         // Failed to load stub DLL into serv
   SVL_FAILED_TO_ENABLE_STUB_SYMBOLS,                       // Loaded DLL, but failed to enable
   SVL_NOT_LOADED,                                          // Couldn't unload DLL because DLL n
   SVL_FAIL_UNLOAD,                                         // Couldn't unload DLL because could
   SVL_FAIL_TO_CLEANUP_INTERNAL_HEAP,                       // Couldn't get the internal stub he
   SVL_FAIL_MODULE_HANDLE                                   // Couldn't get the stub DLL handle
   SVL_FAIL_SETSERVICECALLBACK,                             // Couldn't call the set service cal
   SVL_FAIL_COULD_NOT_FIND_ENTRY_POINT,                     // Couldn't find the DLL entry point
   SVL_FAIL_TO_START,                                       // Failed to start the Validator
   SVL_FAIL_SETSERVICECALLBACKTHRESHOLD,                    // Couldn't call the set service cal
   SVL_FAIL_PATHS_DO_NOT_MATCH,                             // Path to service in env vars doesn
   SVL_FAIL_INCORRECT_PRODUCT_PREFIX,                       // Wrong validator
   SVL_FAIL_X86_VALIDATOR_FOUND_EXPECTED_X64_VALIDATOR,     // Found wrong bit depth validator
   SVL_FAIL_X64_VALIDATOR_FOUND_EXPECTED_X86_VALIDATOR,     // Found wrong bit depth validator
   SVL_FAIL_DID_YOU_MONITOR_A_SERVICE_FROM_VALIDATOR,       // Looks like Monitor A Service wasn
   SVL_FAIL_ENV_VAR_NOT_FOUND,                              // Env Var not found
   SVL_FAIL_VALIDATOR_ENV_VAR_NOT_FOUND,                    // Env Var identifying validator not
   SVL_FAIL_VALIDATOR_ID_NOT_SPECIFIED,                     // Validator process not specified
   SVL_FAIL_VALIDATOR_ID_NOT_A_PROCESS,                     // Validator process identified does
   SVL_FAIL_VALIDATOR_NOT_FOUND,                            // Validator process identified does
} SVL_SERVICE_ERROR;
```

svlMVStub_LoadMemoryValidator

```
extern "C"
SVL_SERVICE_ERROR svlMVStub_LoadMemoryValidator();
```

To load the Memory Validator stub svlMemoryValidatorStub.dll into your service, use svlMVStub_LoadMemoryValidator(), *not* LoadLibrary().

This loads the DLL and sets up a few internal variables in the DLL to ensure that symbols are sent from the stub to the Memory Validator user interface.

This is necessary because the Memory Validator user interface can't open a process handle to a service and so is unable to get symbols from the process.

To solve this, symbols are sent from the stub to the user interface as needed.

If you just call `LoadLibrary()` on the DLL, symbols will *not* be sent to the `Memory` Validator user interface and you won't get meaningful function names in your stack traces.

This function can be used when monitoring:

- 32 bit services or applications with `Memory` Validator
- 64 bit services or applications with `Memory` Validator x64

If you are monitoring 32 bit applications with `Memory` Validator x64 you should use `svlMVStub_LoadMemoryValidator6432()`.

Which function you should call is shown in the table below.

	32 bit `Memory` Validator	64 bit `Memory` Validator
32 bit service	`svlMVStub_LoadMemoryValidator()`	`svlMVStub_LoadMemoryValidator6432()`
64 bit service	N/A	`svlMVStub_LoadMemoryValidator()`

svlMVStub_LoadMemoryValidator6432

```
extern "C"
SVL_SERVICE_ERROR svlMVStub_LoadMemoryValidator6432();
```

To load the `Memory` Validator stub `svlMemoryValidatorStub6432.dll` into your service, use `svlMVStub_LoadMemoryValidator6432()`, *not* `LoadLibrary()`.

This loads the DLL and sets up a few internal variables in the DLL to ensure that symbols are sent from the stub to the `Memory` Validator user interface.

This is necessary because the `Memory` Validator user interface can't open a process handle to a service and so is unable to get symbols from the process.

To solve this, symbols are sent from the stub to the user interface as needed.

If you just call `LoadLibrary()` on the DLL, symbols will *not* be sent to the `Memory` Validator user interface and you won't get meaningful function names in your stack traces.

This function should only be used when monitoring 32 bit services or applications with Memory Validator x64.

svlMVStub_StartMemoryValidator

```
extern "C"
SVL_SERVICE_ERROR svlMVStub_StartMemoryValidator();
```

To start Memory Validator inspecting the service call `svlMVStub_StartMemoryValidator()`.

svlMVStub_StartMemoryValidatorForIIS

```
extern "C"
```

```
SVL_SERVICE_ERROR svlMVStub_StartMemoryValidatorForIIS();
```

To start Memory Validator inspecting IIS call `svlMVStub_StartMemoryValidatorForIIS()`.

Example usage.

svlMVStub_ShutdownMemoryValidator

```
extern "C"
SVL_SERVICE_ERROR svlMVStub_ShutdownMemoryValidator();
```

To stop Memory Validator inspecing the service call `svlMVStub_Shutdown`Memory`Validator()`.

This sends the shutting down notification and removes any hooks for your process.

Calling this function is optional. ***You can stop your service without calling this function.***

svlMVStub_UnloadMemoryValidator

```
extern "C"
SVL_SERVICE_ERROR svlMVStub_UnloadMemoryValidator();
```

To unload Memory Validator call `svlMVStub_UnloadMemoryValidator()`, *do not call* `FreeLibrary()`.

Calling this function is optional. ***You can stop your service without calling this function.***

svlMVStub_SetServiceCallback

```
extern "C"
SVL_SERVICE_ERROR svlMVStub_SetServiceCallback(serviceCallback_FUNC callback,
                                               void*                userParam);
```

`svlMVStub_SetServiceCallback` is used to setup a service callback that is used to inform the Windows service cor

`userParam` is a value you can supply which will then be passed to the callback every time the callback is called during instrumentation.

Why is a service callback needed?

When a service is starting, Windows requires the service to inform the Service Control Manager (SCM) that is starting at least every ten seconds.

Failure to do so results in Windows concluding that the service has failed to start, and the service is terminated.

Instrumenting your service may well take *more* than 10 seconds, depending on the complexity and size of your service.

The solution is for *Memory Validator* to periodically call a user supplied callback from which you can regularly inform the SCM of the appropriate status.

We strongly recommend that you setup a service callback. Not setting a service callback can result in failure of your

Debugging functions

The following functions are provided to help you log information about the progress, success or failure of the NT Service API attaching Memory Validator to your service.

We strongly recommend that you use these logging functions so that you can understand why Memory Validator might fail to connect to a service.

To see example usage of these debugging functions please look in `service.cpp` in the `examples\service` directory in the Memory Validator install directory.

svlMVStub_setLogFileName

```
extern "C"
void svlMVStub_setLogFileName(const wchar_t* fileName);
```

Call `svlMVStub_setLogFileName` to set the name of the filename used for logging.

This function must be called before you can use any of the other debugging functions.

Setting this filename also sets the filename used by some of these API functions - you will find additional logging data from those functions that will help debug any issues with the service.

svlMVStub_deleteLogFile

```
extern "C"
void svlMVStub_deleteLogFile();
```

This function deletes the log file.

svlMVStub_writeToLogFileA

```
extern "C"
void svlMVStub_writeToLogFileA(const char* text);
```

This function writes a standard ANSI character string to the log file.

The ANSI string will be converted to Unicode prior to writing to the log file.

svlMVStub_writeToLogFileW

```
extern "C"
```

```
void svlMVStub_writeToLogFileW(const wchar_t* text);
```

This function writes a Unicode character string to the log file.

svlMVStub_writeToLogFile

```
extern "C"
void svlMVStub_writeToLogFile(SVL_SERVICE_ERROR errCode);
```

This function writes a human readable description of the SVL_SERVICE_ERROR error code to the log file.

svlMVStub_writeToLogFileLastError

```
extern "C"
void svlMVStub_writeToLogFileLastError(DWORD errCode);
```

This function writes a human readable description of the Windows error code to the log file.

The errCode parameter is the error code returned from GetLastError().

svlMVStub_dumpPathToLogFile

```
extern "C"
void svlMVStub_dumpPathToLogFile();
```

This function writes the contents of the PATH environment variable to the log file.

This can be useful if you want to know what the search path is when trying to debug why a DLL wasn't found during an attempt to load the Validator DLL.

7.1.3 Troubleshooting

Troubleshooting - Service fails to start

If a service takes too long to start the service control manager kills the service.

The way to stop this is for a service to call `ReportStatusToSCMgr()` to tell the service control manager that the service is still OK.

Memory Validator can't do this for you as the call requires some data from any earlier call you have made.

The solution is that you provide a callback using `svlMVStub_SetServiceCallback()` that Memory Validator can call during the process of attaching to the service, and you can call the appropriate function.

Example code to set the callback:

```
	errCode = svlMVStub_SetServiceCallback(serviceCallback,		// the
callback
						NULL);			// some
user data (we don't have any, so set NULL)
	if (bLogging)
	{
		if (errCode != SVL_OK)
		{
			svlMVStub_writeToLogFileW(L"Setting service callback failed.
\r\n");
			svlMVStub_writeToLogFile(errCode);
		}

		svlMVStub_writeToLogFileW(L"Starting Memory Validator\r\n");
	}
```

Example callback:

```
static void serviceCallback(void  *userParam)
{
	// just tell the Service Control Manager that we are still busy
	// in this example userParam is not used
	//
	// note that prior to the Validator loading it's DLL ssStatus.dwCurrentState
must have been initialised, most likely to SERVICE_START_PENDING
	// you could pass a fixed value here, but it would need to change once the
service has finished starting up so that you don't unintentionally change the service
state
	// when this callback is called. This callback is called whenever
instrumentation happens (when a DLL is loaded). Thus you can't assume this is only
called during service startup,
	// it may also get called later in the service lifetime.

	ReportStatusToSCMgr(ssStatus.dwCurrentState,	// service state
			NO_ERROR,		// exit code
			3000);			// wait hint
}
```

We strongly recommend that you set a service callback. It won't harm your program and it will remove any likelihood of your service being killed by the service control manager.

Troubleshooting - Service starts, Memory Validator gets no data

If you have problems getting Memory Validator to monitor your service you'll need to find out what's failing.

Until Memory Validator loads correctly and successfully connects to the graphical user interface you have no way of knowing what is happening.

The solution is to set a log file that Memory Validator can write status messages to. You can also write your own status messages to this log file.

Set the log file using `svlMVStub_setLogFileName`. Write to it using `svlMVStub_writeToLogFile()`, `svlMVStub_writeToLogFileA()`, `svlMVStub_writeToLogFileW()`.

Then when things are not working as expected take a look at the log file to see the errors. The Memory Validator will often suggest what the problem is.

We strongly recommend that you configure the log file and use it when working with services. It has saved us a lot of time.

7.2 Working with IIS

Configuring IIS for use with ISAPI

We assume that you are familiar with IIS. This is not a topic we can provide advice for.

That said, we wrote a blog article about configuring IIS for use with ISAPI.

Example ISAPI

We have provided an example ISAPI extension configured for use with Memory Validator.

This example is provided as source code and project files. You will need to build it yourself, you may need to change an include path to find the appropriate headers. The resulting ISAPI will need to be copied to your website for testing and the website configured to allow the ISAPI to execute (please see the above mentioned blog article for details on that).

You can find the example ISAPI in the **isapiExample** folder in the Memory Validator installation directory.

Using Memory Validator with IIS

IIS is a service application. It runs as one of the more restricted applications on Microsoft Windows.

Memory mapped files created by IIS cannot be opened by user mode programs (Memory Validator, for example). DLLs, executables and files cannot be opened by IIS except if they are in directories which IIS has access to. These are security measures intended to make your computer secure from attack.

These security measures make it hard for tools like Memory Validator to work.

- We have to communicate settings information to Memory Validator via text file
- All DLLs and helper programs we want to use need to be copied to the web root (or a subdirectory within the web root) so that they can be used.
- We need to have our own data transport because our usual high speed memory mapped data transport is not available.

It's also not possible to launch IIS or inject into a running IIS instance.

The only way to work with IIS is by using the NT Service API, and using the `svlMVStub_StartMemoryValidatorForIIS()` function instead of `svlMVStub_StartMemoryValidator()`.

We've provide some example code to show you how to attach and detach from your ISAPI extension.

Workflow

1) Start monitoring your ISAPI by using the Monitor ISAPI dialog.

Launch menu > **IIS** menu > **Monitor ISAPI...**

2) When you have finished interacting with the web pages that use the ISAPI component shutdown IIS, wait for Memory Validator's status to indicate "Ready" and examine the results.

Launch menu > **IIS** menu > **Stop IIS**

7.3 Example Source Code

Service Example

Example demonstrating how to monitor a service.

Also see the example service that ships with Performance Validator.

You can find this in the `\examples\service` directory in the Performance Validator install directory.

Also see the example service and child process that ships with Performance Validator.

You can find this in the `\examples\serviceWithAChildProcess` directory in the Performance Validator install directory.

IIS Example

Example demonstrating how to monitor an ISAPI DLL.

Also see the example ISAPI DLL that ships with Performance Validator.

You can find this in the `\examples\isapiExample` directory in the Performance Validator install directory.

7.3.1 Example Service Source Code

Where to put your code

When you use the functions to load and unload Memory Validator from your service, it is important that you put the function calls in the correct place in your software.

The correct place to put them is in a 'balanced' location, such that you would expect no memory leaks to occur between the load and the unload function call, assuming the service was working correctly.

Typically, this means that Memory Validator is:

- loaded as the first action in the `service_main()` function
- unloaded just before the service control manager is informed of the stopped status

The source code shown below shows an example `service_main()` function used in a service, demonstrating where to load and unload Memory Validator.

The long comment covers problems with the way services are stopped and what may be displayed in a debugger if this happens.

➡ The code is extracted from `service\service.cpp`, part of the full example of an NT service, client and a utility for controlling whether the service uses Memory Validator.

⊞ Show the C++ example `service_main()` function

```
void serviceCallback(void   *userParam)
{
   // just tell the Service Control Manager that we are still busy
   // in this example userParam is not used

   static DWORD dwCheckPoint = 1;

   ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
   ssStatus.dwServiceSpecificExitCode = 0;

   ssStatus.dwControlsAccepted = 0;

   ssStatus.dwCurrentState = dwCurrentState;
   ssStatus.dwWin32ExitCode = dwWin32ExitCode;
   ssStatus.dwWaitHint = dwWaitHint;
   ssStatus.dwCheckPoint = dwCheckPoint++;

   // Report the status of the service to the service control manager.

   return SetServiceStatus(sshStatusHandle, &ssStatus);
}

void WINAPI service_main(DWORD   dwArgc, LPTSTR   *lpszArgv)
{
   if (bLogging)
   {
      svlMVStub_setLogFileName(SZLOGFILENAME);
      svlMVStub_deleteLogFile();
   }
```

```
   // register our service control handler:

   sshStatusHandle = RegisterServiceCtrlHandler(TEXT(SZSERVICENAME), service_ctrl);
   if (sshStatusHandle != 0)
   {
      DWORD   dwErr = 0;

      // **MV_EXAMPLE** start

#if _HONOUR_MV_MUTEX_LOAD
      if (bMemoryValidator)
#endif   //#if _HONOUR_MV_MUTEX_LOAD
      {
         // load Memory Validator (but if monitoring a 32 bit service with C++ Memory
Validator x64 use svlMVStub_LoadMemoryValidator6432())

         if (bLogging)
         {
            svlMVStub_writeToLogFileW(_T("About to load C++ Memory Validator\r\n"));
         }

         SVL_SERVICE_ERROR   errCode;
#ifdef IS6432
         // x86 with x64 GUI
         errCode = svlMVStub_LoadMemoryValidator6432();
#else   //#ifdef IS6432
         // x86 with x86 GUI
         // x64 with x64 GUI
         errCode = svlMVStub_LoadMemoryValidator();
#endif   //#ifdef IS6432
         if (bLogging)
         {
            if (errCode != SVL_OK)
            {
               DWORD   lastError;

               lastError = GetLastError();
               svlMVStub_writeToLogFileW(_T("C++ Memory Validator load failed.
\r\n"));
               svlMVStub_writeToLogFileLastError(lastError);
               svlMVStub_writeToLogFile(errCode);

               svlMVStub_dumpPathToLogFile();
            }
            else
            {
               svlMVStub_writeToLogFileW(_T("C++ Memory Validator load success.
\r\n"));
            }
         }

         // setup a service callback so that the Service Control Manager knows the
service
         // is starting up even if instrumentation takes longer than 10 seconds (which
it will
         // for a non-trivial application)
```

```
         if (bLogging)
            svlMVStub_writeToLogFileW(_T("Setting service callback C++ Memory
Validator\r\n"));

         errCode = svlMVStub_SetServiceCallback(serviceCallback,      // the callback
                                                NULL);            // some user data
(we don't have any, so set NULL)
         if (bLogging)
         {
            if (errCode != SVL_OK)
            {
               svlMVStub_writeToLogFileW(_T("Setting service callback failed. \r\n"));
               svlMVStub_writeToLogFile(errCode);
            }

            svlMVStub_writeToLogFileW(_T("Starting C++ Memory Validator\r\n"));
         }

         errCode = svlMVStub_StartMemoryValidator();
         if (bLogging)
         {
            if (errCode != SVL_OK)
            {
               DWORD   lastError;

               lastError = GetLastError();
               svlMVStub_writeToLogFileW(_T("Starting C++ Memory Validator failed.
\r\n"));
               svlMVStub_writeToLogFileLastError(lastError);
               svlMVStub_writeToLogFile(errCode);
            }

            svlMVStub_writeToLogFileW(_T("Finished loading C++ Memory
Validator\r\n"));
         }
      }
#if _HONOUR_MV_MUTEX_LOAD
      else
      {
         if (bLogging)
            svlMVStub_writeToLogFileW(_T("Not using C++ Memory Validator, DLL will not
be loaded\r\n"));
      }
#endif   //#if _HONOUR_MV_MUTEX_LOAD

      // **MV_EXAMPLE** end

      // SERVICE_STATUS members that don't change in example

      ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
      ssStatus.dwServiceSpecificExitCode = 0;

      // report the status to the service control manager.

      if (ReportStatusToSCMgr(SERVICE_START_PENDING, // service state
```

```
                                NO_ERROR,              // exit code
                                3000))                 // wait hint
      {
         // deliberately allocate some memory so we can see that
         // with Memory Validator

         char   *someLeakedMemory;

         someLeakedMemory = (char *)malloc((SIZE_T)3456);

         // do work

         dwErr = ServiceStart(dwArgc, lpszArgv);

         // finished doing work
      }

      // **MV_EXAMPLE** start

#if _HONOUR_MV_MUTEX_LOAD
      if (bMemoryValidator)
#endif   //#if _HONOUR_MV_MUTEX_LOAD
      {
         // unload Memory Validator here
         // IMPORTANT.
         // Because of the way services work, you can find that this thread which is
trying to gracefully unload
         // MemoryValidator is ripped from under you by the operating system. This
prevents Memory Validator from
         // removing all its hooks successfully. If Memory Validator does not remove
all of its hooks successfully
         // because this happens, then you may get a crash when the service stops.
         //
         // An alternative fix is to spawn another thread which then unloads Memory
Validator.
         // See the code for ServiceStop() for comments relating to this.
         //
         // A callstack for such a crash is shown below. If you see this type of crash
you need to put your code to
         // unload Memory Validator somewhere else. The stack trace may be different,
but a fundamental point is the
         // code calling through doexit(), exit() and ExitProcess()
         //
         //NTDLL! 77f64e70()
         //SVLMEMORYVALIDATORSTUB!
         //MSVCRT! 78001436()
         //MSVCRT! 7800578c()
         //DBGHELP! 6d55da25()
         //DBGHELP! 6d55de83()
         //DBGHELP! 6d53705d()
         //DBGHELP! 6d51cc69()
         //DBGHELP! 6d51f6e8()
         //DBGHELP! 6d524ebf()
         //DBGHELP! 6d52a7b0()
```

```
            //DBGHELP! 6d52b00a()
            //DBGHELP! 6d526487()
            //DBGHELP! 6d5264d7()
            //DBGHELP! 6d5264f7()
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //SVLMEMORYVALIDATORSTUB!
            //MSVCRT! 78001436()
            //MSVCRT! 780057db()
            //KERNEL32! 77f19fdb()
            //SVLMEMORYVALIDATORSTUB! ExitProcess hook
            //doexit(int 0x00000000, int 0x00000000, int 0x00000000) line 392
            //exit(int 0x00000000) line 279 + 13 bytes
            //mainCRTStartup() line 345
            //KERNEL32! 77f1b9ea()

            svlMVStub_UnloadMemoryValidator();
        }

        // **MV_EXAMPLE** end

        // try to report the stopped status to the service control manager.

        (VOID)ReportStatusToSCMgr(SERVICE_STOPPED, dwErr, 0);
    }

    return;
}
```

7.3.2 Example ISAPI Source Code

Where to put your code

When you use the functions to load and unload Memory Validator from your service, it is important that you put the function calls in the correct place in your ISAPI extension.

Typically, this means that Memory Validator is:

- loaded as the first action in the `GetExtensionVersion()` function of your ISAPI extension.
- unloaded in the `TerminateExtension()` function of your ISAPI extension.

Example source code

The source code shown below shows an example `GetExtensionVersion()` and an example `TerminateExtension()` used in an ISAPI, demonstrating where to load and unload Memory Validator.

This example code logs errors. We strongly recommend that you do this in your example. Because IIS is a protected process that can't communicate to the outside world except via HTTP/HTTPS when anything fails during the loading and start of Memory Validator the only means we have of communicating that failure to you is via the log file. Please use the log file, it will make debugging any mistakes very much easier, simpler and quicker than any other method.

This process is almost identical to working with a regular service, except that `svlMVStub_StartMemoryValidator()` is replaced with `svlMVStub_StartMemoryValidatorForIIS()`.

This example assumes the web root is located `C:\\testISAPIWebsite`

Show the C++ example ISAPI functions

```
#include "svlMVStubService.h"
#include "svlServiceError.h"

BOOL WINAPI GetExtensionVersion(HSE_VERSION_INFO *pVer)
{
	// some setup work to define what the extension is

	pVer->dwExtensionVersion = HSE_VERSION;
	strncpy(pVer->lpszExtensionDesc, "Validate ISAPI Extension",
HSE_MAX_EXT_DLL_NAME_LEN);

	// load Validator here

	svlMVStub_setLogFileName(L"C:\\testISAPIWebsite\\svl_MV_log.txt");
	svlMVStub_deleteLogFile();

	SVL_SERVICE_ERROR   errCode;
#ifdef IS6432
	// x86 with x64 GUI
	errCode = svlMVStub_LoadMemoryValidator6432();
#else   //#ifdef IS6432
	// x86 with x86 GUI
	// x64 with x64 GUI
	errCode = svlMVStub_LoadMemoryValidator();
#endif   //#ifdef IS6432
	if (errCode != SVL_OK)
	{
		DWORD   lastError;

		lastError = GetLastError();
		svlMVStub_writeToLogFileW(L"C++ Memory Validator load failed. \r\n");
		svlMVStub_writeToLogFileLastError(lastError);
		svlMVStub_writeToLogFile(errCode);

		svlMVStub_dumpPathToLogFile();
	}
	else
	{
		svlMVStub_writeToLogFileW(L"C++ Memory Validator load success. \r\n");

		errCode = svlMVStub_StartMemoryValidatorForIIS();
		if (errCode != SVL_OK)
```

```
		{
			DWORD	lastError;

			lastError = GetLastError();
			svlMVStub_writeToLogFileW(L"Starting C++ Memory Validator failed.
\r\n");
			svlMVStub_writeToLogFileLastError(lastError);
			svlMVStub_writeToLogFile(errCode);
		}

		svlMVStub_writeToLogFileW(L"Finished starting C++ Memory
Validator\r\n");
	}

	return TRUE;
}


BOOL WINAPI TerminateExtension(DWORD	dwFlags)
{
	// unload Validator here

	svlMVStub_UnloadMemoryValidator();

	return TRUE;
}
```

Part

8 Working with Marmalade game SDK

This topic describes how to use C++ Memory Validator when you are developing programs with the Marmalade game SDK.

Assumptions about Marmalade

We assume that you have installed the Marmalade SDK and Marmalade Hub in the default location of c:\marmalade.

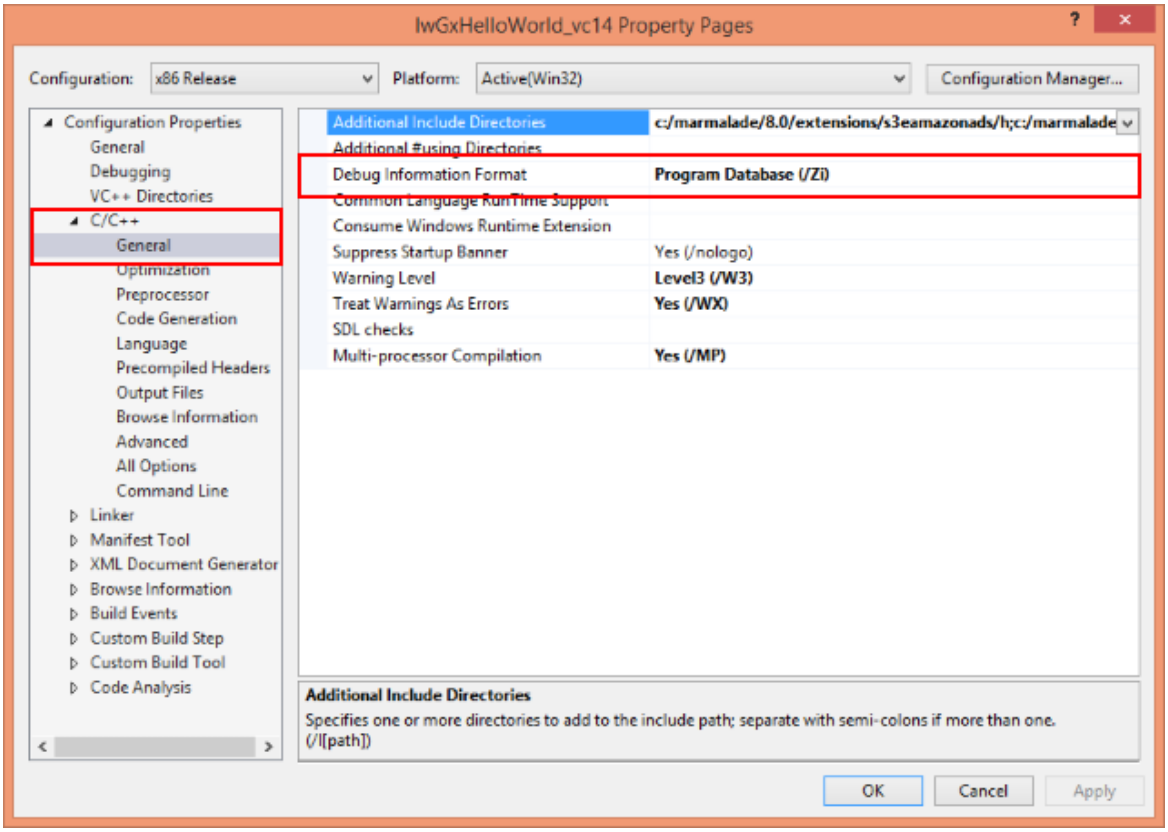
We assume that you are developing your game for many targets, but that for the purpose of testing with C++ Memory Validator you are also targeting **x86 Debug** and **x86 Release** configurations.

Preparing your game

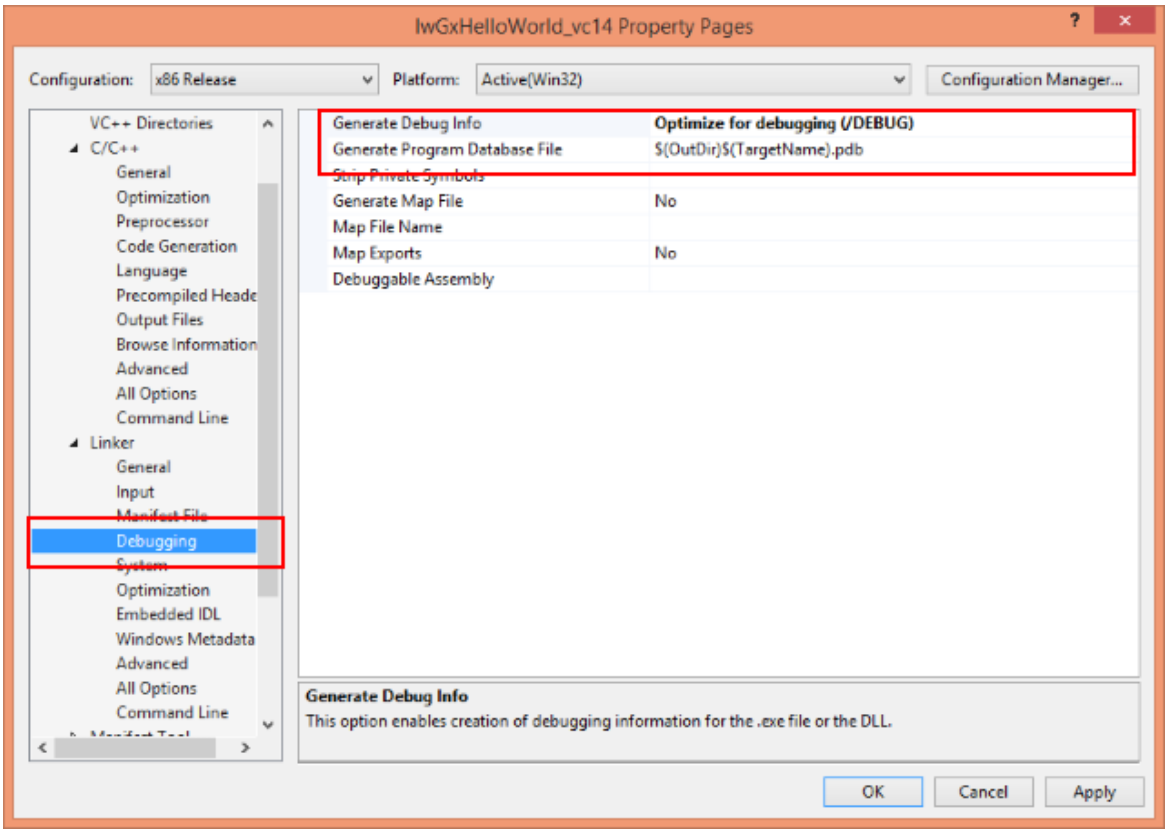
To work with C++ Memory Validator you need to build the x86 Debug configuration and/or the x86 Release configuration of your Marmalade project using Visual Studio.

These configurations need to be built so that they create debug information and so that a PDB file containing debug information is created. The example projects that ship with Marmalade do not do this - you will need to edit them to make the linker stage create debug information.

- Editing the compile stage debug information



- Editing the link stage debug information



You must ensure that both compile and link stages have the correct settings set. If only compile or only link is set you will not get debugging symbols.

Debugging symbols are important for two reasons:

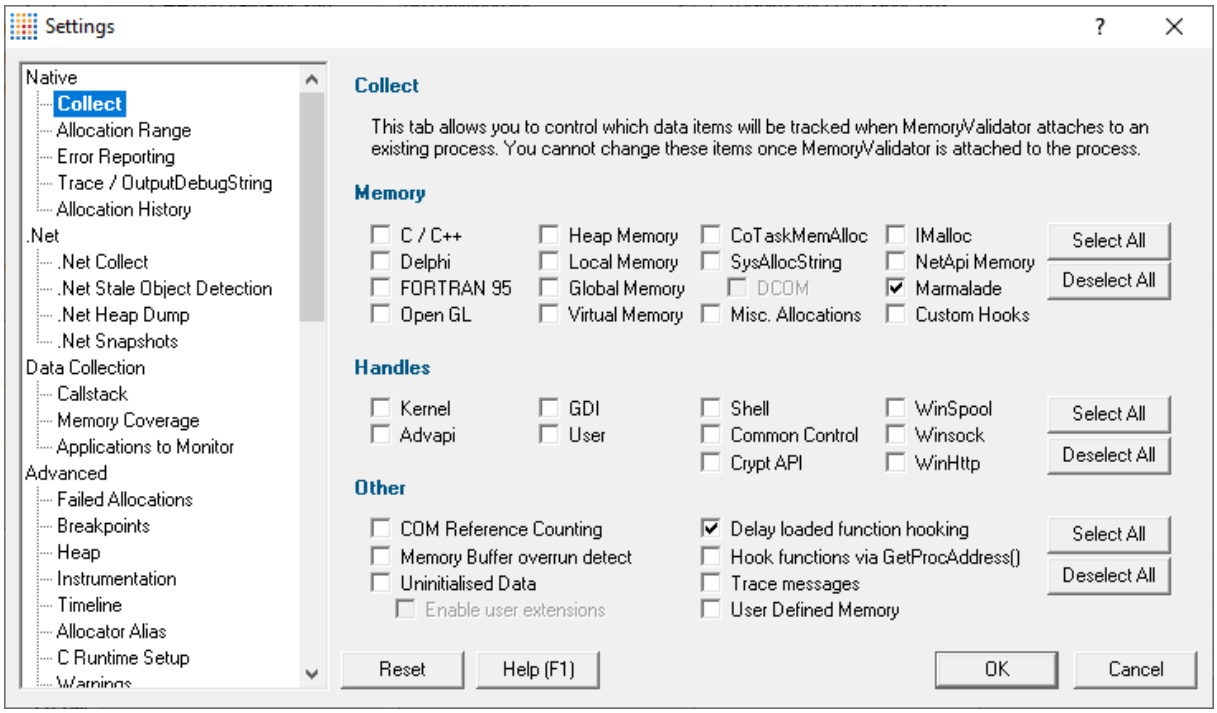
- Without symbols C++ Memory Validator cannot find the Marmalade memory allocators and will not be able to track the Marmalade memory allocations your game makes

- Without symbols C++ Memory Validator will not be able to turn callstack addresses into class names, function names, filenames and line numbers.

Preparing C++ Memory Validator

To work with Marmalade it's best to go back to a known state then make a few changes to setup C++ Memory Validator for use with Marmalade.

1. Open the settings dialog, click Reset.
2. Go to the Collect settings
3. Disable all check boxes in the Memory section, except for Marmalade.
4. Disable all check boxes in the Handles section.
5. Click OK.



The reason for selecting these options is that it prevents C++ Memory Validator from monitoring the actions of the Marmalade simulator itself. If you're a developing a Marmalade game you only care about what your game does - the memory and handle allocation behaviour of the simulator are not your concern, thus you don't want to monitor them. That is why we've turned everything off except for the Marmalade check box.

Launching the game

To launch a Marmalade game with C++ Memory Validator we launch the Marmalade simulator and specify the game to run using the Marmalade **-via** command line argument.

If Marmalade is installed in `c:\marmalade` then the path to the simulator is

```
c:/marmalade/8.0/s3e\win32\s3e_simulator_release.exe
```

If an example game (shipped with Marmalade) is found at this location

```
c:
\Marmalade\8.0\examples\GameTutorial\Lua\Stage9\build_temp\build_stage9_vc14\Stage9
_vc14_release.via
```

then the command line is

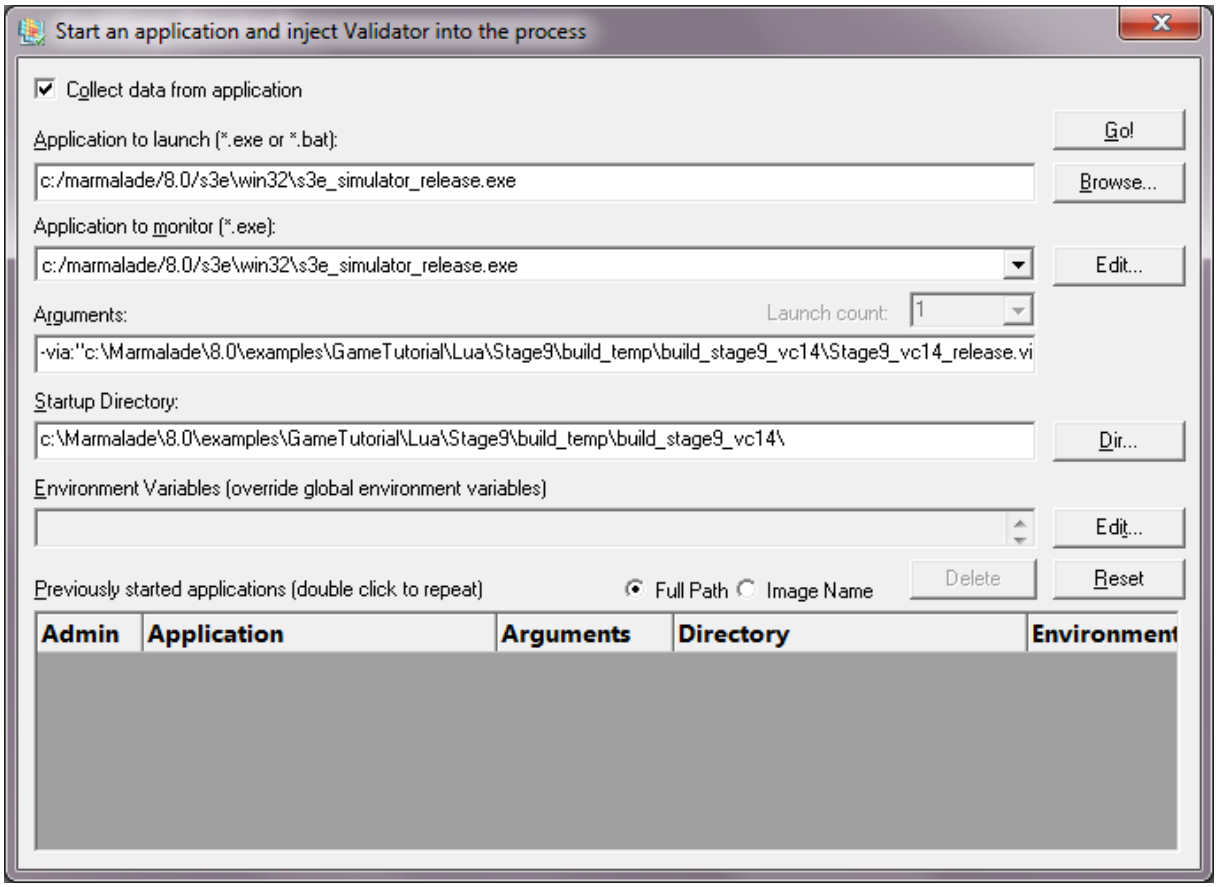
```
-via:"c:
\Marmalade\8.0\examples\GameTutorial\Lua\Stage9\build_temp\build_stage9_vc14\Stage9
_vc14_release.via"
```

and the startup directory is

```
c:\Marmalade\8.0\examples\GameTutorial\Lua\Stage9\build_temp\build_stage9_vc14\
```

This is how the launch dialog looks when you are launching this game.

We leave the Application to monitor unchanged. It should have the same value as Application to launch.



Part

9 Working with Intel Math Kernel Library

This topic describes how to use C++ Memory Validator when you are developing programs with the Intel Math Kernel Library.

Intel Math Kernel Library

The Intel Math Kernal Library (MKL) provides memory allocation functions for you to use to allocate memory aligned on specific memory boundaries for performance reasons.

The allocations functions provided by MKL mimic the basic C allocation functions malloc(), calloc(), realloc() and free(). These are provided in the form of mkl_malloc(), mkl_calloc(), mkl_realloc() and mkl_free().

It is natural that you would wish Memory Validator to track these allocations. MKL is provided in both statically linked and dynamically linked forms. Because of the way MKL is built it is not possible for Memory Validator to just track the MKL allocations as with a normal allocator. This is because MKL uses the C runtime heap internally and also allocates various memory for housekeeping tasks. To track your usage of MKL correctly we should not report on the internal workings of MKL or it's housekeeping tasks.

The solution to this to call MKL functions via some Software Verify functions. We provide two source files for you to permanently include in your software. If your program is not being profiled by Memory Validator, the mkl functions are called directly. If your program is being profiled by Memory Validator, the mkl function (de)(re)allocations are tracked and the associated underlying CRT allocations and housekeeping are ignored.

Usage

Follow the following steps to change your program to use MKL and allow MKL allocations to be tracked by Memory Validator.

Source Files
Include `svl_mkl_helper.h` and `svl_mkl_helper.cpp` in each DLL/EXE that uses Intel Math Kernel Library.

You can use these two source files without any royalty payments to Software Verify Limited. We expect these two files to be used by your software. While we believe these source files are fit for purpose and we have tested them to our satisfaction, you use them at your own risk and should satisfy yourself that they work correctly and to your satisfaction.

Initialisation
Call svl_mkl_init() before you call any MKL function. This applies to each DLL/EXE.

For example if your program has an EXE that uses MKL and a DLL that uses both the EXE and the DLL must call svl_mkl_init().

Replace function calls
call `svl_mkl_malloc()` rather than `mkl_malloc()`
call `svl_mkl_calloc()` rather than `mkl_calloc()`
call `svl_mkl_realloc()` rather than `mkl_realloc()`
call `svl_mkl_free()` rather than `mkl_free()`

call `svl_mkl_free_buffers()` rather than `mkl_free_buffers()`
call `svl_mkl_free_thread_buffers()` rather than `mkl_free_thread_buffers()`

The parameters passed to the svl_mkl_...() functions are the same as the parameters pass to mkl_...() functions. No change is needed.

Running your program without Memory Validator

Your program will run as normal when Memory Validator is not present. The MKL library functions will be called.

Running your program with Memory Validator

Your program will run as normal when Memory Validator is present. The MKL library functions will be called and the allocations and deallocations will be tracked by Memory Validator

Part

10 Working with Visual Basic 6 (VB6)

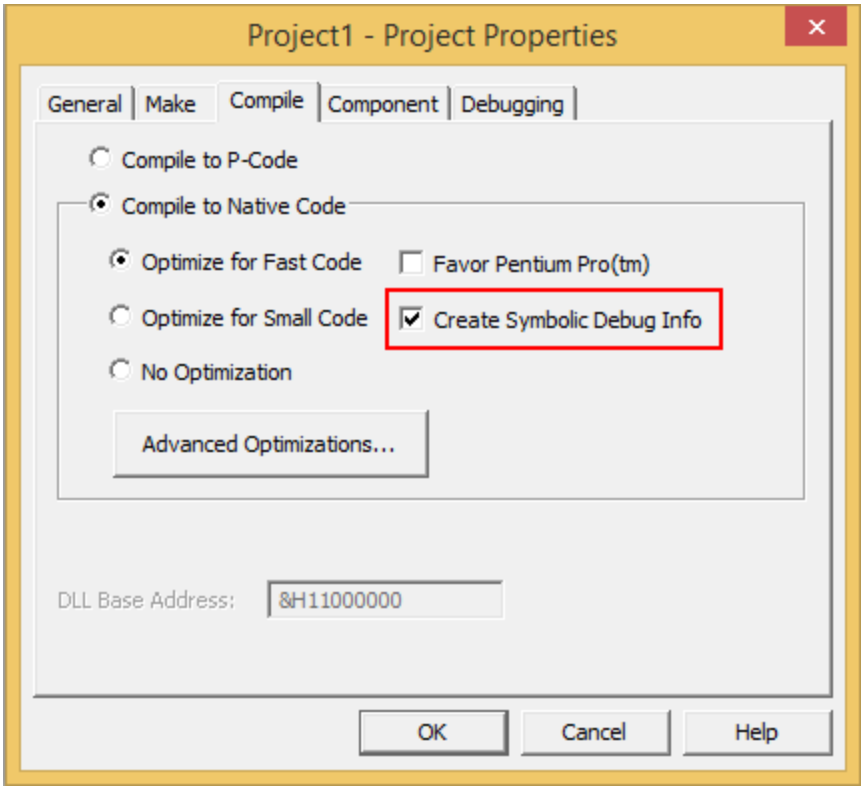
To work with VB6 you need to make two simple changes to your VB6 way of working.

Enable debug information

Debug information needs to be generated to enable Memory Validator to be able to report function names, filename and line numbers.

To do this, in Visual Basic 6, go to the **Project** menu, choose **Project Properties...**.

Then go to the **Compile** tab and select **Create Symbolic Debug Info**. Click **OK**.



Compile your program into an executable

The next step is to build your program as an executable.

From the **File** menu, choose **Make <name-of-program.exe>**.

Your program will be compiled as an executable. Another file will be created with the same name, but instead of .exe (or .dll) as an extension, the extension will be .pdb. This file contains the debugging information.

For example *test.exe* will create a debugging information file called *test.pdb*.

If you always keep the pdb file in the same directory as the .exe Memory Validator will be able to find the debugging symbols.

Part

11 Extending Memory Validator

There are a small set of cases where Memory Validator can be extended by user supplied DLLs:

- custom display of data associated with custom heap allocations
- notification of entry/exit for constructor and destructors
- handling of commands sent by the user via Memory Validator

User Interface extensions

User interface extensions affect data collected by the family of custom heap tracking functions in the API.

The data is displayed by Memory Validator using your own custom message format defined in the extension DLL.

A simple example of a ui extension DLL is provided with Memory Validator.

Stub extensions

Stub extensions allow custom code to be executed at the start and end of each constructor and destructor that is called in the target program.

These extension functions are provided with:

- a 'this' parameter identifying the object being constructed or destructed
- a function address
- a function name

The stub extension DLL may choose to store the data in an internal store etc.

Stub extension DLLs also allow for a (`char*`) command to be sent from Memory Validator by the user to one or all stub extension DLLs.

The extension DLL may act on the command as it wishes.

A simple example of a stub extension DLL is provided with Memory Validator.

 You should ensure that UI and stub extension DLLs don't have memory leaks or access violations for example, as these will prevent Memory Validator from working correctly.

11.1 Example user interface extension DLL

The user interface extension DLL functionality lets you provide additional descriptive messaging for displaying with user defined allocations.

➜ See also the mvUserCustomAlloc family of functions used for custom heap tracking using the Memory Validator API.

Example UI extension project files

The example extension project can be found in the `uiExtDLL` subdirectory in the directory where Memory Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

There are two project files in the directory:

- **uiExtDLL.dsp** › for Microsoft® Developer Studio® 6.0
- **uiExtDLL.vcproj** › for Microsoft® Visual Studio / .net

UI extension DLL functions

A user interface extension DLL needs to provide just two functions:

- **getDllID** › uniquely identifies the user interface extension DLL

  ```
  DWORD getDllID();
  ```

  Each ui extension DLL must return a different value than any of the other user interface extension DLLs.

  A value of -1 returned means that the DLL should not be used as an extension DLL.

- **getDescription** › allocates a buffer to return the description of an object

  ```
  int getDescription(DWORD   userData1,
                     DWORD   userData2,
                     DWORD   address,
                     DWORD   size,
                     DWORD   refCount,
                     wchar_t **description);
  ```

  The function is passed the userData1 and userData2 values that were passed into the API function that reported the object e.g. `mvUserCustomAlloc()`.

  The address, size and reference count are also passed in.

  The function must allocate a buffer using `HeapAlloc()` and use it to return the description of the object.

The buffer should be on the program's default Heap as returned by GetProcessHeap().

Defining the functions for export and import

In the `uiExtDLL` example, the functions are defined as `extern "C"` functions which are exported from the DLL.

See the header file `uiExtDLL.h` for an example way of declaring the functions so that when building the DLL, the functions are exported, but anyone including the header file sees the functions as imported.

Example ui extension DLL code

The code snippet below is taken directly from the `uiExtDLL.cpp` implementation file in the uiExtDLL project

Although userData1 and userData2 are not interpreted here, an example use might be to pass indices into a known array of strings or enumerated values used to generate a message.

Show the C++ example ui extension functions

```
//-NAME---------------------------------
// getDllID
//.DESCRIPTION.........................
// Provide ID for Memory Validator Extension DLL
//.PARAMETERS.........................
//.RETURN.CODES.......................
// -1 means don't use
// Otherwise return a unique ID to identify this DLL.
//-------------------------------------

UIEXTDLL_API DWORD getDllID()
{
   return 1;   // make this ID unique amongst all the UI extension DLLs you have
}

//-NAME---------------------------------
// getDescription
//.DESCRIPTION.........................
// Generate a description for the data provided.
//.PARAMETERS.........................
// userData1         -in-  Supplied to mvUserCustomAlloc
// userData2         -in-  Supplied to mvUserCustomAlloc
// address           -in-  Address/Handle/ID to track
// size              -in-  Size for the address
// refCount          -in-  Reference count for the address
// **description     -out- Returned string allocated with HeapAlloc(GetProcessHeap(), (
//.RETURN.CODES.......................
// TRUE for got a description.
// FALSE for failed.
//-------------------------------------

UIEXTDLL_API int getDescription(DWORD   userData1,
                        DWORD   userData2,
                        DWORD   address,
                        DWORD   size,
                        DWORD   refCount,
                        wchar_t   **description)
{
   TCHAR   text[200];
   int      len;

   _stprintf(text, _T(" UI-Ext-DLL UserObject: address:0x%08x size:0x%08x ref:%d 0x%08x
            address, size, refCount, userData1, userData2);
   len = _tcslen(text);

   *description = (TCHAR *)HeapAlloc(GetProcessHeap(), 0, sizeof(TCHAR) * (len + 1));
   if (*description != NULL)
   {
      _tcscpy(*description, text);
      return TRUE;
   }

   return FALSE;
}
```

11.2 Example stub extension DLL

The stub extension DLL functionality allows a custom DLL to be notified at the start and end of each constructor and destructor visited in the target program.

One additional function can receive a command which may be sent to all stub extensions or a specific one.

Example stub extension project files

The example extension project can be found in the stub`ExtDLL` subdirectory in the directory where Memory Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

There are two project files in the directory:

- **stubExtDLL.dsp** › for Microsoft® Developer Studio® 6.0
- **stubExtDLL.vcproj** › for Microsoft® Visual Studio / .net

UI extension DLL functions

A stub extension DLL can provide all or none of the following functions. If a function isn't defined, the stub won't call it.

All of these functions return TRUE / FALSE for success or failure, but at the time of writing, the stub doesn't act on the return codes.

Constructor notification:

- **startConstructor** › called at the start of the C++ object's constructor, before any member variables have been initialised

```
int startConstructor(void *thisPtr,
                     void *functionAddress,
                     char *functionName);
```

- **finishConstructor** › called at the end of the C++ object's constructor, after any member variables have been initialised

```
int finishConstructor(void *thisPtr,
                      void *functionAddress,
                      char *functionName);
```

Destructor notification:

- **startDestructor** › called at the start of the C++ object's destructor

```
int startDestructor(void *thisPtr,
                    void *functionAddress,
                    char *functionName);
```

- **finishDestructor** › called at the end of the C++ object's destructor

```
int finishDestructor(void *thisPtr,
                     void *functionAddress,
                     char *functionName);
```

- **command** › called when you use Memory Validator's send command to stub utility to send a custom message to a one or all stub DLLs

```
int command(char *command);
```

Derived classes

For class B derived from class A, the `startConstructor()` and `finishDestructor()` calls will be called for the constructor of class A and class B for the same object.

The same is true for destructors, but note that class B's destructor is called prior to that of class A for the same object.

Example stub extension DLL code

Example code for all the above functions is provided in the `stubExtDLL.cpp` implementation file in the stubExtDLL project

Defining the functions for export and import

In the `stubExtDLL` example, the functions are defined as `extern "C"` functions which are exported from the DLL.

See the header file `stubExtDLL.h` demonstrating a way of declaring the functions so that when building the DLL, the functions are exported, but anyone including the header file sees the functions as imported.

Part XII

12 Examples

The need for examples

We know Memory Validator is a complex product, but sometimes the programs that need to be validated are even more complex, and are certainly all different.

For this reason, it's important to be able to test and demonstrate the features of Memory Validator in an easy and repeatable way.

The example application makes this possible:

- It has examples of most of the types of errors that can be detected
- It lets you trigger events in your own time so you can observe them with Memory Validator
- It provides source code to demonstrate usage, correctly or otherwise!

This section has help for the example application followed by some examples of using it in conjunction with Memory Validator.

Some additional projects provide examples of using NT Services.

All example projects are supplied as source code and projects. You'll need to build the example or services before you can use them.

12.1 The example application

The example application

The example application is a great way to explore the capabilities of Memory Validator.

The source and projects are included in the installation, but you'll need to build the example application yourself.

Once built, you can use nativeExample.exe in conjunction with Memory Validator to

- monitor the behaviour of the application as you use it
- easily allocate memory and handles in many different ways
- observe deliberate memory and handle leaks
- cause and catch allocation and corruption errors

The example program can also be linked with the Memory Validator API to demonstrate API usage.

How to use these examples

The best way to understand how Memory Validator works is by example.

We recommend launching the example application from Memory Validator and observing how the menu actions affect captured information.

Examining the source code is the best way to see what's going on in the example application.

For convenience, we have provided the source locations where each menu action runs a test.

All test locations are in the `CTeststakView` class of `nativeExample\TESTSVW.CPP`

Menu options - a summary

- **File** › exit only!
- **Allocation** › typical memory allocation, reallocation and deallocation as well as memory leaks
- **Memory Errors** › typical memory errors, such as buffer overrun, underrun, corruption, incorrect deallocation or reallocation, uninitialized data detection etc.
- **Handles** › creation and deletion of Win32 resources. Thread creation and deletion to test cross thread allocation detection
- **Trace** › send example TRACE messages to Memory Validator
- **DLL** › explicit loading and unloading of DLLs
- **Reporting** › call various API functions dumping leaking or uninitialised objects to a file, or to callback functions
- **Help** › shows the about box (which allocates and leaks memory in the process)

We give more detail on the menu options in the example application, but the *real* detail of allocations and errors are in the source code itself.

The code is commented and some areas have detailed explanations of what is going on and the likely consequences so do take a look!

A warning!

Some of the tests deliberately corrupt the C runtime heap or the program stack, to demonstrate errors and Memory Validator's ability to detect them.

Generally, the example application is robust, but by the very nature of some of these examples, it's not unreasonable that it may crash on occasion!

12.1.1 Building the example application

Where to find the example application

The example project is in the **nativeExample** subdirectory where Memory Validator installation directory.

If the directory is not present, reinstall your software and choose *custom* or *full* installation to include the examples.

Solutions and projects

There are a variety of solutions and projects for different versions:

- **nativeExample.dsp** › for Microsoft® Developer Studio® 6.0
- **nativeExample_VSx_x.sln** › for Microsoft® Visual Studio / .net

Configurations

There are a number of configurations in each project, the first two of which are linked with Memory Validator for API use:

- **Debug / Release** › linked to the **svlMemoryValidatorStubLib(_x64).lib** demonstrating use with the Memory Validator API

The other configurations are standalone programs:

- **DebugNonLink** / **ReleaseNonLink** › not linked for API use
- **DebugStatic / ReleaseStatic** › statically linked to the C runtime and not linked for API use
- **ESA Debug / ESA Release** › uses the Cherrystone Extremely Scalable Allocator (ESA)

  You'll need the full or evaluation version of ESA from Cherrystone Software Labs Inc. for these configurations to work.

Using Visual Studio Express?

You might find you can't build the example application with Express versions of Visual Studio because it doesn't provide all the necessary libraries.

If that's the case, try searching for the missing libraries in one of the freely available Windows SDKs from the Microsoft website.

If you use Visual Studio Express to build your *own* application, Memory Validator will still work with it just fine.

Other projects

There are some other projects installed alongside the nativeExample directory.

These optional projects build various DLLs that can be used by the example program, subject to user interaction.

They demonstrate detection of the loading and unloading of DLLs, memory leaks caused in DLLs that are subsequently unloaded, and other testable behaviours.

12.1.2 Allocation menu

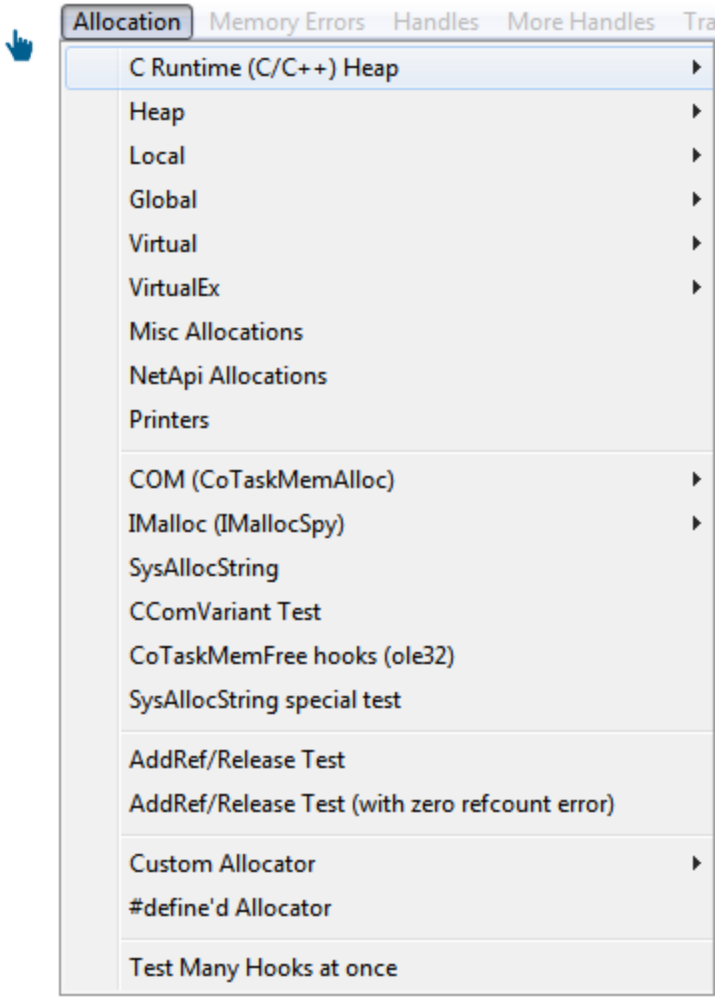
The allocation menu

The **Allocation** menu tests and demonstrates different groups of allocation, reallocation and deallocation examples.

Each test is listed below with the source code location in the example application.

➡ How to use these examples

Click on the picture below to jump to the relevant sections:



Allocation menu : C Runtime (C/C++) Heap

Allocation menu › **C Runtime (C/C++) Heap** submenu › **...**

Tests CRT heap allocation functions.

Allocate memory	`OnTestAllocatememory()` Prompts for a number of bytes to allocate and how many times to do it. The memory is not deallocated.
Reallocate memory	`OnTestReallocatememory()` Prompts for a number of bytes to allocate and how many times to reallocate it, growing in size each time. The memory is not deallocated.
Allocate using malloc()	`OnTestAllocateusingmalloc()` Allocates 100 bytes of memory. The memory is not deallocated.
Allocate using calloc()	`OnTestAllocateusingcalloc()` Prompts for a number of bytes to allocate. The memory is not deallocated.
Expand memory using _expand()	`OnTestExpandmemoryusingexpand()` Allocated memory and then expands it using `_expand()`. Two more calls to expand the memory pass incorrect memory pointers.
Shrink memory using _expand()	`OnTestShrinkmemoryusingexpand()` Allocates memory and then shrinks it using `_expand()`. Two more calls to shrink the memory pass incorrect memory pointers.
Allocate CObject	`OnTestAllocatecobject()` Allocates a **CWnd** object.
Allocate CString	`OnTestAllocatecstring()` Allocates various **TestClass** objects freeing all but one.
Multiple Allocation of CObject	`OnTestMultipleallocationofcobject()` Allocates 5 **CWnd** objects.
Multiple Allocation of CString	`OnTestMultipleallocationofcstring()` Allocates 5 **CString** objects
Multiple Allocation of CObject Leaving 1	`OnTestMultipleallocationofcobjectLeaving1()` Allocates 5 **CWnd** objects, then deletes 4 of them
Multiple Allocation of CObject Leaving 2	`OnTestMultipleallocationofcobjectLeaving2()` Allocates 5 **CWnd** objects, then deletes 3 of them
Multiple reallocation using malloc() and realloc()	`OnTestMultiplereallocation()` Allocates and reallocates 3 pointers in a loop so the callstacks for each allocation are identical.
Multi allocation, reallocation, deallocation using malloc, realloc, free	`OnMultiAlloc()` Prompts for a number of allocations to make, each of random size. Allocations are freed

Allocation menu : Heap

Allocation menu › **Heap** submenu › **...**

Tests Win32 heap functions `HeapCreate()`, `HeapDestroy()`, `HeapAlloc()`, `HeapRealloc()` and `HeapFree()`.

Create Heap	`OnHandlesCreateheap()` Create a heap using `HeapCreate()`. **Do this first to enable the other tests.**
Destroy Heap	`OnHandlesDestroyheap()` Destroy the heap created above.
Allocate Memory	`OnHeapAllocatememory()` Allocate 100 bytes on the heap
Allocate and Reallocate	`OnHeapAllocateandreallocate()` Allocate 100 bytes and then reallocate 200 bytes on the heap
Allocate and Free	`OnHeapAllocateanddelete()` Allocate 100 bytes on the heap and deallocate it
Create Heap, Allocate Memory, Destroy Heap	`OnCreateHeapAllocateDestroy()` Creates a heap, makes three allocations and destroys the heap. Allocations should not show as a leak.

Allocation menu : Local

Allocation menu › **Local** submenu › **...**

Tests Win32 local heap functions `LocalAlloc()`, `LocalRealloc()` and `LocalFree()`.

Allocate Memory	`OnLocalAllocatememory()` Allocate 4096 bytes using `LocalAlloc()`.
Allocate and Reallocate	`OnLocalAllocateandreallocate()` Allocates 4096 bytes and reallocates to 8192 bytes using `LocalRealloc()`.
Allocate and Free	`OnLocalAllocateanddelete()` Allocates 4096 bytes and deallocates with `LocalFree()`.
Other functions that call LocalAlloc()	`OnLocalAllocFunctions()` Allocate and leak memory using indirect calls to `LocalAlloc()`, eg `FormatMessage()`.

Allocation menu : Global

Allocation menu › **Global** submenu › **...**

Tests Win32 global heap functions `GlobalAlloc()`, `GlobalRealloc()` and `GlobalFree()`.

Allocate Memory `OnGlobalAllocatememory()`
Allocate 4096 bytes using `GlobalAlloc()`.

Allocate and Reallocate `OnGlobalAllocateandreallocate()`
Allocates 4096 bytes and reallocates to 8192 bytes using `GlobalRealloc()`.

Allocate and Free `OnGlobalAllocateanddelete()`
Allocates 4096 bytes and deallocates with `GlobalFree()`.

Allocation menu : Virtual

Allocation menu > **Virtual** submenu > **...**

Tests Win32 virtual memory allocation functions `VirtualAlloc()`, and `VirtualFree()`.

Allocate `OnAllocationVirtualAllocate()`
Commit virtual memory using `VirtualAlloc()`.

Free `OnAllocationVirtualFree()`
Decommit virtual memory using `VirtualFree()`.

Allocation menu : VirtualEx

Allocation menu > **Virtual** submenu > **...**

Tests Win32 virtual memory allocation functions `VirtualAllocEx()`, and `VirtualFreeEx()`.

Allocate `OnAllocationVirtualexAllocate()`
Commit virtual memory using `VirtualAllocEx()`.

Free `OnAllocationVirtualexFree()`
Decommit virtual memory using `VirtualFreeEx()`.

Allocation menu : Misc allocations

Misc allocations `OnMiscAllocations()`
Tests various functions including environment strings and array usage.
Some allocations are leaked.

Allocation menu : NetApi allocations

NetApi allocations `OnNetApiAllocations()`
Tests various functions including api buffer use.
Some allocations are leaked.

Allocation menu : Printers

Printers `OnTestPrinters()`
Opens and closes a printer and uses the printer notification api. Some allocations are leaked.

Allocation menu : COM (CoTaskMemAlloc)

Allocation menu › **Local** submenu › **...**

Tests Win32 COM functions `CoTaskMemAlloc()`, `CoTaskMemRealloc()` and `CoTaskMemFree()`.

› Allocate Memory `OnCotaskmemAllocatememory()`
Allocate 4000 bytes using `CoTaskMemAlloc()`.

› Allocate and Reallocate `OnCotaskmemAllocateandreallocate()`
Allocates 4000 bytes and reallocates to 50000 bytes using `CoTaskMemRealloc()`.

› Allocate and Reallocate and Free `OnCotaskmemAllocateandreallocateandfree()`
Allocates 4000 bytes, reallocates to 50000 and deallocates with `CoTaskMemFree()`.

› Allocate and Free `OnCotaskmemAllocateanddelete()`
Allocate 4000 bytes, and deallocates with `CoTaskMemFree()`.

Allocation menu : IMalloc (IMallocSpy)

Allocation menu › **IMalloc (IMallocSpy)** › **...**

Tests the Win32 OLE memory allocation spy functions via IMallocSpy.

All these functions use `CoGetMalloc()` to get the OLE allocator.

› Allocate Memory `OnImallocAllocatememory()`
Allocate 4000 bytes using the OLE allocator.

› Allocate and Reallocate `OnImallocAllocateandreallocate()`
Allocates 4000 bytes and reallocates to 50000 bytes.

› Allocate and Reallocate and Free `OnImallocAllocateandreallocateandfree()`
Allocates 4000 bytes, reallocates to 50000 and then deallocates it.

› Allocate and Free `OnImallocAllocateanddelete()`
Allocate 4000 bytes, and deallocates it.

Allocation menu : SysAllocString

SysAllocString	`OnAllocationSysallocstring()` Tests String use with `SysAllocString()`, `SysReAllocString()`, `SysFreeString()`, etc. Some allocations are leaked.

Allocation menu : CComVariant Test

CComVariant Test	`OnAllocationCcomvarianttest()` Tests BSTR allocations use with `Variant*()` functions. Some allocations are leaked.

Allocation menu : CoTaskMemFree hooks

CoTaskMemFree hooks (ole32)	`OnTestCoTaskMemFreeHooks()` Tests various functions where memory is freed with CoTaskMemFree.

Allocation menu : SysAllocString special test

SysAllocString special test	`OnAllocationSysallocstringspecialtest()` Tests multiple allocations of zero length BSTRs. Allocations are leaked.

Allocation menu : AddRef/Release Tests

AddRef / Release Test	`OnAllocationAddrefreleasetest()` Tests reference counting using `AddRef()` and `Release()` Allocations are leaked.
AddRef / Release Test (with zero refcount error)	`OnAllocationAddrefreleasetestzerocount()` As above but with additional attempt to `Release()` at a zero refcount. This may bring up the assertion dialog in the Debug versions

Allocation menu : Custom Allocator

Allocation menu › **Custom Allocator** submenu › **...**

Demonstrates use of a custom heap via the API for C++ and C.

These tests require a version of the example application linked to the Memory Validator Stub.

The Debug or Release Non Link configurations will disable these menu options.

Allocate Memory	`OnCustomAllocatememory()` Allocates memory using `mvUserCustomAlloc()`
Allocate and Reallocate	`OnCustomAllocateandreallocate()` Allocates and then reallocates using `mvUserCustomReAlloc()`
Allocate and Reallocate and Free	`OnCustomAllocateandreallocateandfree()` Allocates, reallocates and frees using `mvUserCustomFree()`
Allocate and Free	`OnCustomAllocateanddelete()` Allocates and frees memory
Increase Reference Count	`OnAllocationCustomallocatorIncreasereferencecount()` Demonstrates use of `mvUserCustomRefCountIncrement()`
Decrease Reference Count	`OnAllocationCustomallocatorDecreasereferencecount()` Demonstrates use of `mvUserCustomRefCountDecrement()`
mvCollect(TRUE)	`OnMvCollectTRUE()` Demonstrates use of `mvSetCollect()`
mvCollect(FALSE)	`OnMvCollectFALSE()` Demonstrates use of **`mvSetCollect()`**

Allocation menu : #define'd Allocator

#define'd Allocator	`OnAllocationDefinedallocator()` Demonstrates use of `#define` to define a wrapper macro around a standard memory allocation function. The Coverage tab is able to handle such cases.

Allocation menu : Test Many Hooks at once

Test Many Hooks at once	`OnTestManyHooks()` A convenience test that includes a selection of the tests above and more!

12.1.3 Memory Errors menu

The Memory Errors menu

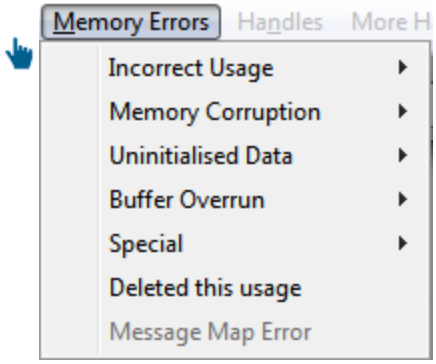
This section tests and demonstrates different types of memory error, including:

- mismatched reallocations and deallocations
- memory and stack corruption
- uninitialised data
- buffer underrun and overrun
- other incorrect usage that can be detected

Each test is listed below with the source code location in the example application.

➜ How to use these examples

Click on the picture below to jump to the relevant sections:



Memory Errors menu : Incorrect Usage

Memory Errors menu › **Incorrect Usage** submenu › **...**

Demonstrates detection of incorrect methods for reallocating or deallocating allocated memory using C Runtime Heap functions.

Includes memory that has not been allocated, or pointers near to pointers that have been allocated.

Double delete of memory	`OnTestDoubledeleteofmemory()` Allocate a character array using `new()`, and deletes the array twice
Double free() of memory	`OnTestDoublefreeofmemory()` Allocates a character array using `malloc()`, and frees the array twice
Incorrect free() of memory	`OnTestIncorrectfreeofmemory()` Allocates an array with `malloc()`, then frees pointers that point into the array, not at the start of it.
Incorrect delete of memory	`OnTestIncorrectdeleteofmemory()` Delete a pointer and an array pointer that is not in the heap. Allocates an array, and tries to delete pointers that point into the array, but not at the start. Deletes the array using the incorrect object form of delete. Allocates an array, and deletes it using the correct array form of delete. Allocates an array, and deletes it incorrectly with object form of delete. Allocates an object and deletes it using the wrong array form of delete.
Incorrect realloc() of memory	`OnTestIncorrectreallocofmemory()` Tries to reallocate a pointer that is not in the heap. Allocates using `new`, then tries to reallocate using `realloc()`, before deleting. Allocates using `malloc()`, then reallocates using pointers before and after the correct memory location. Reallocates the memory using `realloc()`, then frees the memory.
Delete malloc'd memory	`OnMemoryerrorsIncorrectusageDeletemallocdmemory()` Allocates with `malloc()`, deallocates with `delete`, then `delete []`, then `free()`. Allocates memory with `calloc()`, deallocate with `delete`, then `delete []`, then `free()`.
Delete realloc'd memory	`OnMemoryerrorsIncorrectusageDeletereallocdmemory()` Allocates with `malloc()`, reallocates using `realloc()`, deallocates with `delete`, then `delete []`, then `free()`.
Free new'd memory	`OnMemoryerrorsIncorrectusageFreenewdmemory()` Allocates using `new`, deallocates the memory using `free()`, then deallocates using `delete []`.
realloc() new'd memory	`OnMemoryerrorsIncorrectusageReallocnewdmemory()` Allocates using `new`, reallocates using `realloc()`, then deallocates using `delete []`.
Incorrect HeapRealloc()	`OnMemoryerrorsIncorrectusageIncorrectheaprealloc()` Allocates 4096 bytes on the heap and tries to reallocate wrong locations inside and outside the allocated area
Incorrect HeapFree()	`OnMemoryerrorsIncorrectusageIncorrectheapfree()` Allocates 4096 bytes and tries to free wrong locations inside and outside the allocated area

Memory Errors menu : Memory Corruption

Memory Errors menu › **Memory Corruption** submenu › **...**

Demonstrates corruption by overwriting the start and end of arrays allocated using C Runtime Heap functions.

Allocate memory and overwrite end of memory
`OnTestAllocatememoryandoverwriteendofmemory()`
Prompts for a memory size to allocate, then zeros 1 byte after the end of memory.

Allocate memory and overwrite beginning of memory
`OnTestAllocatememoryandoverwritebeginningofmemory()`
Prompts for a memory size to allocate, then zeros 1 byte before the start.

Memory Errors menu : Uninitialised Data

Memory Errors menu **Uninitialised Data** submenu **...**

Demonstrates detection of uninitialized data in C++ objects on both the stack and the C runtime heap.

Objects on Stack
`OnUninitialiseddataObjectsonstack()`
Creates a `TestClass` object on the stack.
Some of its constructors deliberately leave data members uninitialized so can be tracked.

Objects on Heap
`OnUninitialiseddataObjectsonheap()`
Creates a `TestClass` object on the *heap*, then deletes it.

Memory Errors menu : Buffer Overrun

Memory Errors menu **Buffer Overrun** submenu **...**

Demonstrates buffer overrun and buffer underrun detection on the C Runtime heap and the stack.

Underruning the stack is usually benign, but overrunning it usually results in a program crash as the function call return address has usually been damaged.

Overrun allocated memory	`OnBufferoverrunOverun()` Allocates 100 bytes using `new`, and uses `memcpy()` to copy memory, deliberately overruning the end.
Underrun allocated memory	`OnBufferoverrunUnderrun()` As above but underruning the start
Overrun stack memory	`OnBufferoverrunOverrunstackmemory()` Declares a 100 byte array on the stack, and uses `memcpy()` to copy, deliberately overruning the end of the array
Underrun stack memory	`OnBufferoverrunUnderrunstackmemory()` As above but underruning the start of the array
strcpy and wcscpy test	`OnMemoryerrorsBufferoverrunStrcpyandwcscpytests()` Allocate some character arrays and use these functions to overwrite beginning end of the buffers

Memory Errors menu : Special

Memory Errors menu **Special** submenu **...**

Demonstrates some of Memory Validator's heap watching functions.

Speculative Leak Test1	`OnLeaksSpeculativeleaktest1()` Allocates an array of 10 pointers and a `CString` object for each one. Deletes the array, but not the pointers. If speculative leak detection is enabled, the leaked objects will be identified.
Speculative Leak Test2	`OnLeaksSpeculativeleaktest2()` Allocates an array of 4 pointers populated with a `CWnd`, `CBrush`, `CPen` and `CWnd` objects. DeleteS the array, but not the pointers. If speculative leak detection is enabled, the leaked objects will be identified.
Allocate 0 bytes	`OnLeaksAllocate0bytes()` Allocates an array of 0 characters using new
Allocate -10 bytes	`OnLeaksAllocate10bytes()` Allocates an array of -10 characters using new
Allocate and don't use memory before deleting memory	`OnLeaksAllocateanddontusememorybeforedeletingmemory()` Allocates memory using new, then deletes it without using it. If the detect unused memory feature is enabled, the unused memory will be reported.
Allocate a class that allocates data and leaks	`OnMemoryerrorsSpecialAllocateaclassthatallocatesdataandlea` An `allocateInsideThis` object is allocated. Functions are called that allocate but don't free data held by the object, and then it's deleted. If speculative leak detection is enabled, leaked objects will be identified.
Deeply recursive call to test hotspots	`OnMemoryerrorsSpecialDeeplyrecursivecalltotestcallstacks()` Allocates and leaks an array of 50 bytes on the way out of each level of a 1500 level recursive call.

Memory Errors menu : Deleted this usage

Deleted this usage	`OnMemoryerrorsDeletedthisusage()` Allocates a class, deletes it and then calls a method in the deleted class. If detection of calling functions for deleted objects is enabled, this will be identified.

Memory Errors menu : Message Map Error

This test is only enabled in Visual Studio versions prior to 7.0 as the code will not compile in that version or above.

Message Map Error	`OnMemoryerrorsMessagemaperror()` Calls this function with too many parameters despite a signature with no params. If detection of Broken Message Map hooks is enabled, this will be identified in Debug and Release builds.

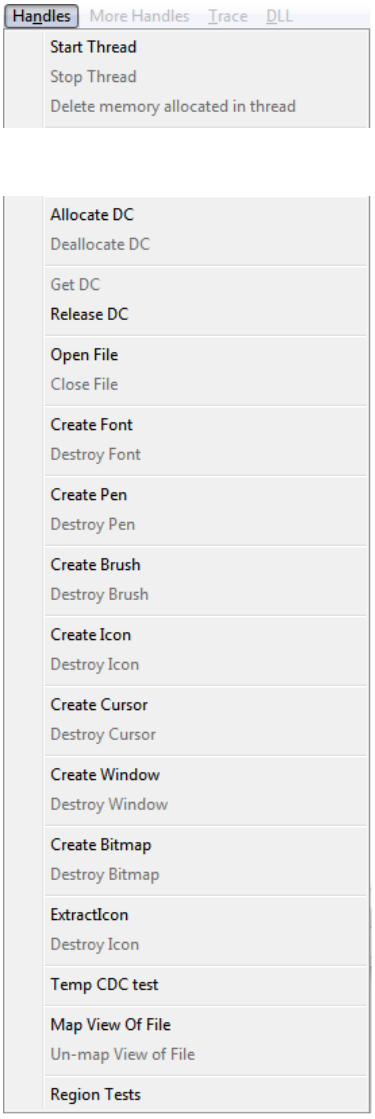
12.1.4 Handles and More Handles menus

The Handles menu

The Handles menu is a convenient way to allocate and free Win32 resource handles and track the events in Memory Validator.

Leaks are created by creating but not freeing each type of resource below.

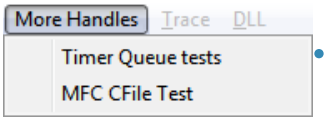
The thread creation and thread deletion examples can test the detection of cross thread allocation.



- Start a thread and allocate some memory in it to be deallocated below
- Stop the thread
- The deallocation will happen from a *different* thread to the one that allocated.
- Allocate a device context
- Deallocate it
- Get a device context using `GetDC()`
- Release it using `ReleaseDC()`
- Open a file. If the file is not closed the file handle will be leaked.
- Close it
- Create a font
- Destroy it
- Create a pen
- Destroy it
- Create a brush
- Destroy it
- Create an icon
- Destroy it
- Create a mouse cursor
- Destroy it
- Create a window
- Destroy it
- Create a bitmap
- Destroy it
- Extract an icon from a bitmap
- Destroy the icon
- Test the lifetime of a temporary CDC object
- Map a view of a file into memory
- Unmap the view
- Create some regions using `CreateRectRgn()`, one of which leaks
- See `CTeststakView::OnRegionTests()`

The More Handles menu

The More Handles menu is...well, more of the same - i.e. convenient ways to create different types of handles and oberve the events in Memory Validator.



- Create some Timer Queue and Timer handles - see `CTeststakView::OnTimerQueueTests()`
- Create and MFC `CFile` object, write to it and then close it.

12.1.5 Trace menu

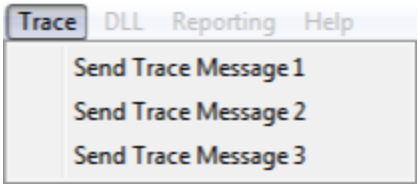
The Trace menu

There are three similar options - each of which outputs a message to the debugger (and Memory Validator) using the `TRACE()` macro.

To observe these events in Memory Validator, enable collection of TRACE messages.

The messages are of the form `"Trace Message 1/2/3";`

➜ See also examples of trace message monitoring.

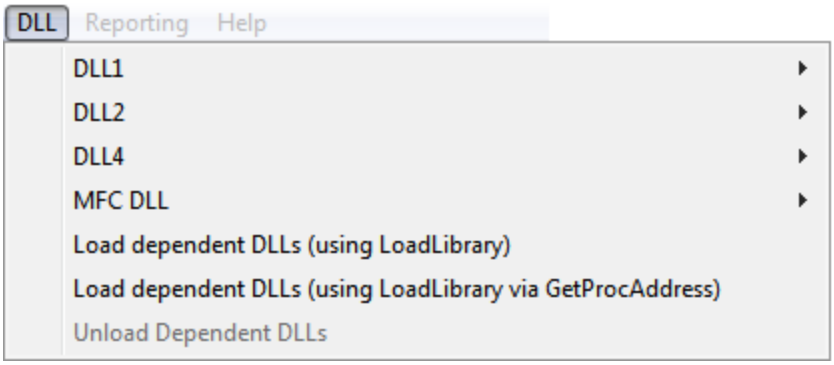


12.1.6 DLL menu

The DLL menu

The DLL menu demonstrates the capability of Memory Validator to detect dynamic loading and unloading of DLLs via one of the following Win32 calls:

- `LoadLibrary()`
- `LoadLibraryEx()`
- `FreeLibrary()`



DLL1, DLL2, DLL4, and MFC DLL

These four DLLs can each be loaded and unloaded and each has a test function in it, for example:

DLL1 submenu › **...**

- **Load DLL** › load the DLL
- **Unload DLL** › unload the DLL
- **Test DLL** › call the supplied test function in the DLL

  The test function performs some memory allocation functions, which may result in memory leaks.

See the `CTeststakView` class methods: `OnDllLoaddll1(), OnDllUnloaddll1(), OnDllTestdll1(), etc`

Dependent DLLS

The next three options load and unload DLLs having dependent DLLs which should also be loaded:

- **Load Dependent DLLs (using LoadLibrary)** › loads dllADependentOnB.dll

  This is dependent on dllBDependentOnC.dll which, in turn, is dependent on dllC.dll
- **Load Dependent DLLs (using LoadLibrary via GetProcAddress)** › Similar to the above, but different methods
- **Unload Dependent DLLs** › frees the library if loaded

See the `CTeststakView` class methods: `OnDllLoaddependentdlls(), OnDllLoaddependentdlls_LLGPA() and OnDllUnloaddependentdlls()`.

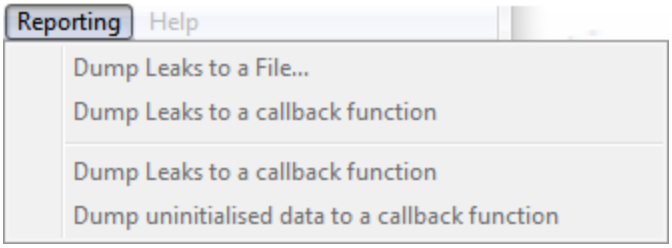
12.1.7 Reporting menu

The Reporting menu

The Reporting menu provides access to various reporting capabilities that are present in the Memory Validator API:

These tests require a version of the example application linked to the Memory Validator Stub.

The Debug or Release Non Link configurations will disable these menu options.



Reporting menu › **...**

› **Dump Leaks to a File...**	`OnMemoryerrorsDumpleakstoafile()` Dumps leaks to a file use the API function `mvUserDumpLeaks`
› **Dump Leaks to a callback function...**	`OnMemoryerrorsDumpleakstoacallbackfunction()` Dump leaks to a callback function using `mvUserDumpLeaks`
› **Dump Leaks to a callback function...**	`OnMemoryerrorsDumpleakstoacallbackfunction2()` Dump leaks to a callback function using `mvLeakDetect`
› **Dump uninitialised data to a callback function...**	`OnMemoryerrorsDumpuninitialiseddatatoacallbackfunction()` Dump uninitialised data to a callback function using `mvDetectUninitialised`

12.1.8 Help menu

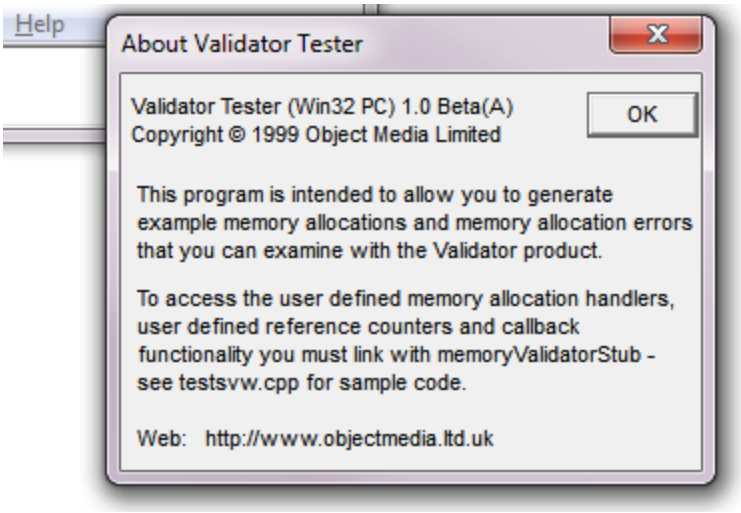
The Help menu

The Help menu only has one option that displays an 'about' box.

However, this is also a test, as when the about box is displayed, memory is allocated but not freed.

Help menu › **About nativeExample...** › shows the about box below

The about box leaks each time it's displayed and when the program is closed, Memory Validator detects the leaks.



See `CTeststakApp::OnAppAbout()` in nativeExample.cpp

12.2 Finding memory leaks

Detecting memory leaks

This test (and most of those that follow) uses the example application **nativeExample.exe**.

The example program is run once and we use the Memory view to observe and investigate the memory leak.

Memory Validator

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

File menu › **Exit**

The example program automatically generates some memory leaks when it starts.

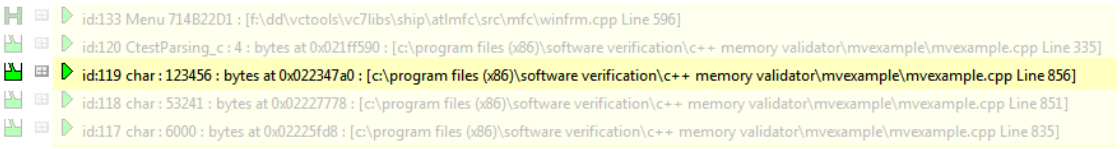
Memory Validator

wait for data transfer to complete

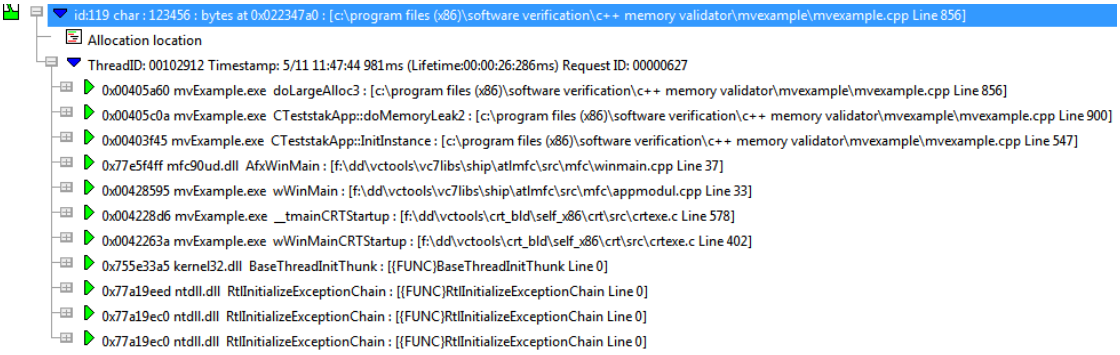
The data collection indicators will be disabled when collection has stopped and all data for the session has been processed.

- **Memory** tab › **Refresh** › shows all leaked memory in the colour defined for leaks - yellow by default

One of the entries is an allocation of 123456 bytes:

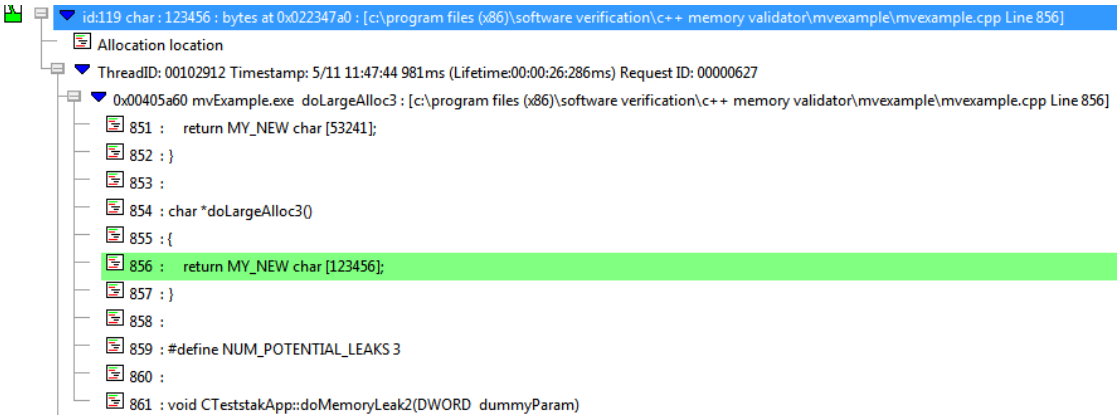


- **expand** the entry › shows the allocation callstack



- **expand** the topmost entry on the callstack › the source code fragment shows where the memory was allocated

  If there is no source code, or it can't be found, you'll be prompted for the location.



  To edit the source code, double click on the source code.

12.3 Finding handle leaks

Detecting handle leaks

The example program is run once and we use the Memory view to observe and investigate the handle leak.

Memory Validator

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

Handles menu › **Start Thread** › **File** menu › **Exit**

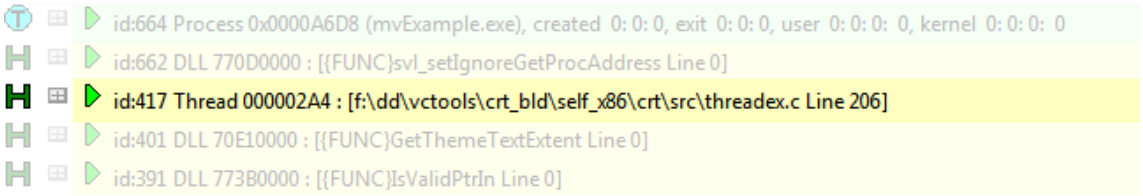
Memory Validator

wait for data transfer to complete

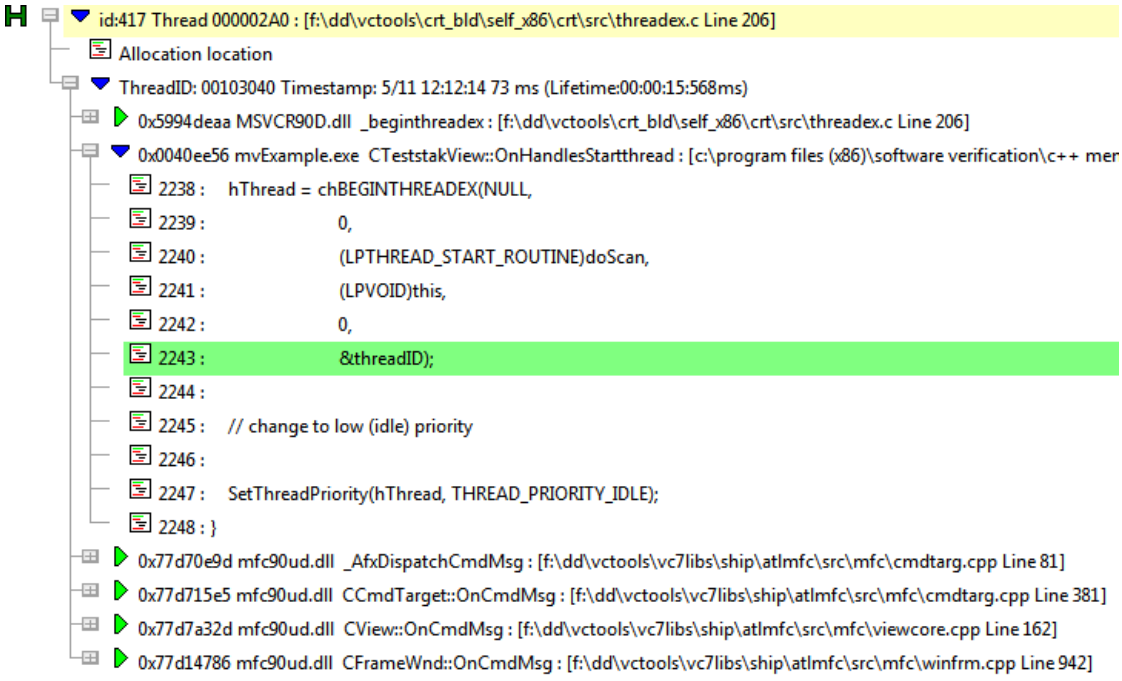
- **Memory** tab › **Refresh** › shows all leaked memory and any handles that have not been deallocated

  Common handle leaks might come from forgetting to release a device context (DC), forgetting to reset brushes, pens, and other graphical items, file, DLL or thread handles.

  One of the leaks is a thread handle:



- **expand** the entry › **expand** the `OnHandlesStartthread` entry on the callstack › the source shows where the handle was allocated in the example application



  The top entry in the callstack shows the `_beginthreadex()` call in `threadex.c` (MSVCR90D.DLL) where `CreateThread()` would be called.

Hey, some leaks aren't leaks!

Handles are sometimes shown that aren't really leaked, but where deallocation hasn't been 'seen' by Memory Validator.

This is typically because the deallocations are done by kernel internals and don't pass via Memory Validator's hooking mechanism.

Other handles might be leaked by errors in 3rd party code.

As a result, handles such as bitmaps and icons sometimes show up in the display.

For your program, you'll need to decide which handle leaks are important by examining callstacks and using filters to remove unwanted handles from the display.

12.4 Finding uninitialised memory

Detecting uninitialised memory

The example program is run once and we use the Memory and Analysis view to observe and investigate any uninitialised memory.

Memory Validator

Settings menu › **Edit Settings...** › **Data Collection** › **Collect** page › check **Uninitialised Data** › enables the use of **Uninitialized Data** hooks

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

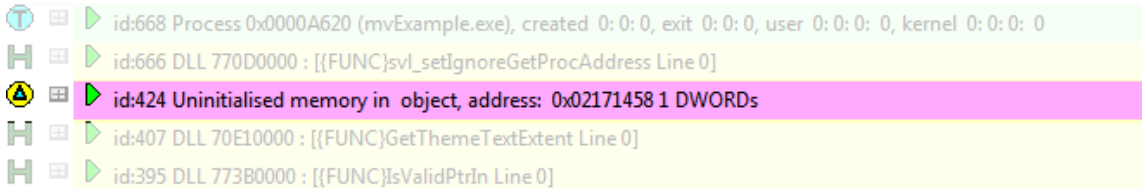
Memory Errors menu › **Uninitialized Data** › **Objects on Heap** › Allocates and frees an object with uninitialized data members

File menu › **Exit**

Memory Validator

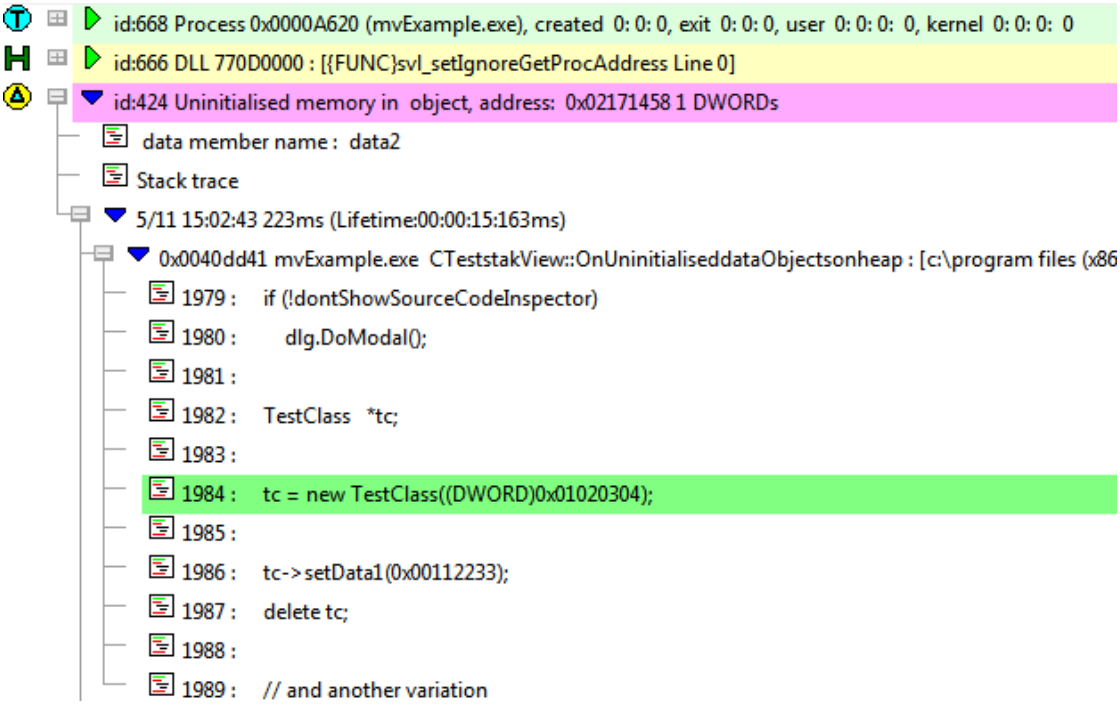
wait for data transfer to complete

- **Memory** tab › **Refresh** › shows the usual leaks and also uninitialised data use in the colour defined.



- **expand** the most recent uninitialised item › **expand** the `OnUninitialiseddataObjectsonheap` entry on the callstack › shows the source for the allocation

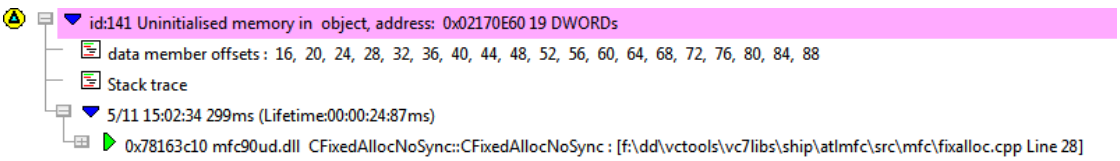
  The information shows that in the allocation of a `TestClass` object, the data member `data2` was not initialised:



  The callstack may not always be present for all uninitialized data.

  The display should show another uninitialised data item earlier in the event history.

  This doesn't show a full callstack or data member names, but does show the size, address and relative offsets of the uninitialised data

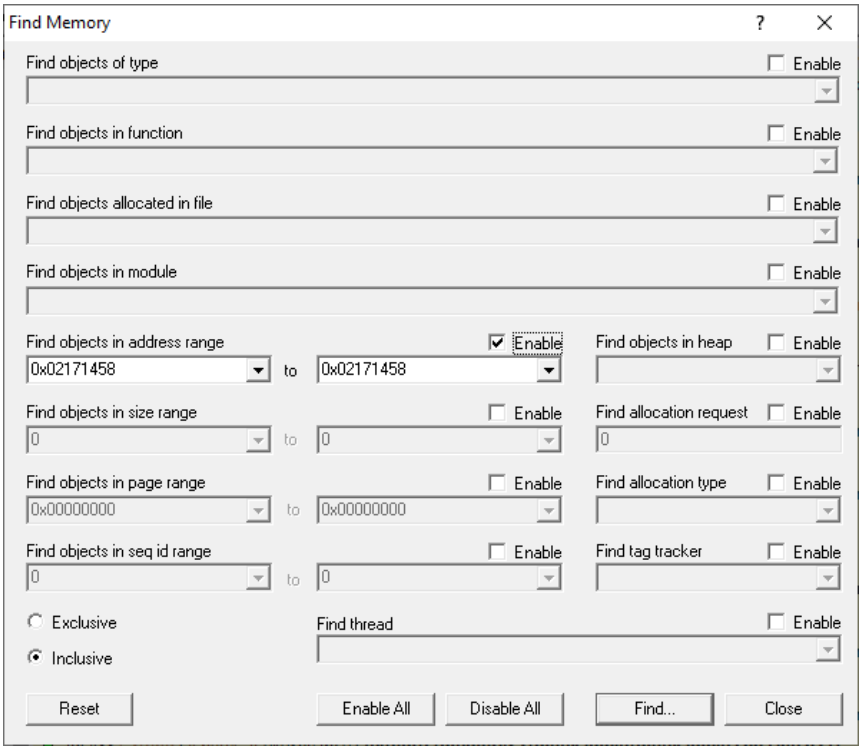


No callstack? Which object?

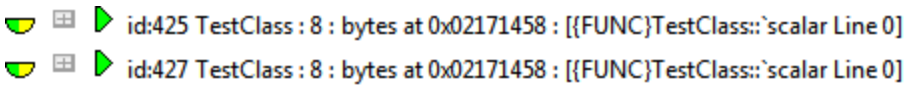
If your program has uninitialised data but the callstack isn't available:

- Make a note of the object address - for example in our first example above `0x02171458`
- **Analysis** tab › **Memory...** › shows the Find Memory in Analysis dialog
- **Disable All** › enable **Find objects in address range** › enter **address range** (e.g. `0x02171458` to `0x02171458`) › **Find...** › shows matching objects

  You can also set the range to search for object *containing* that address or the allocation address plus the offset size.

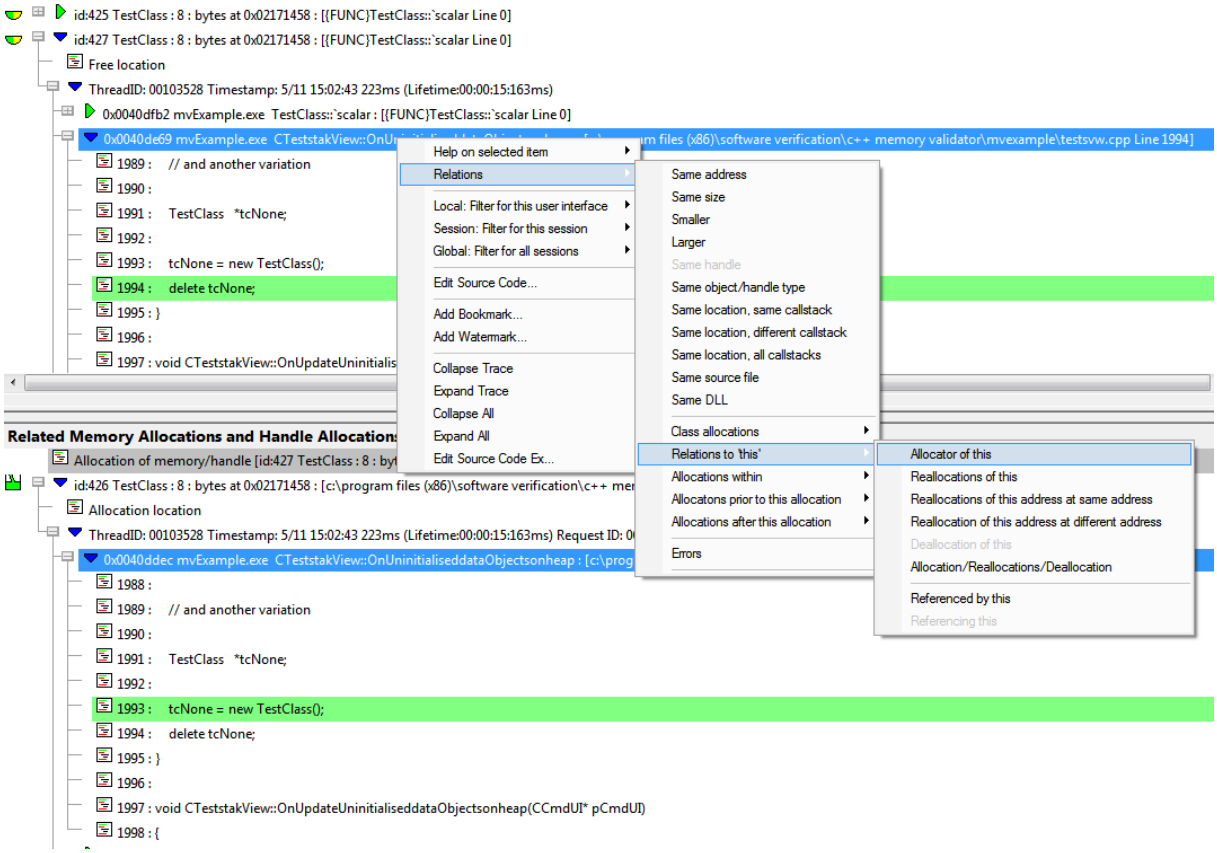


The results of the search (if any) are highlighted on the display, showing a `TestClass` object of size 8 bytes at that address:



- select the *later* of the two results › **Relations** › **Relations to 'this'** › **Allocator of this** › shows the allocation event for this object in the Relations pane

  Expanding these traces shows the callstacks for the deallocation and the allocation of the object that has the uninitialized data.



12.5 Finding double deallocations

Detecting double deallocations

The example program is run once and we use the Memory view to observe and investigate any double deallocations.

For each double deallocation, Memory Validator displays the allocation and deallocation locations.

Memory Validator

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

Memory Errors menu › **Incorrect Usage** › **Double Delete** › forces a double deallocation in the example application

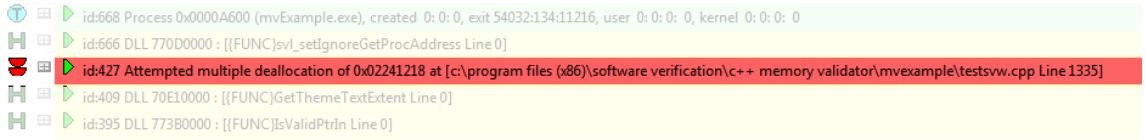
If a heap warning dialog appears, choose **Ignore** to continue.

File menu › **Exit**

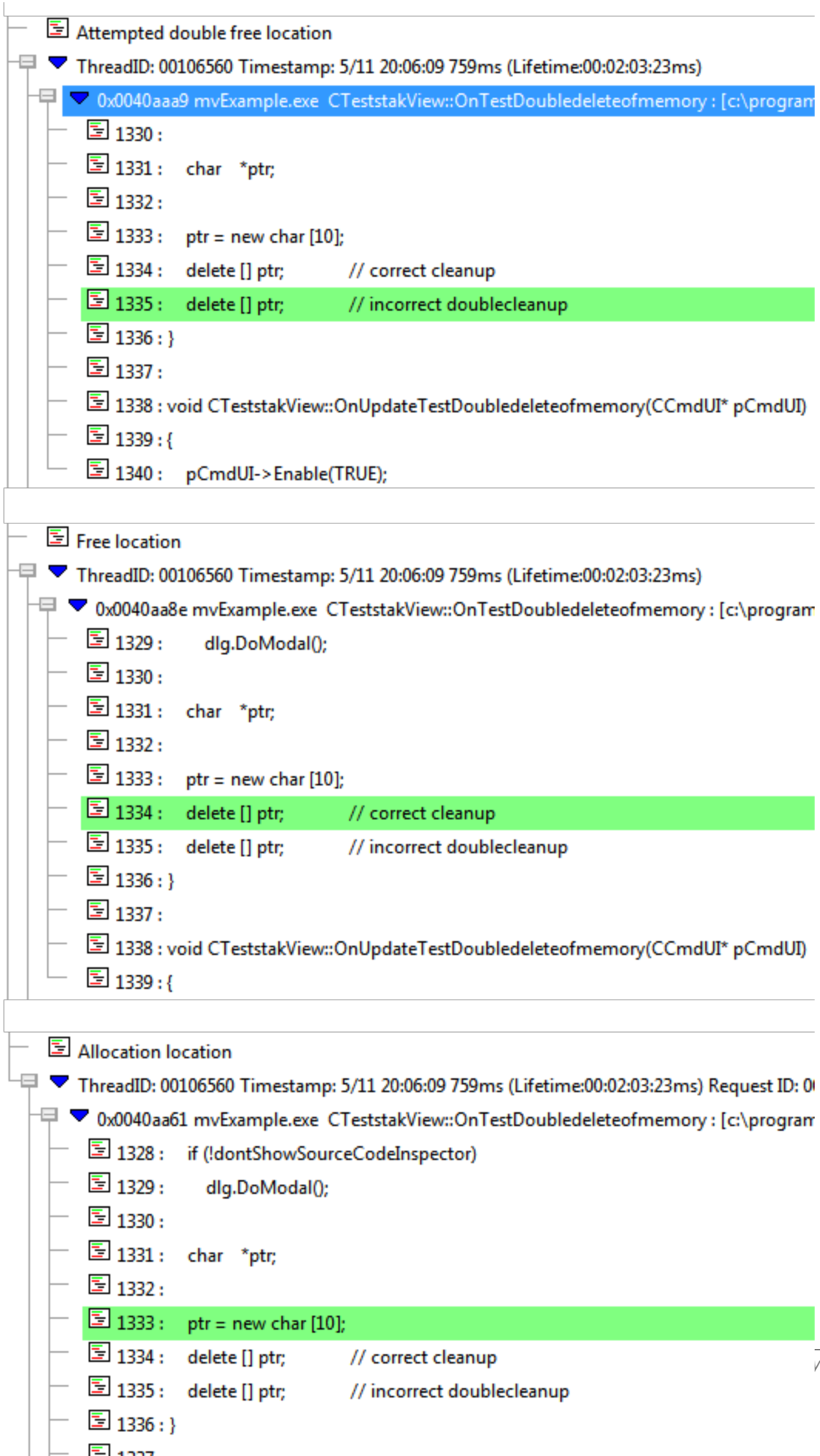
Memory Validator

wait for data transfer to complete

- **Memory** tab › **Refresh** › shows the usual leaks and also multiple deallocations using the colour defined.



- **expand** the most recent double deallocation › shows the locations for allocation, free and double free
- **expand** the topmost entries in each callstack › shows the successive lines of the test in `CTeststakView::OnTestDoubledeleteofmemory()`


Attempted double free location
ThreadID: 00106560 Timestamp: 5/11 20:06:09 759ms (Lifetime:00:02:03:23ms)
0x0040aaa9 mvExample.exe CTeststakView::OnTestDoubledeleteofmemory : [c:\program
1330 :
1331 : char *ptr;
1332 :
1333 : ptr = new char [10];
1334 : delete [] ptr; // correct cleanup
1335 : delete [] ptr; // incorrect doublecleanup
1336 : }
1337 :
1338 : void CTeststakView::OnUpdateTestDoubledeleteofmemory(CCmdUI* pCmdUI)
1339 : {
1340 : pCmdUI->Enable(TRUE);
Free location
ThreadID: 00106560 Timestamp: 5/11 20:06:09 759ms (Lifetime:00:02:03:23ms)
0x0040aa8e mvExample.exe CTeststakView::OnTestDoubledeleteofmemory : [c:\program
1329 : dlg.DoModal();
1330 :
1331 : char *ptr;
1332 :
1333 : ptr = new char [10];
1334 : delete [] ptr; // correct cleanup
1335 : delete [] ptr; // incorrect doublecleanup
1336 : }
1337 :
1338 : void CTeststakView::OnUpdateTestDoubledeleteofmemory(CCmdUI* pCmdUI)
1339 : {
Allocation location
ThreadID: 00106560 Timestamp: 5/11 20:06:09 759ms (Lifetime:00:02:03:23ms) Request ID: 0
0x0040aa61 mvExample.exe CTeststakView::OnTestDoubledeleteofmemory : [c:\program
1328 : if (!dontShowSourceCodeInspector)
1329 : dlg.DoModal();
1330 :
1331 : char *ptr;
1332 :
1333 : ptr = new char [10];
1334 : delete [] ptr; // correct cleanup
1335 : delete [] ptr; // incorrect doublecleanup
1336 : }

12.6 Finding memory corruptions

Detecting double deallocations

The example program is run once and we use the Memory view to observe and investigate any double deallocations.

For each double deallocation, Memory Validator displays the allocation and deallocation locations.

Memory Validator

- **Memory** tab › **Display...** › check **Memory Errors** if not already checked.

Settings menu › **Edit Settings...** › **Data Collection** › **Collect** page › check **Memory Buffer detect** › enables the use of **Uninitialized Data** hooks

> The target program will run slower than usual with this option as all functions related to strings and memory copying are monitored.

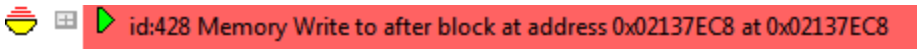
launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

Memory Errors menu › **Buffer overrun** submenu › **Overrun allocated memory** › allocates an array and the end of the array is deliberately overrun

Memory Validator

In the Memory tab, the overrun corruption is detected and displayed

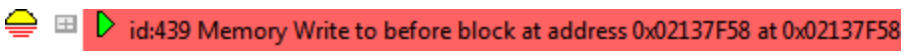


nativeExample.exe

Memory Errors menu › **Buffer overrun** submenu › **Underrun allocated memory** › allocates an array and the end of the array is deliberately underrun

Memory Validator

In the Memory tab, the underrun corruption is detected and displayed



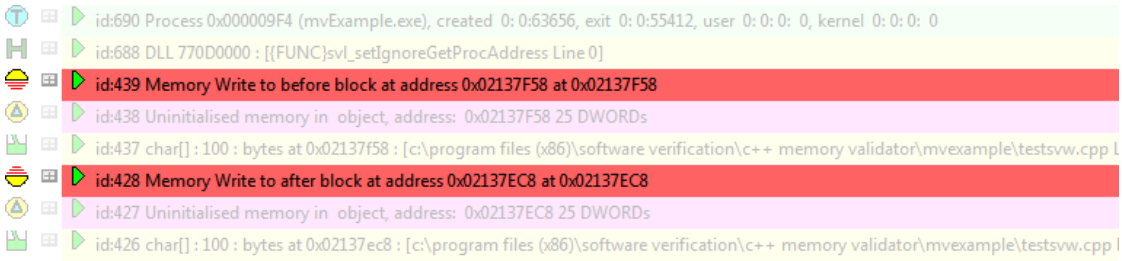
nativeExample.exe

File menu › **Exit**

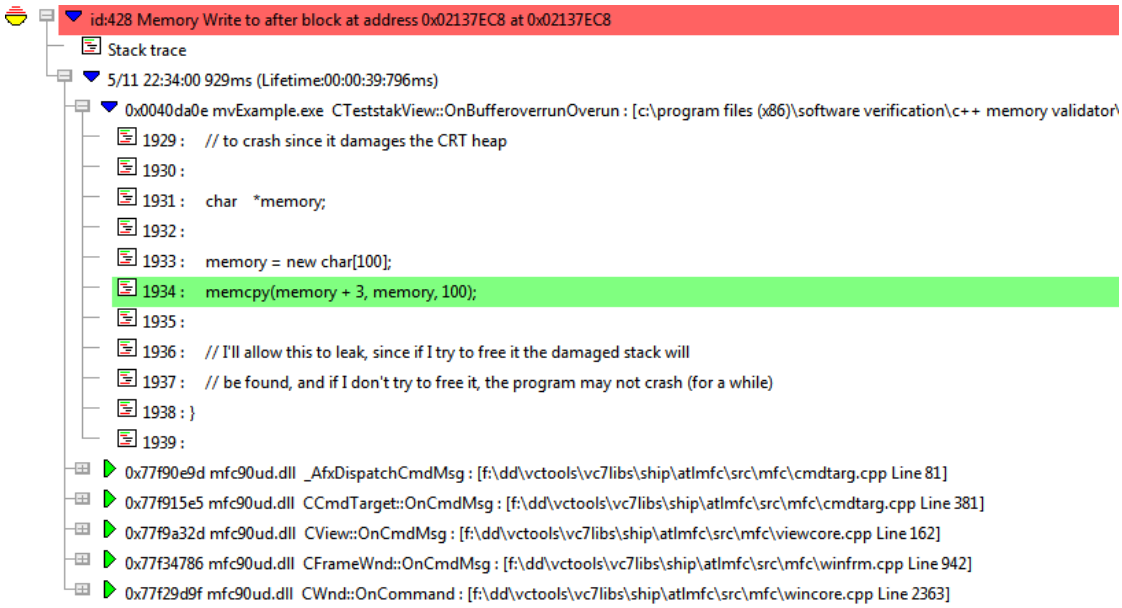
Memory Validator

wait for data transfer to complete

- **Memory** tab › **Refresh** › shows the usual leaks and also the two corrupted blocks using the colour defined.



- **expand** the most recent corruption › shows the callstack at the point of corruption
- **expand** the topmost entry in the callstack › shows the source at the corruption point in `CTeststakView::OnBufferoverrunOverun()`



Searching for Memory Allocations

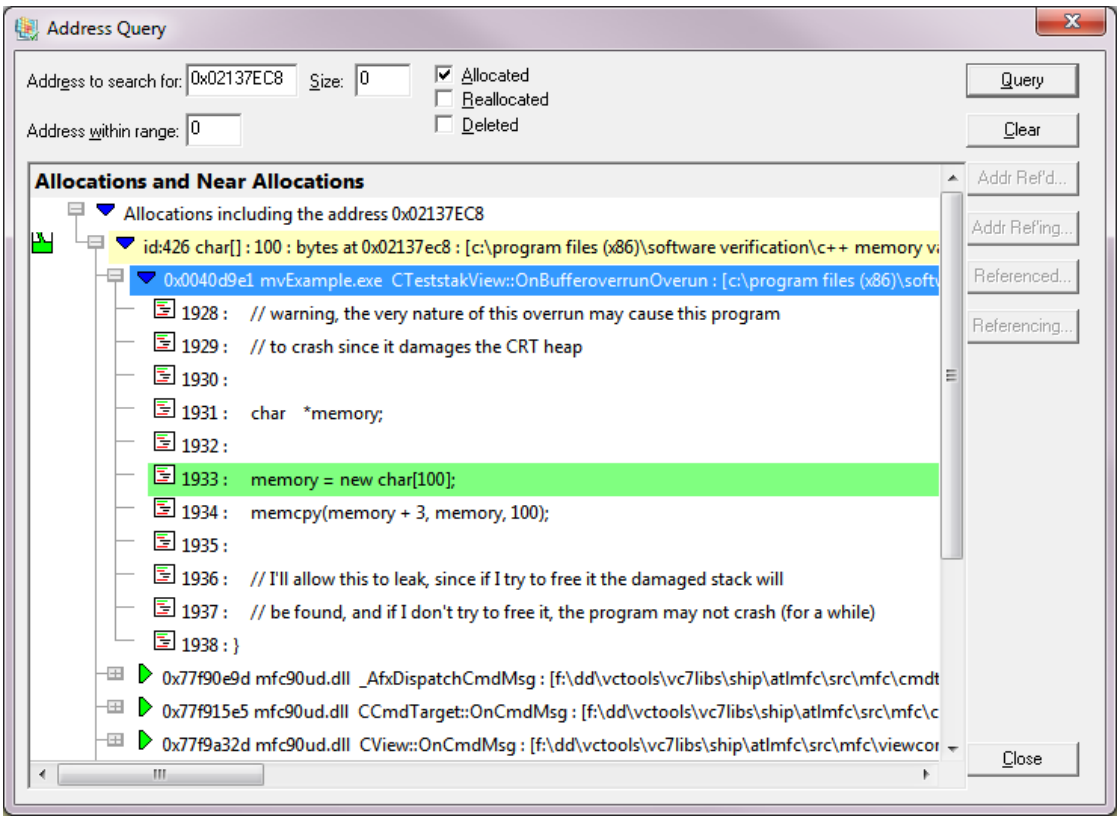
In the simple code of the example above we can see where the allocation is - on the line above the corruption point.

If we didn't know where the memory was being allocated, we can use the query address dialog.

- Make a note of the object address - e.g. in our example above `0x02137EC8`

Query menu › **Query Address...** › enter the address in **Address to search for** › **Query** › shows all allocations including that address.

This should show a result that can be expanded to show the allocation location as the line above the corruption point we saw above.



12.7 Finding crashes due to deleted objects

Access violations

Often crashes are caused by an Access Violation Exception.

This occurs when the program tries to access memory that is no longer valid, as in the following examples:

For each case we need to know the address that we are interested in querying.

- accessing memory that has been freed and released by the virtual memory manager

  This memory address is the one we are interested in.

- accessing memory through a bad pointer held by an object that's since been deleted

  The address of the object or data structure that held that pointer is the one we are interested in.

- calling a function on an object that has been deleted

  The address of the object is the one we are interested in.

Note, this case won't crash if all the following are true:

- the function is not virtual
- the function accessed no data members of the object
- the function calls no functions on the object that are virtual

Analysis Window

Knowing the address, we can use the analysis view and the Find Memory dialog to search for the allocation, reallocation and deallocation locations of the memory.

In the case of that memory then being re-used in an allocation for another object we can find that the analysis window and spot cases of code accessing one object when in reality it is accessing a completely different object in a re-used memory space.

The process is identical to the way we found objects using an address in the example of finding uninitialised memory.

Searching for Memory Allocations

Knowing the address, we can also use the query address dialog to find memory allocations with the specified address.

A description of this is in the example of finding memory corruptions.

The query address dialog can find all memory locations that *contain* pointers to the memory allocations and all memory locations that are *pointed to* by pointers in the memory allocations.

12.8 Finding allocations and reallocations

Finding allocations and reallocations

You may wish to trace the history of a reallocated memory block.

The example program is used to examining where memory was allocated and reallocated.

This example is for the CRT, but the same techniques work for

- Win32 heaps
- GlobalAlloc() heaps
- LocalAlloc() heaps
- CoTaskMemAlloc()
- IMallocSpy
- user defined (custom allocators) heaps.

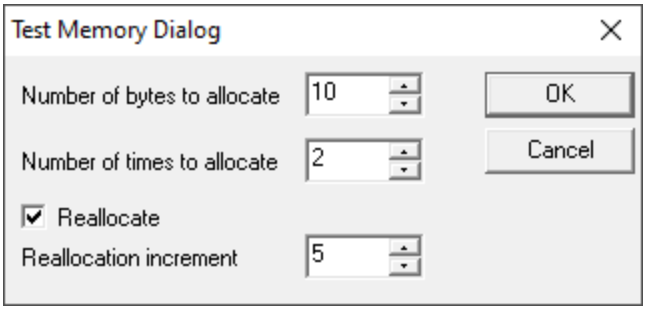
Memory Validator

- **Memory** tab › **Display...** › ensure **All Memory (leaks, errors, unleaked)** is selected in the first combo box

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

Allocations menu › **C Runtime (C/C++) Heap** › **Reallocate memory...** › shows the **Test Memory** dialog

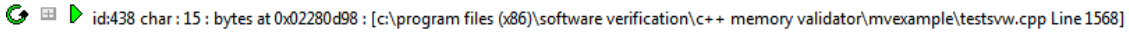


Set the following to allocate a 10 bytes block and reallocate it twice, growing in size by 5 bytes each time.

- **Number of bytes to allocate** › set to 10
- **Number of times to allocate** › set to 2
- **Reallocate** › check this
- **Reallocation increment** › set to 5 › **OK**

Memory Validator

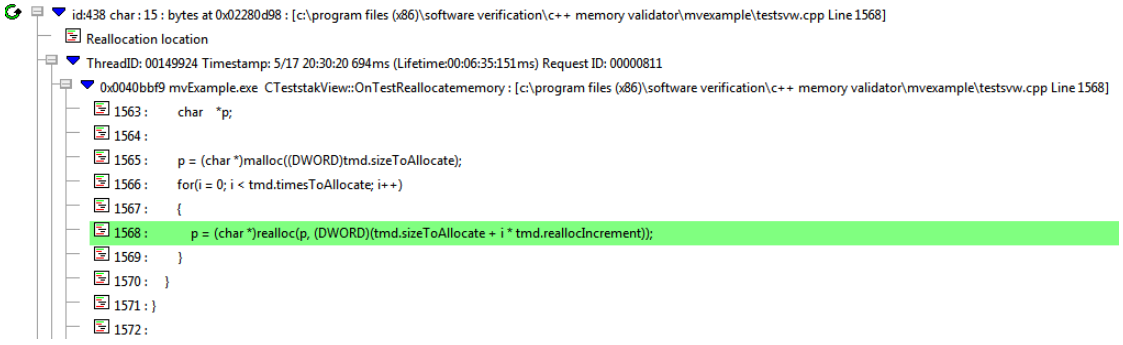
- **Memory** tab › **Refresh** › the display should refresh to include an item like the one below



- **expand** the item › shows the reallocation history - an allocation and two reallocations

- **expand** the topmost entry in the callstack › show the source code for the reallocation in `CTeststakView::OnTestReallocatememory()`

  Note that the two reallocation locations are the same (because they are in a loop).



mvExample.exe

File menu › **Exit**

12.9 Finding incorrect deallocations

Finding incorrect deallocations

This example demonstrates detection of incorrect deallocation of allocated memory.

Examples where this can happen include the following, all of which Memory Validator can detect

- an allocate with malloc() or calloc(), but then deallocated with delete or delete []
- a reallocate with realloc() or _expand(), but then deallocated with delete or delete []
- an allocate with new, but then deallocated with free()
- an allocate with new, but then reallocated with realloc() or _expand()

Memory Validator

- **Memory** tab › **Display...** › ensure **Memory Errors** is checked

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

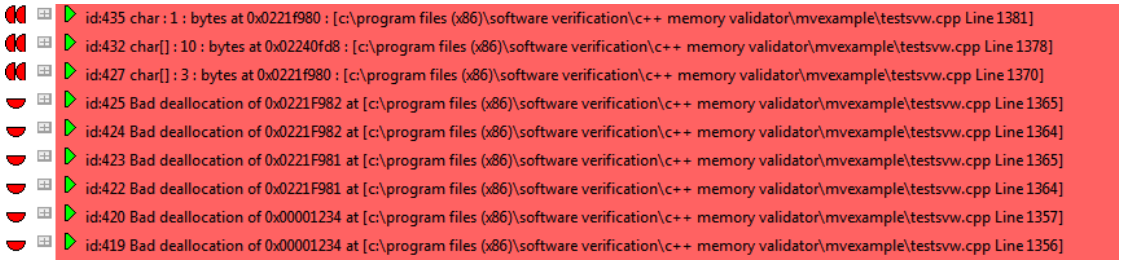
Memory Errors menu › **Incorrect Usage** › **Incorrect delete of memory** › incurs a selection of deliberately incorrect deletion calls

File menu › **Exit**

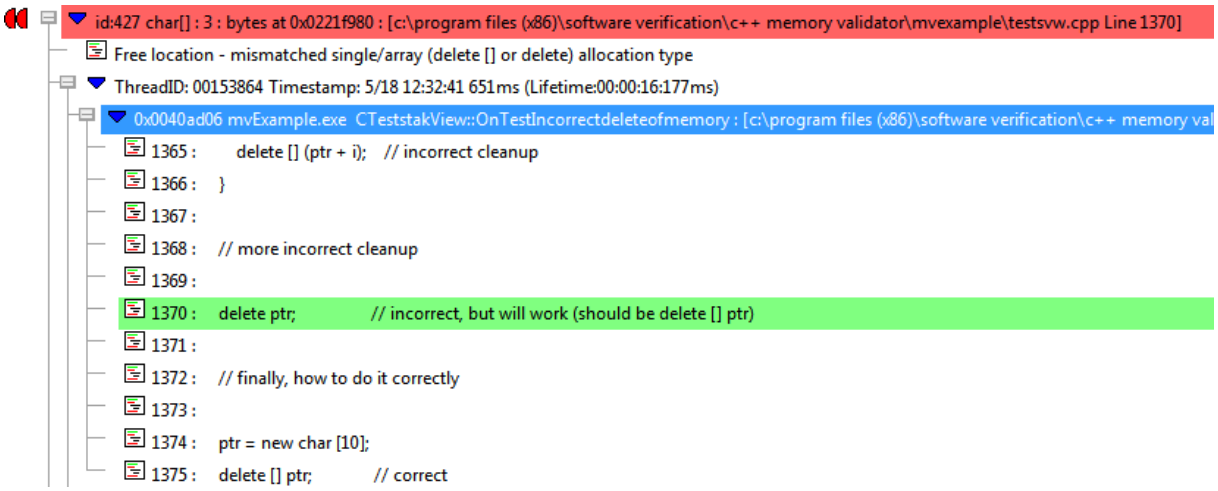
Memory Validator

wait for data transfer to complete

- **Memory** tab › **Refresh** › shows the usual leaks and also the memory errors using the colour defined.



- **expand** any of the items › shows the allocation and deallocation callstack locations
- **expand** the topmost entry in any of those callstacks › shows the source code in `CTeststakView::OnTestIncorrectdeleteofmemory()`



Using delete and delete [ ]

Note the difference between `delete` and `delete []` in the example above.

For simple types (char, int, etc) the difference is technical, rather than a memory leak.

For class objects, it's important and misuse can result in leaks. Undefined behaviour, including crashes, can result from using `delete []` instead of `delete`.

➡ There is an archive available (at the time of writing) of an interesting discussion on delete and delete [] in C++

Mistakes that can happen

Sometimes Memory Validator will appear to get a type wrong or get the array classification wrong leading to an incorrect warning about an incorrect deallocation. Why does this happen?

There are two reasons this can happen.

1. You have specified that you are building with a particular version of Visual Studio, but you are actually building with a different version of Visual Studio. This will mean that for all CRT and MFC source file parsing Memory Validator will be looking at the wrong source files and wrong header files.

   To fix this, change the version of Visual Studio in the Symbol Lookup settings to match the Visual Studio you are using.

2. The binary you are testing doesn't match the source code that is indicated by the debug information (or that is found by the source code locations on the File Locations settings).

   To fix this, check the File Locations settings to ensure no incorrect paths are listed. Also check that the source code indicated by the debug information is the correct source code for the build you are testing.

12.10 Reducing data in the display

Finding the signal in the noise

When you have a lot of data, it can be difficult to see the data that really matters - the signal within the noise.

This is especially the case when looking for a specific leak, rather than looking for *all* leaks.

There are a number of ways to reduce the amount of data:

- reduce the data collected
- reduce the data that is displayed
- filtering the data that is displayed
- use watermarks to display data between two points

Reducing collected data

Collected data can be controlled at different levels of granularity via the settings dialog.

Some settings pages can be used for fine tuning of how the data is collected and which hooks are enabled.

The collect tab allows quite coarse control over which types of hook are installed in the target program.

 **Settings** menu › **Edit Settings...** › **Data Collection** › **Collect** page

Reducing displayed data

Data that ends up being collected can be removed from the display.

Most of the main tabs have a **Display** button to control what's shown:

- Memory
- Types
- Sizes
- Locations
- Generations
- Ages
- Hotspots
- Analysis

Filtered Data

Filters operate by matching source file and line number, callstack and datatype to keep or remove data.

A combination of permanent and temporary filters can be used, as well as global, session and local filters.

Watermarks

Watermarks let you not only add markers where certain events happened, but also let you display only the data for allocation events that happened between those watermarks.

The example usage of watermarks demonstrates this at the end

Example of reducing data in the display

Here's a simple example of reducing displayed data via the display settings dialog:

Memory Validator

launch nativeExample.exe › **wait** until attaching is complete

mvExample.exe

File menu › **Exit**

Memory Validator

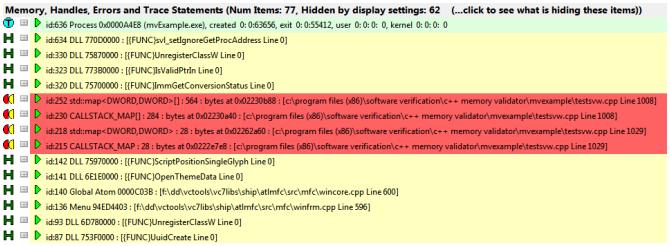
wait for data transfer to complete

- **Memory** tab › **Refresh** › shows the usual leaks and also memory errors

  Roughly half the data is shown here:

- **Memory** tab › **Display...** › select **Leaks and errors** in the first combo box › uncheck **CRT Memory** › OK
- **Memory** tab › **Refresh** › shows a much reduced set of data

  All the data displayed is shown here - a much reduced set of information!



Note the header helpfully shows the number of items hidden by the display settings, 62 compared with 2 before

The same approach could be done with other data, for example if you have allocations on other heaps:

- **Memory** tab › **Display...** › uncheck **Heap Memory** › uncheck **Global Alloc Memory** › uncheck **Local Alloc Memory** › OK › **Refresh**

Or to remove handle allocations from the display:

- **Memory** tab › **Display...** › uncheck **Handles** › OK › **Refresh**

When you enable or disable the **Memory Errors** check box in the Memory tag display settings dialog, you may also need to make a change to the first combo box to make the display change. This is because memory error traces are not considered memory leaks.

12.11 Session comparison

Session comparison

Session comparison can be used as part of manual regression testing to detect any differences between sessions including leaks, errors, or anything else.

A good practice might be to also use session comparison with automated regression tests to provide similar data on a daily basis.

This helps developers or quality assurance departments track issues on a continual basis rather than when just things get so bad that something needs to be done.

Example session comparison

To demonstrate a simple session comparison we'll run the example application **twice**.

In the second run we'll allocate more objects that are not deallocated than in the first run.

We'll use Memory Validator to compare the two sessions to show which leaks are common to both runs and which leaks were present only in the second run.

Memory Validator

Managers menu › **Session Manager** › set **Maximum number of sessions** to at least 2 › allows several sessions to be kept at the same time

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

File menu › **Exit**

Memory Validator

wait for data transfer to complete

This first run is the *baseline* session.

launch nativeExample.exe › **wait** until attaching is complete

nativeExample.exe

Allocations menu › **C Runtime (C/C++) Heap** › **Allocate memory...** › shows the **Test Memory** dialog › **OK** to accept the default values

File menu › **Exit**

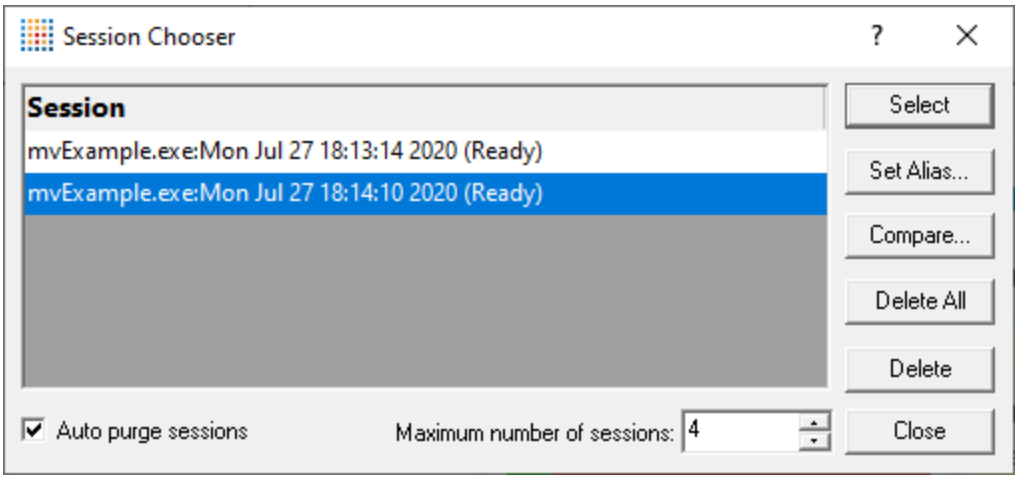
Memory Validator

wait for data transfer to complete

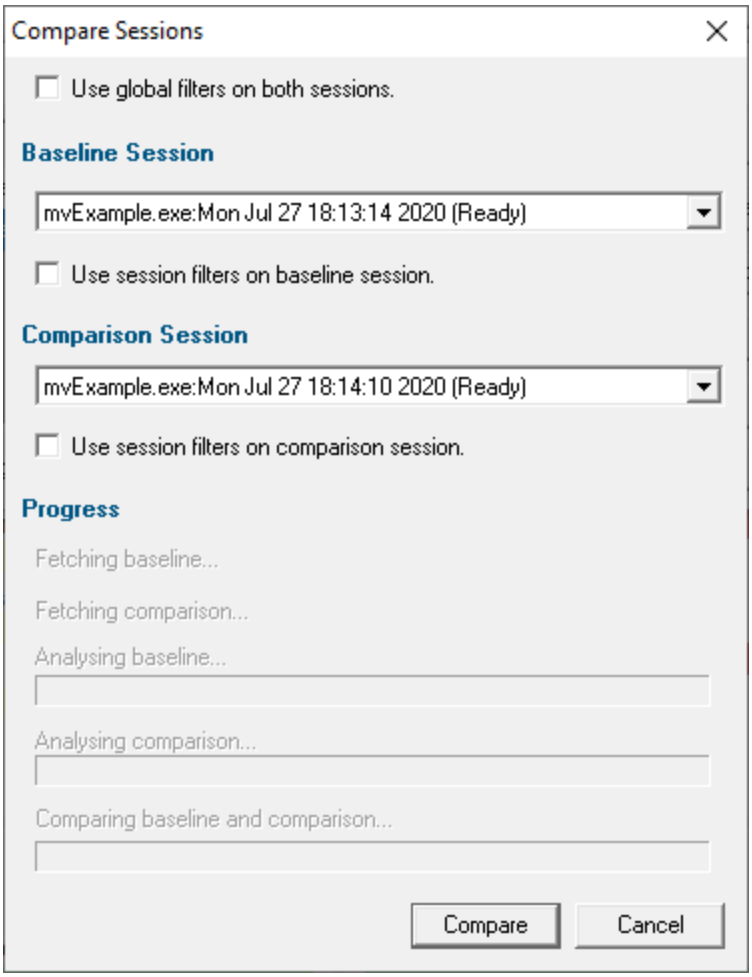
This second run is the *comparison* session.

Now that we have two sessions, we can compare them to look for common leaks, improvements and regressions.

Managers menu › **Session Manager** › **select** the two recent sessions



- **Compare** › shows the compare sessions dialog › check the baseline session is the one with the earlier timestamp

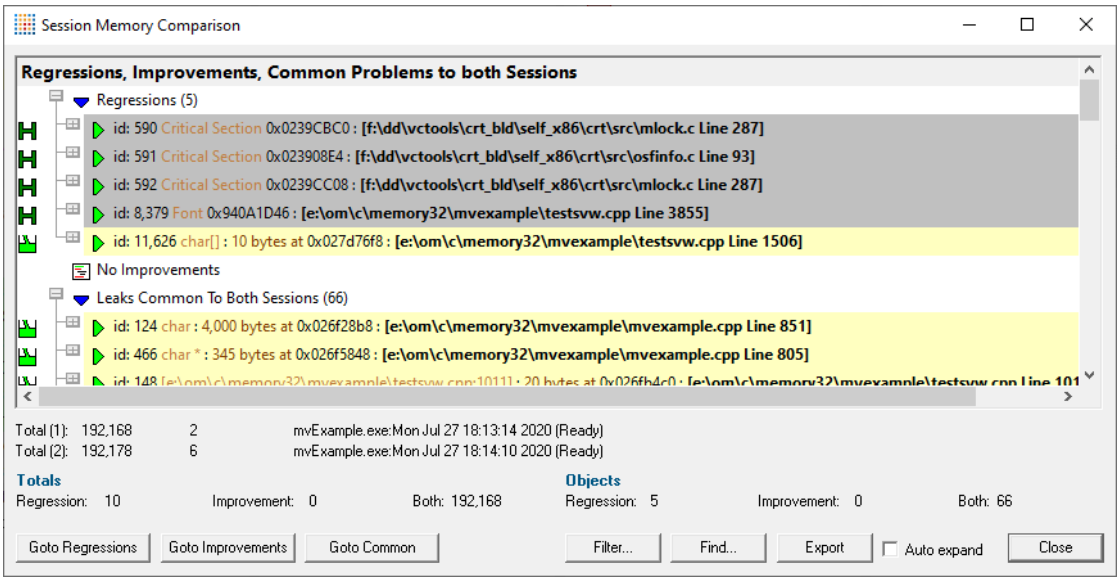


- **Compare** › shows the Session Memory Comparison dialog

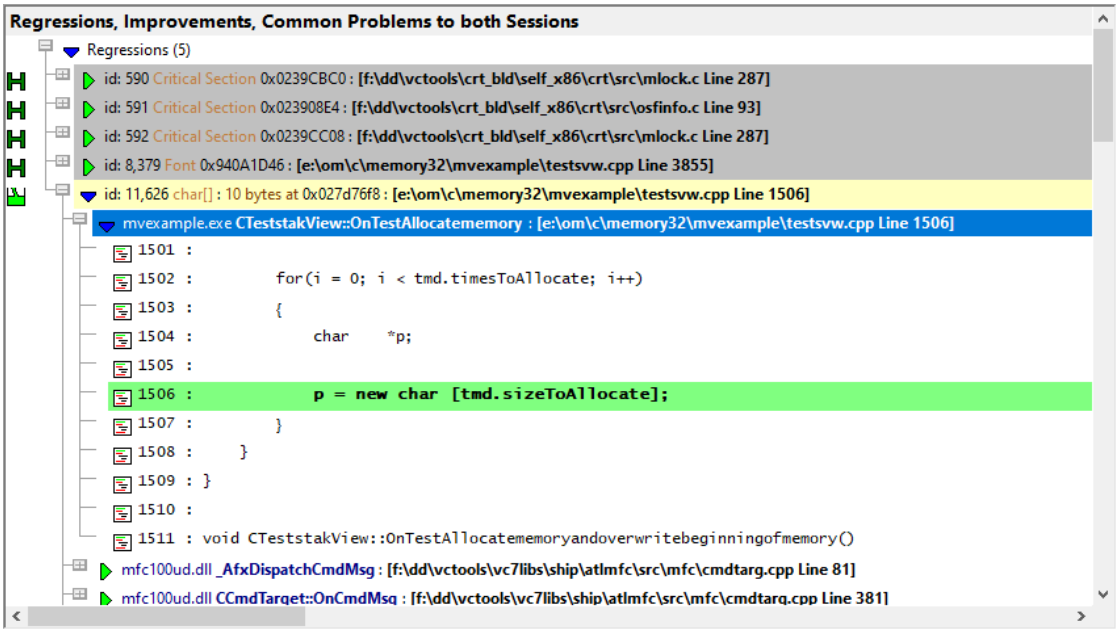
  The Session Memory Comparison dialog displays all the regression, improvements and leaks common to both sessions.

  You can filter, search and export the results, search the results and export the results as HTML or XML.

  In this example you should see one item marked as a regression, and all the rest of the leaks being common to both sessions.



- **expand** the regression item › shows the regression leak allocation in `CTeststakView::OnTestAllocatememory()`



12.12 Using bookmarks

Example use of bookmarks

To demonstrate the use of bookmarks we'll run the example application once, add some bookmarks, do more work and then easily return to the bookmarked items.

Memory Validator

- **Memory** tab › **Display...** › ensure **All Memory (leaks, errors, unleaked)** is selected in the first combo box

launch nativeExample.exe › **wait** until attaching is complete

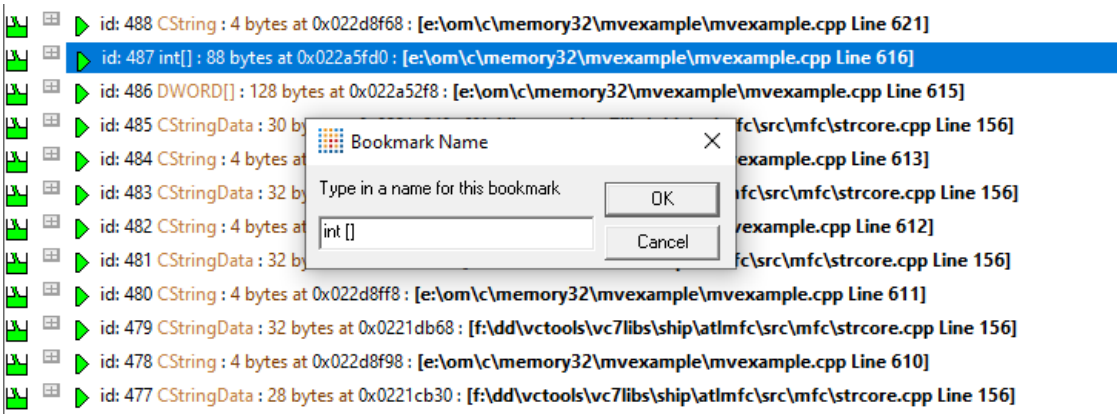
- **Memory** tab › **Refresh** › the display updates to include the usual selection of allocations

Pick three allocations of interest, perhaps the most recent allocations of different datatypes, such as `CString`, `int []` or `DWORD []`

For each one, do the following:

Add Bookmark... › shows the Bookmark Name dialog pre-filled with an automatic name

- **Enter** a name of your choice, for example relating to the data type you selected › **OK** › adds a bookmark to the system



There are now three bookmarks for three different traces in the bookmark manager.

We'll come back to these later, but first we'll make some more allocations.

nativeExample.exe

Allocations menu › **Test Many Hooks at Once** › makes a selection of allocations

Feel free to make any more allocations you wish, using the test options in the example program.

Memory Validator

- **Memory** tab › **Refresh** › the display should refresh to include the new items.

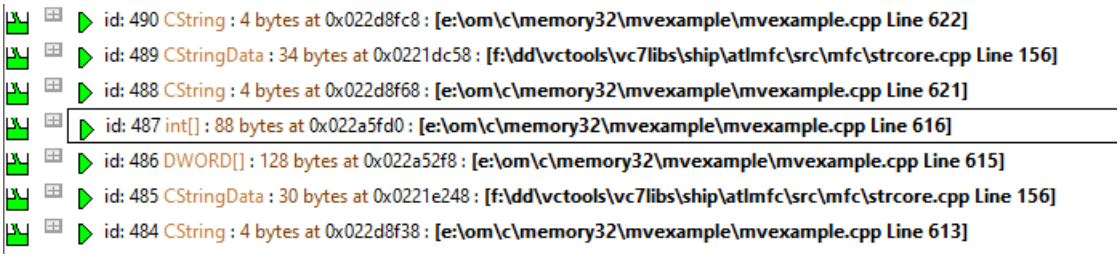
The items you bookmarked are a little further down the display, but in a real world scenario, they might be lost in a sea of data!

Managers menu › **Bookmark Manager...** › displays the Bookmarks dialog

You should see the three bookmarks you added earlier

Select one of the bookmarks. In this case we'll choose **int [ ]**.

- **Goto** › scrolls the display to the original item bookmarked for this name.



Bookmarks can only be used on the Memory tab and the Analysis tab.

If the memory has been deallocated the trace will no longer be displayed on the memory view. The bookmark manager won't be able to find it. Use the Analysis view instead.

nativeExample.exe

File menu › **Exit**

12.13 Using watermarks

Watermarks are your superpower

Watermarks are a really powerful way to partition and associate allocation events to actions or time periods that matter to you and your program.

If you link Memory Validator into your application to use the API, you can also add watermarks (and bookmarks) from within your code.

Using the API to do this is especially useful if your actions or time periods don't map well to user interaction triggers and events.

Example use of watermarks

To demonstrate the use of watermarks we'll run the example application once and add watermarks before and after doing some work.

We'll see a simple case of how to use watermarks to:

- partition allocation events
- reduce data in the display
- help detect memory leaks for specific events

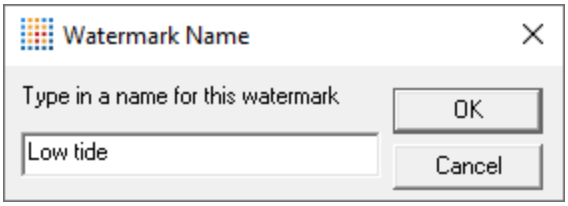
Memory Validator

- **Memory** tab › **Display...** › ensure **All Memory (leaks, errors, unleaked)** is selected in the first combo box

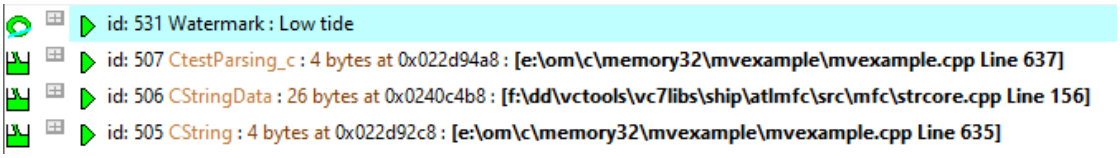
launch nativeExample.exe › **wait** until attaching is complete

- **Memory** tab › **Refresh** › the display updates to show the usual leaks and memory errors

Managers menu › **Add watermark at most recent trace** › shows the Watermark Name dialog



- **Enter** a name of your choice, for example 'Low tide' › **OK** › adds a named watermark to the system
- **Memory** tab › **Refresh** › the display should refresh to include the new watermark



nativeExample.exe

Allocations menu › **C Runtime (C/C++) Heap** › **Allocate memory...** › shows the **Test Memory** dialog › **OK** to accept the default values

Memory Validator

- **Memory** tab › **Refresh** › the display should refresh to include the new allocation.

Managers menu › **Add watermark at most recent trace** › shows the Watermark Name dialog again

- **Enter** a name of your choice, for example 'High tide' › **OK** › adds another named watermark to the system
- **Memory** tab › **Refresh** › the display includes the new watermark

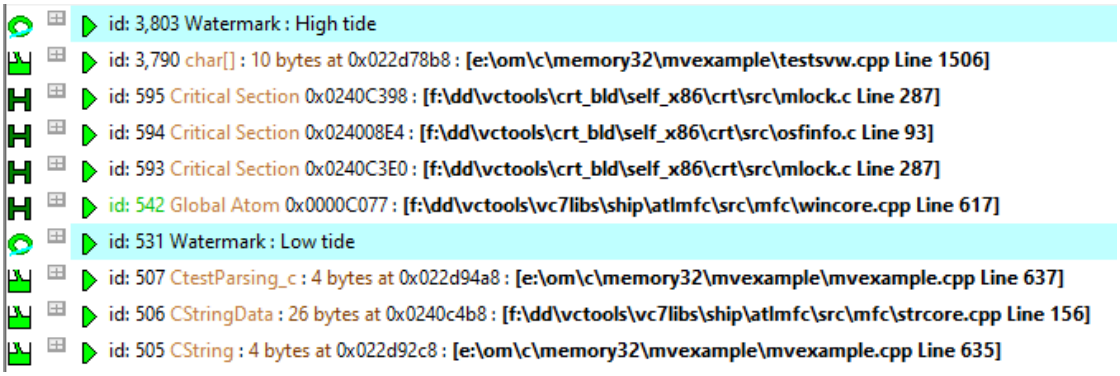
nativeExample.exe

Help menu › **About nativeExample.exe...** › shows the About box, and deliberately leaks some memory at the same time

Memory Validator

- **Memory** tab › **Refresh** › the display includes the new items.

  By now the display looks something like this:



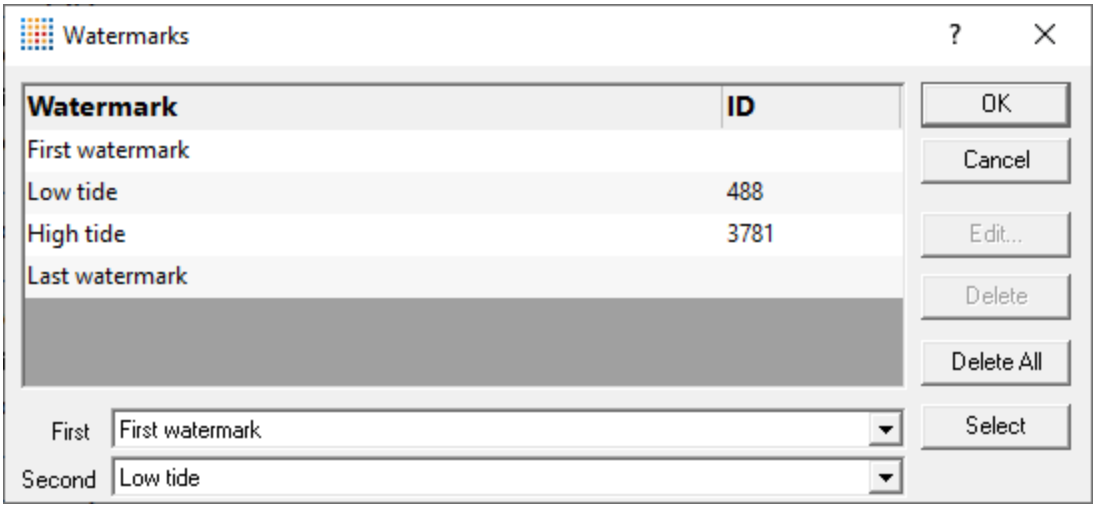
  Note the following:

  - traces *between* the watermarks represent objects allocated after the first watermark and before the second
  - memory allocated *after* the second watermark - i.e. when the About Box was displayed.

Without using watermarks, it would not have been obvious which items in the display related to the test allocation, or the about box.

Managers menu › **Watermark Manager...** › displays the Watermarks dialog

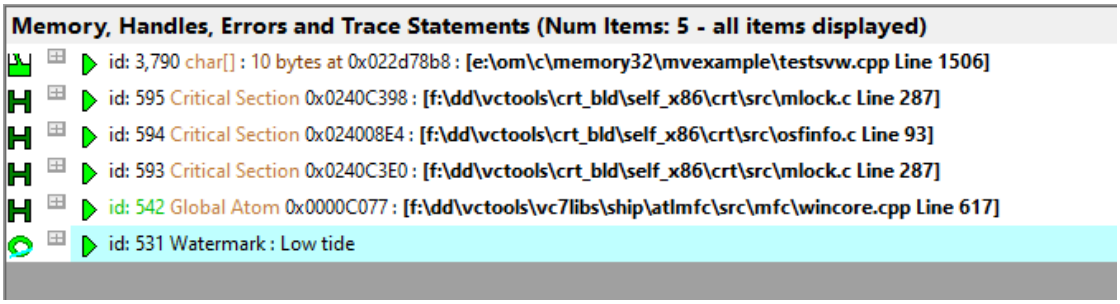
You should see the watermarks you added earlier, as well as the permanent first and last watermarks in the list.



Using watermarks to reduce data in the display

We'll now use the two watermarks we added, to reduce the data in the display

- **First** › **choose** Low tide
- **Second** › **choose** High tide › **OK** › changes the watermark settings in the Memory tab to the selected values
- **Memory** tab › **Refresh** › the display changes to including only the data between the selected watermarks



Using watermarks to help detect memory leaks

nativeExample.exe

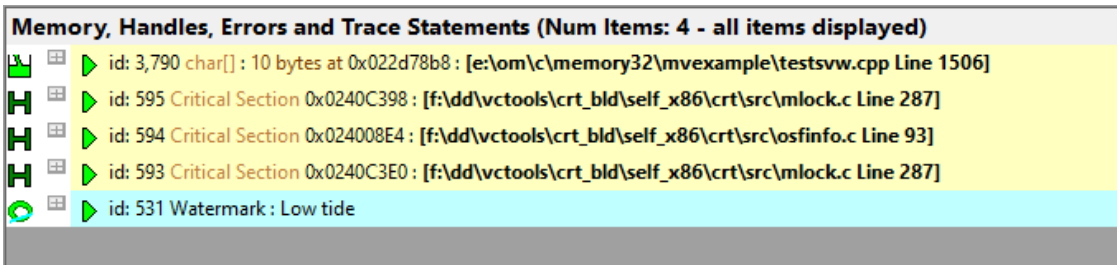
File menu › **Exit**

Memory Validator

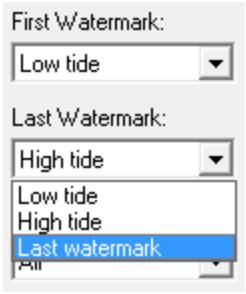
wait for data transfer to complete

- **Memory** tab › **Refresh** › the display updates to show leaked items

  We can see that the allocations that happened between the watermarks are real leaks at the time the program closes.

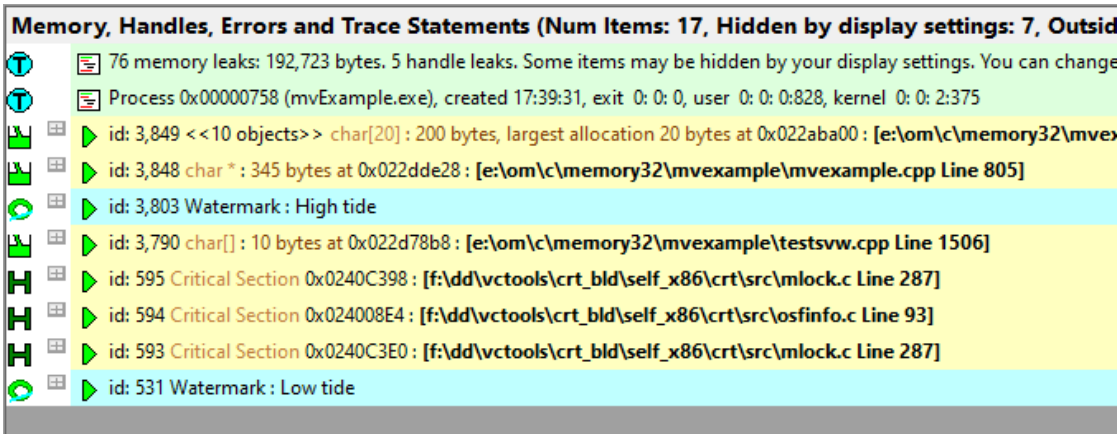


- **Memory** tab › **Last watermark** › select the **Last watermark** option



- **Memory** tab › **Refresh** › the display updates to show leaked items

  We can now see that the allocations that happened after the High tide watermark are also real leaks.



12.14 Example NT Service

The example NT Service

As well as the example application, an example *service* is provided along with details about building it.

There's also an example client

The example service demonstrates how to use the NT Service API to call the two functions required to use Memory Validator with NT Services.

The following tasks are performed when the service is started:

- Loads the Memory Validator stub DLL into the service
- Deliberately leaks some memory so that you can see this in the Memory Validator user interface
- Performs the normal work of the service until it's stopped
- Unloads the Memory Validator stub DLL from the service
- Informs the service control manager that a stop is pending

➡ Read more about working with NT Services.

12.14.1 Building the sample service

Example service project files

The example project can be found in the `examples\service` subdirectory in the directory where Memory Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

There are two project files in the directory:

- **service.dsp** › for Microsoft® Developer Studio® 6.0
- **service.vcproj** › for Microsoft® Visual Studio / .net

Configurations

There are a small number of configurations in each project:

- **Debug / Release** › dynamically links to the `svlMVStubService(_x64).lib` demonstrating use with the NT Service API

- **DebugStatic / ReleaseStatic** › statically linked versions of the same

Using the service

The service is named **MV Simple Service** in the control panel services dialog, and provides the following command line options:

- **-install** › Install the service
- **-remove** › Uninstall the service
- **-start** › Start the service
- **-stop** › Stop the service
- **-debug** › Run as a console application for debugging
- **-?** › Display the help message
- **-help** › Display the help message

Open a cmd prompt in administrator mode, navigate to the location of the service executable, and use one of these commands to install, remove, start, stop the service.

Examples:

```
serviceMV.exe -install

serviceMV.exe -start

serviceMV.exe -stop

serviceMV.exe -remove
```

12.14.2 Building the sample client

If you've already built the sample service, the process is very similar

Project files

The example project can be found in the `examples\serviceClient` subdirectory in the directory where Memory Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

There are two project files in the directory:

- **serviceClient.dsp** › for Microsoft® Developer Studio® 6.0

- **serviceClient.vcproj** › for Microsoft® Visual Studio / .net

Configurations

There are a small number of configurations in each project:

- **Debug / Release** › dynamically links to the `svlMVStubService(_x64).lib` demonstrating use with the NT Service API

Using serviceClient

The service is named **MV Simple Service** in the control panel services dialog, and provides the following command line options:

- **-string** › Sends the following (optionally quoted) text to the service. If the service is running the service will return the string in reverse order

  For example: `serviceClient.exe -string "The quick brown fox"` returns `"xof nworb kciuq ehT"`

- **-help** › Display the help message

12.14.3 Building the sample service utility

The serviceMutex project demonstrates a way of controlling whether Memory Validator is used without having to rebuild your service.

Project files

The example project can be found in the `examples\serviceMutex` subdirectory in the directory where Memory Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

There are two project files in the directory:

- **serviceMutex.dsp** › for Microsoft® Developer Studio® 6.0
- **serviceMutex.vcproj** › for Microsoft® Visual Studio / .net

Configurations

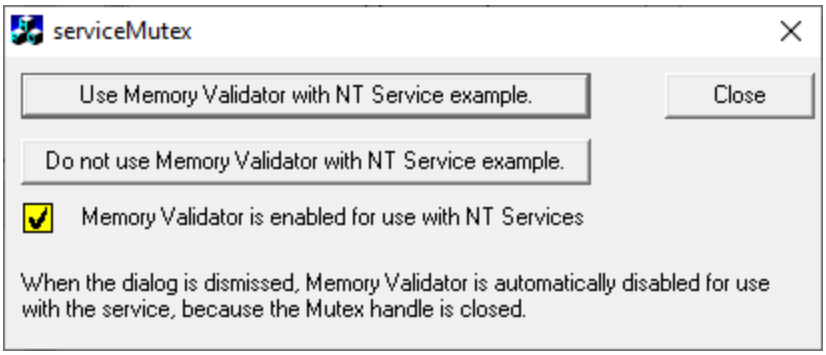
There are a small number of configurations in each project:

- **Debug / Release** › dynamically links to the `svlMVStubService(_x64).lib` demonstrating use with the NT Service API

Using the service utility

The utility provides a dialog box interface to allow the control over the creation of a mutex object with the name specified in the `service.h` header file.

Only if the service is started with the mutex created, does the service load Memory Validator.



If you don't like using mutexes in this way, you could change the code in the service and the utility to communicate through shared memory, a registry setting or another method of your choice.

12.14.4 Monitoring the service

Once the example service and example client has been built, the next step is to test them using Memory Validator.

Installing the service

If you haven't installed the service, do the following:

- open an administrator mode cmd prompt
- navigate to the directory containing the serviceMV.exe to install
- serviceMV.exe -install

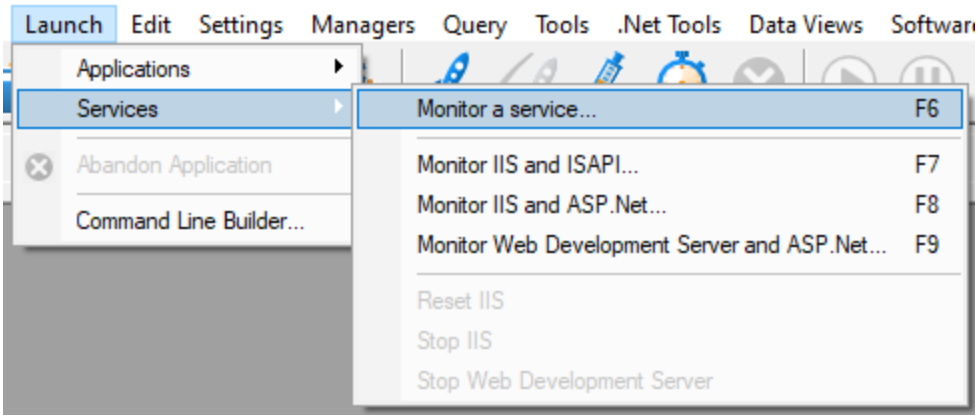
Monitoring the service

Prerequisites

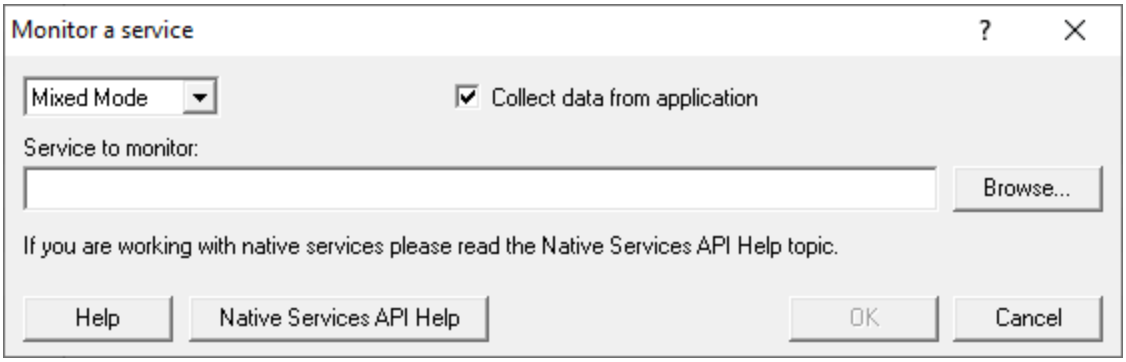
- example service has been installed, but not started (if service has been started, stop the service)
- example service and example client have been built

The following process is used to monitor the application launched by the service:

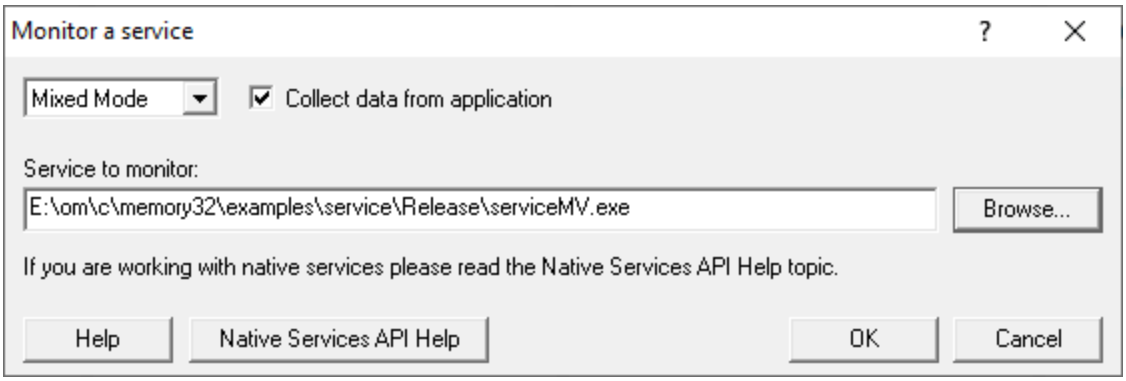
- From the Launch menu choose `Services > Monitor a Service...`



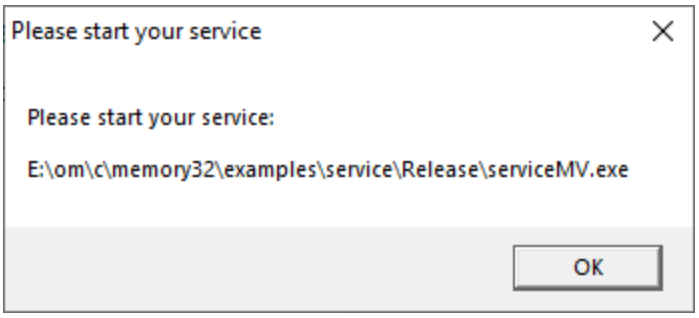
- The Monitor a service dialog is displayed



- Use **Browse...** to open the file chooser dialog and choose the service that will be monitored by Memory Validator.



- Click **OK**
- Memory Validator sets up a variety of parameters then displays a dialog box asking you to start you service. Click **OK** to dismiss the dialog



- Start your service. For the example `serviceMV.exe` do the following
  - open an administrator mode cmd prompt
  - navigate to the directory containing the `serviceMV.exe` to start
  - `serviceMV.exe -start`
  - `serviceMV.exe` starts will be monitored by Memory Validator
- The target application contacts Memory Validator
- Data is collected until the service finishes executing
- Memory Validator displays the results

12.15 Example Application Launched from a Service

The example Application launched from a Service

This pair of projects create an application that is launched from a service.

The purpose of this example is to show how to monitor the application that is launched from the service. This is also the same process for monitoring an application launched by an application launched from a service.

This process is subtly different to the method for working with services (see the example service for that).

Service

The service project is `serviceWithAChildProcess.vcxproj`

The following tasks are performed when the service is started:

- the test application is launched from the service

Application

The application project is `serviceChildProcess.vcxproj`

The application's first task is to load Memory Validator into the application.

- Loads the Memory Validator stub DLL into the application
- Configures the NT Service API to communicate to Memory Validator
- Does some work that can be monitored by Memory Validator
- Exits

Implementation Details

For implementation details see `attachToMemoryValidator();` in `serviceChildProcess.cpp`.

The application will need to link to the NT Service API, for example `..\..\..\svlMVStubService\release_2010_x64\svlMVStubService_x64.lib` (for a release x64 EXE/DLL).

Important. Call `attachToMemoryValidator() as close to the start of your application as possible, before any threads have been created.`

➡ Read more about working with NT Services.

12.15.1 Building the service and application

Example solution files

The example solution can be found in the `examples\serviceWithAChildProcess` subdirectory in the directory where Memory Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

Example project files

The example projects can be found in the subdirectories in the directory where Memory Validator was installed.

`examples\serviceWithAChildProcess\serviceWithAChildProcess`

- **serviceWithAChildProcess.vcproj** › for Microsoft® Visual Studio / .net

`examples\serviceWithAChildProcess\serviceChildProcess`

- **serviceChildProcess.vcproj** › for Microsoft® Visual Studio / .net

Configurations

There are a small number of configurations in each project:

- **Debug / Release** › dynamically links to the `svlMVStubService(_x64).lib` demonstrating use with the NT Service API

Using the service

The service is named **SVL *** MV Child Process** in the control panel services dialog (*** changes depending on the build configuration), and provides the following command line options:

- **-install** › Install the service
- **-remove** › Uninstall the service
- **-start** › Start the service
- **-stop** › Stop the service
- **-debug** › Run as a console application for debugging
- **-?** › Display the help message
- **-help** › Display the help message

Open a cmd prompt in administrator mode, navigate to the location of the service executable, and use one of these commands to install, remove, start, stop the service.

Examples:

```
serviceWithAChildProcess.exe -install

serviceWithAChildProcess.exe -start

serviceWithAChildProcess.exe -stop

serviceWithAChildProcess.exe -remove
```

12.15.2 Monitoring the application launched from the service

Once the example service and example application are built, the next step is to test them using Memory Validator.

Installing the service

If you haven't installed the service, do the following:

- open an administrator mode cmd prompt
- navigate to the directory containing the serviceWithAProcess.exe to install
- serviceWithAProcess.exe -install

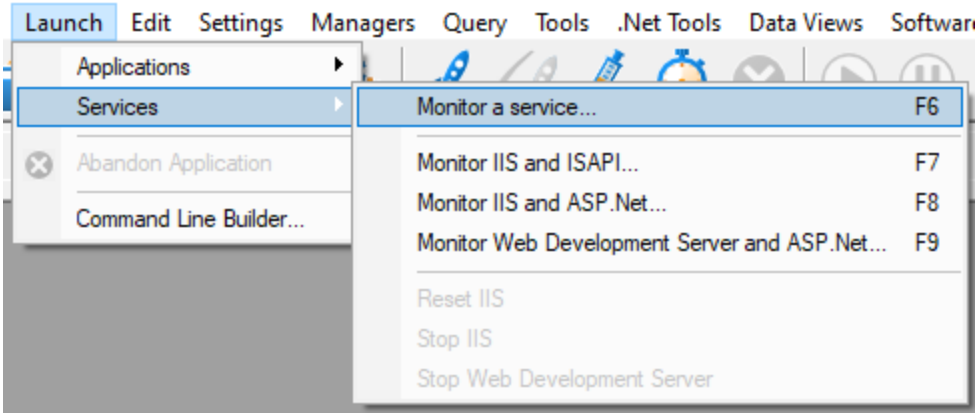
Monitoring the application launched by the service

Prerequisites

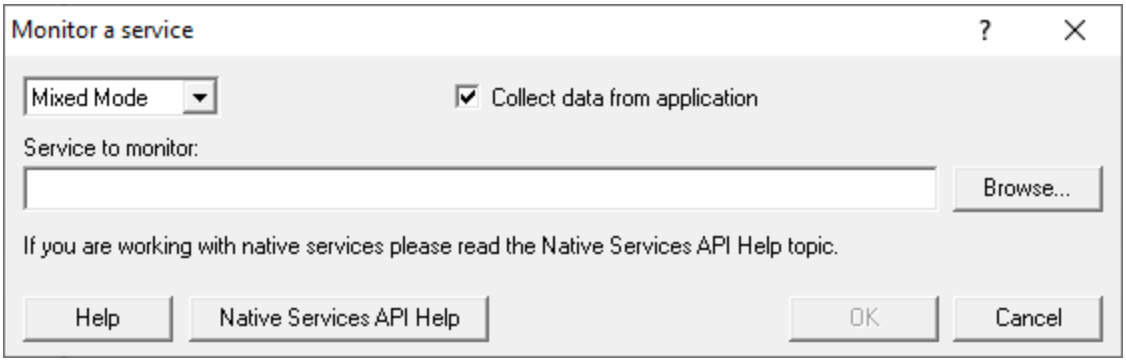
- example service has been installed, but not started (if service has been started, stop the service)
- example service and example application have been built (application must use the NT Service API as demonstrated in attachToMemoryValidator())
- example application executable is in the same directory as the example service (this is only a requirement for the example)

The following process is used to monitor the application launched by the service:

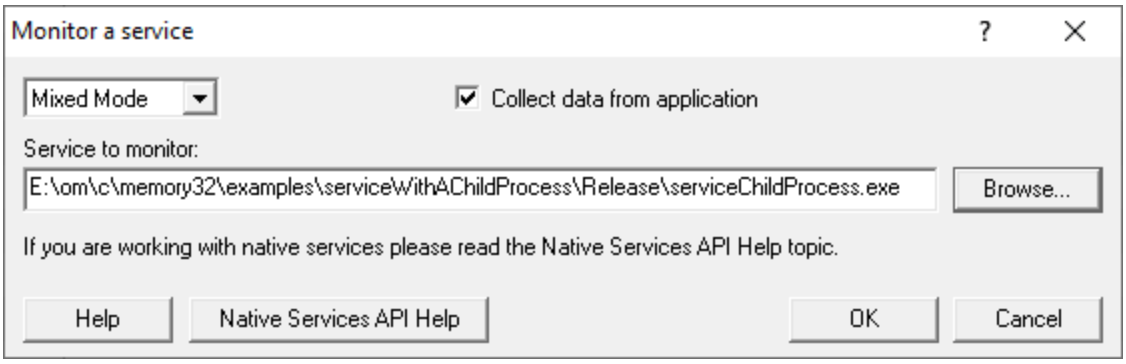
- From the Launch menu choose `Services > Monitor a Service...`



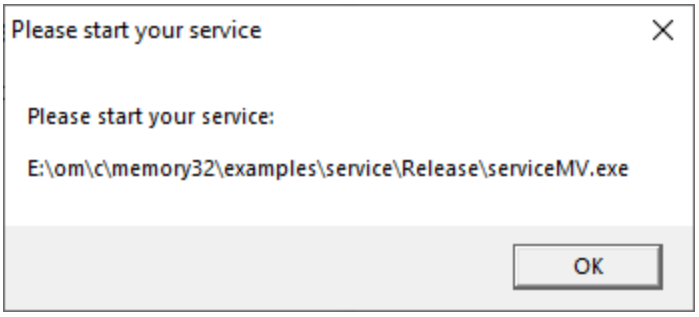
- The Monitor a service dialog is displayed



- Use **Browse...** to open the file chooser dialog and choose the application that will be monitored by Memory Validator. This is the application that is launched by the service. Do not choose the service



- Click **OK**
- Memory Validator sets up a variety of parameters then displays a dialog box asking you to start you service. Click **OK** to dismiss the dialog



- Start your service. For the example `serviceWithAChildProcess.exe` do the following
  - open an administrator mode cmd prompt
  - navigate to the directory containing the `serviceWithAProcess.exe` to start
  - `serviceWithAProcess.exe -start`
  - `serviceWithAProcess.exe` starts and launches the child process `serviceChildProcess.exe` that will be monitored
- The target application contacts Memory Validator
- Data is collected until the target process finishes executing
- Memory Validator displays the results

Part XIII

13 Hook Reference

The tables on the following pages list the hooks used by Memory Validator and are for reference only.

New hooks may be added in later versions of the software.

Enabling and disabling the hooks

You can individually enable and disable in the hook settings:

- Memory allocation hooks

  Includes C/C++, Win32, COM, LocalAlloc & GlobalAlloc, BSTR etc

- Handle allocation hooks

  All handle related hooks - Kernel, Advapi, GDI, Shell, Sockets, Printer etc

- Buffer manipulation hooks

  Kernel buffer
  Memory or string copying, moving and comparisons
  Internet path and registry functions

You can also specify custom hooks for functions that Memory Validator does not initially know about.

Hooking DLLS

Enabling or disabling hooks on a per-DLL basis using the Hooked DLLs settings.

13.1 C/C++ Memory Hooks

C and C++ Runtime memory hooks

For each of the compilers and linkers, the following C/C++ runtime functions are hooked.

Relevant runtime DLLs are listed for each system

⊟ Microsoft / Intel

Function	C Runtime DLLs
	Release

- malloc
- calloc
- realloc
- free
- _expand
- new
- delete

- aligned_malloc
- aligned_realloc
- aligned_offset_malloc
- aligned_offset_realloc
- aligned_free

The following are hooked for debug DLLs only

- malloc_dbg
- calloc_dbg
- realloc_dbg
- free_dbg
- _expand_dbg

- aligned_malloc_dbg
- aligned_realloc_dbg
- aligned_offset_malloc_dbg
- aligned_offset_realloc_dbg
- aligned_free_dbg

MSVCRT40.DLL	Visual Studio 4.0
MSVCRT.DLL	Visual Studio 6.0 (1996)
MSVCR70.DLL	Visual Studio 7.0 (2002)
MSVCR71.DLL	Visual Studio 7.1 (2003)
MSVCR80.DLL	Visual Studio 8.0 (2005)
MSVCR90.DLL	Visual Studio 9.0 (2008)
MSVCR100.DLL	Visual Studio 10.0 (2010)
MSVCR110.DLL	Visual Studio 11.0 (2012)
MSVCR120.DLL	Visual Studio 12.0 (2013)
APPCRT140.DLL	Visual Studio 14.0 (2014)

Debug

MSVCRT40D.DLL	Visual Studio 4.0
MSVCRTD.DLL	Visual Studio 6.0 (1996)
MSVCR70D.DLL	Visual Studio 7.0 (2002)
MSVCR71D.DLL	Visual Studio 7.1 (2003)
MSVCR80D.DLL	Visual Studio 8.0 (2005)
MSVCR90D.DLL	Visual Studio 9.0 (2008)
MSVCR100D.DLL	Visual Studio 10.0 (2010)
MSVCR110D.DLL	Visual Studio 11.0 (2012)
MSVCR120D.DLL	Visual Studio 12.0 (2013)
APPCRT140D.DLL	Visual Studio 14.0 (2014)

Metrowerks CodeWarrior for Windows V8 & V9

Function	C Runtume DLLs
• malloc • calloc • realloc • free • _expand • new • delete	**Release** MSL_All-DLL80_x86.DLL MSL_All-DLL90_x86.DLL **Debug** MSL_All-DLL80_x86_D.DLL MSL_All-DLL90_x86_D.DLL

Salford Software FORTRAN95

Function	C Runtume DLLs
• __ALLOCATE1 • __DEALLOCATE • __PALLOCATE1 • __PALLOCATE2 • __PFREE • __PFREE1	SALFLIBC.DLL

Borland C++ Builder

Function	C Runtume DLLs
• malloc • calloc • realloc • free • _expand • new • delete	CC3250.DLL CC3250MT.DLL

Borland Delphi

Function	C Runtume DLLs
• GetMem • AllocMem • ReallocMem • FreeMem • SysGetMem • SysAllocMem • SysReallocMem • SysFreeMem	BORLNDMM.DLL

13.2 Win32 Memory Hooks

Win32 memory and virtual memory hooks

Kernel32.dll (Memory allocations)

- GlobalAlloc
- GlobalReAlloc
- GlobalFree

- HeapAlloc
- HeapReAlloc
- HeapFree

- LocalAlloc
- LocalReAlloc
- LocalFree

- AllocateUserPhysicalPages
- MapViewOfFile
- MapViewOfFile2
- MapViewOfFile3
- MapViewOfFileFromApp
- MapViewOfFileNumaW
- MapViewOfFileEx
- UnmapViewOfFile
- UnmapViewOfFile2
- UnmapViewOfFileEx

- VirtualAlloc
- VirtualAlloc2
- VirtualAlloc2FromApp
- VirtualAllocFromApp

- AllocateUserPhysicalPages2
- AllocateUserPhysicalPagesNuma
- FreeUserPhysicalPages
- VirtualFree
- VirtualAllocEx
- VirtualAllocExNuma
- VirtualFreeEx
- VirtualAllocVlm
- VirtualFreeVlm

Ole32.dll (COM/OLE memory allocations)

- CoTaskMemAlloc
- CoTaskMemRealloc
- CoTaskMemFree

- IMallocSpy

OleAut32.dll (BSTR memory allocations)

- SysAllocString
- SysAllocStringByteLen
- SysAllocStringLen
- SysReAllocString
- SysReAllocStringLen
- SysFreeString

- VariantClear
- VariantCopy
- VariantCopyInd
- VariantChangeType
- VariantChangeTypeEx
- SafeArrayCreate
- SafeArrayCreateEx
- SafeArrayCreateVector
- SafeArrayCreateVectorEx
- SafeArrayDestroy
- SafeArrayAllocData
- SafeArrayDestroyData
- SafeArrayGetElement
- SafeArrayPutElement
- SafeArrayAllocDescriptor
- SafeArrayAllocDescriptorEx
- SafeArrayDestroyDescriptor
- SafeArrayLock
- SafeArrayUnlock
- SafeArrayAccessData
- SafeArrayUnaccessData

13.3 Handle Hooks

Kernel32.dll

- CloseHandle

- CreateFileW
- CreateFileA
- OpenFile
- DeleteFileW
- DeleteFileA

- LockFile
- LockFileEx
- UnlockFile
- UnlockFileEx

- CreateEventW
- CreateMutexW
- CreateMutexA
- OpenMutexW
- OpenMutexA
- ReleaseMutex

- CreateNamedPipeW
- CreateNamedPipeA
- CreatePipe
- DisconnectNamedPipe

- CreateProcessW
- CreateProcessA
- OpenProcess
- CreateJobObjectW
- CreateJobObjectA
- OpenJobObjectW
- OpenJobObjectA
- TerminateJobObject

- BeginUpdateResourceW
- BeginUpdateResourceA
- EndUpdateResourceW
- EndUpdateResourceA

- CreateConsoleScreenBuffer
- GetConsoleWindow

- CreateEventA
- OpenEventW
- OpenEventA

- CreateFileMappingW
- CreateFileMappingA
- CreateFileMapping2
- CreateFileMappingFromApp
- CreateFileMappingNumaW
- OpenFileMappingW
- OpenFileMappingA
- OpenFileMappingFromApp

- FindFirstFileW
- FindFirstFileA
- FindFirstFileEx
- FindClose

- HeapCreate
- HeapDestroy

- LoadLibraryW
- LoadLibraryA
- LoadLibraryExW
- LoadLibraryExA
- FreeLibrary

- CreateFiber
- DeleteFiber

- CreateMailslotW
- CreateMailslotA

- TerminateProcess

- CreateSemaphoreW
- CreateSemaphoreA
- OpenSemaphoreW
- OpenSemaphoreA
- ReleaseSemaphore

- CreateThread
- CreateRemoteThread
- OpenThread
- TerminateThread

- CreateWaitableTimerW
- CreateWaitableTimerA
- OpenWaitableTimerW
- OpenWaitableTimerA
- CancelWaitableTimer

- CreateTimerQueue
- DeleteTimerQueue
- DeleteTimerQueueEx
- CreateTimerQueueTimer
- DeleteTimerQueueTimer

- CreateIoCompletionPort

- AddAtomW
- AddAtomA
- DeleteAtom
- GlobalAddAtomW
- GlobalAddAtomA
- GlobalDeleteAtom

- FindFirstVolume
- FindFirstVolumeMountPoint
- FindVolumeClose
- FindFirstVolumeMountPointClose

- FindFirstChangeNotificationW
- FindFirstChangeNotificationA
- FindCloseChangeNotification

- InitializeCriticalSection
- InitializeCriticalSectionAndSpinCount
- DeleteCriticalSection

- _lopen
- _lclose

Advapi32.dll

- RegCreateKeyW
- RegCreateKeyA
- RegCreateKeyExW
- RegCreateKeyExA
- RegOpenKeyW
- RegOpenKeyA
- RegOpenKeyExW
- RegOpenKeyExA
- RegCloseKey

- OpenBackupEventLogW
- OpenBackupEventLogA

- OpenEventLogW
- OpenEventLogA
- CloseEventLog

- RegConnectRegistryW
- RegConnectRegistryA

- RegisterEventSourceW
- RegisterEventSourceA
- DeregisterEventSource

- RegOpenCurrentUser
- RegOpenUserClassesRoot

- OpenProcessToken
- OpenThreadToken
- CreateRestrictedToken
- DuplicateToken
- DuplicateTokenEx

GDI32.dll

- CreateBitmap
- CreateBitmapIndirect
- CreateCompatibleBitmap
- CreateDIBitmap
- CreateDIBSection
- CreateDiscardableBitmap

- CreateBrushIndirect
- CreateDIBPatternBrush
- CreateDIBPatternBrushPt
- CreateHatchBrush
- CreatePatternBrush
- CreateSolidBrush

- CreateDCW
- CreateDCA
- DeleteDC
- CreateCompatibleDC

- CreateEnhMetaFileW
- CreateEnhMetaFileA
- DeleteEnhMetaFile
- CloseEnhMetaFile

- CreateFontW
- CreateFontA
- CreateFontIndirectW
- CreateFontIndirectA

- CreateMetaFileW
- CreateMetaFileA
- DeleteMetaFile
- CloseMetaFile

- CreatePalette

- CreatePen
- CreatePenIndirect
- ExtCreatePen

- CombineRgn
- CreateEllipticRgn
- CreateEllipticRgnIndirect
- CreatePolygonRgn
- CreatePolyPolygonRgn
- CreateRectRgn
- CreateRectRgnIndirect
- CreateRoundRectRgn
- ExtCreateRegion
- PathToRegion

- DeleteObject

User32.dll

- CreateAcceleratorTableW
- CreateAcceleratorTableA
- DestroyAcceleratorTable

- LoadBitmapW
- LoadBitmapA

- CreateCaret
- DestroyCaret

- CreateCursor
- CopyCursor
- DestroyCursor

- ReleaseDC
- GetDC
- GetDCEx
- GetWindowDC

- DdeConnect
- DdeDisconnect
- DdeConnectList
- DdeDisconnectList

- GetThreadDesktop

- SetWindowsHookExW
- SetWindowsHookExA
- UnhookWindowsHookEx

- CreateIcon
- CreateIconFromResource
- CreateIconFromResourceEx
- CreateIconIndirect
- CopyIcon
- DestroyIcon

- LoadImageW
- LoadImageA
- CopyImage

- CreateMenu
- DestroyMenu
- CreatePopupMenu
- LoadMenuW
- LoadMenuA
- LoadMenuIndirectW
- LoadMenuIndirectA

- CreateWindowW
- CreateWindowA
- DestroyWindow
- CreateWindowExW
- CreateWindowExA

- CreateDialogParamW
- CreateDialogParamA
- CreateDialogIndirectParamW
- CreateDialogIndirectParamA

- CreateMDIWindowW
- CreateMDIWindowA

- BeginDeferWindowPos
- EndDeferWindowPos

- GetProcessWindowStation

- SetWindowRgn

Shell32.dll

- ExtractAssociatedIcon
- ExtractIcon
- ExtractIconEx
- DuplicateIcon

Comctl.dll

- ImageList_Create
- ImageList_Destroy
- ImageList_Duplicate
- ImageList_GetIcon
- ImageList_LoadImage
- ImageList_Merge
- CreateMappedBitmap
- CreateToolbarEx
- CreateUpDownControl

Sockets

- socket
- accept
- closesocket

WinHttp

- WinHttpOpen
- WinHttpOpenConnect
- WinHttpOpenRequest
- WinHttpCloseHandle

Printer

- OpenPrinterW
- OpenPrinterA
- ClosePrinter

- FindFirstPrinterChangeNotification
- FindClosePrinterChangeNotification
- FindNextPrinterChangeNotification

- FreePrinterNotifyInfo

13.4 COM Related Hooks

COM memory allocation

These four are also listed in the OleAut32.dll memory allocation functions

- SysAllocString
- SysReAllocString
- SysFreeString

- VariantClear

COM object creation

- CoCreateInstance
- CoCreateInstanceEx
- CoGetInstanceFromFile

- CoGetClassObject
- CoRegisterClassObject

- BindMoniker
- CreateAntiMoniker
- CreateClassMoniker
- CreateFileMoniker
- CreateItemMoniker
- CreatePointerMoniker
- MonikerCommonPrefixWith
- MonikerRelativePathTo
- CreateBindCtx
- CreateGenericComposite
- MkParseDisplayName

- OleCreate
- OleCreateDefaultHandler
- OleCreateEx
- OleCreateFontIndirect
- OleCreateFromData
- OleCreateFromFile
- OleCreateLink
- OleCreateStaticFromData
-
- OleGetClipboard
-
- OleLoad
- OleLoadFromStream
- OleLoadPicture
- OleLoadPictureFile
-
- OleRegEnumFormatEtc
- OleRegEnumVerbs

- StgCreateDocFile
- StgCreateDocFileOn
- StgGetIFillLockBytesOn
- StgOpenAsyncDocfileOn
- StgOpenStorage
- StgOpenStorageOn

- CreateDataAdviseHolder
- CreateOleAdviseHolder

- GetRunningObjectTable

COM Reference Counting

To track creation and lifetime of COM objects they are all hooked for the following methods:

- QueryInterface
- AddRef
- Release

.

13.5 Buffer Manipulation Hooks

Kernel32.dll buffer function hooks

- CopyMemory
- FillMemory

- MoveMemory
- ZeroMemory

CRT memory and string copying and moving

- memcpy
- memmove
- memset
- _memccpy

- strcat
- strcpy
- strdup
- strncat
- strncpy

- _strset
- _strnset

- wcscat
- wcscpy
- wcsncat
- wcsncpy

- _wcsdup
- _wcsset
- _wcsnset

- _mbscat
- _mbscpy
- _mbsdup
- _mbsset
- _mbsnbcat
- _mbsncpy
- _mbsnset
- _mbsnbset

CRT memory and string comparisons

- memchr
- memcmp

- _memicmp

- strchr
- strcmp
- strcoll
- strcspn
- strftime
- stricmp
- strlen
- strncmp
- strpbrk
- strrchr
- strspn
- strstr
- strtod
- strtok
- strtol
- strtoul
- strxfrm

- _stricoll
- _strlwr
- _strncoll
- _strnicmp
- _strnicoll
- _strupr

- wcschr
- wcscmp
- wcscoll
- wcscspn
- wcsftime
- wcsicmp
- wcslwr
- wcsncmp
- wcspbrk
- wcsrchr
- wcsspn
- wcstod
- wcstok
- wcstol
- wcstombs
- wcstoul
- wcsupr
- wcsxfrm

- _wcsicoll
- _wcslen
- _wcsncoll
- _wcsnicmp
- _wcsnicoll
- _wcsstr

- mbsspn
- mbstok
- mbstowcs

- _mbschr
- _mbscmp
- _mbscoll
- _mbscspn
- _mbsdec
- _mbsicmp
- _mbsicoll
- _mbsinc
- _mbslen
- _mbslwr
- _mbsnbcmp
- _mbsnbcnt
- _mbsnbcpy
- _mbsnbicmp
- _mbsnccnt
- _mbsncmp
- _mbsncoll
- _mbsnextc
- _mbsnicmp
- _mbsnicoll
- _mbsninc
- _mbspbrk
- _mbsrchr
- _mbsspnp
- _mbsstr
- _mbstrlen
- _mbsupr

Locale functions

- get_current_locale
- create_locale
- free_locale
- _get_current_locale
- _create_locale
- _free_locale

Internet functions

- StrChr
- StrChrI
- StrCmpN
- StrCmpNI
- StrCpyN
- StrCSpn
- StrCSpnI
- StrDup
- StrFormatByteSize
- StrFromTimeInterval
- StrIsIntlEqual
- StrNCat
- StrPBrk
- StrRChr
- StrRStrI
- StrSpn
- StrStr
- StrStrI
- StrToInt
- StrToIntEx
- StrTrim

Path functions

- PathAddBackslash
- PathAddExtension
- PathAppend
- PathBuildRoot
- PathCanonicalize
- PathCombine
- PathCompactPath
- PathCompactPathEx
- PathCommonPrefix
- PathFileExists
- PathFindExtension
- PathFindFileName
- PathFindNextComponent
- PathFindOnPath
- PathGetArgs
- PathGetDriveNumber
- PathIsContentType
- PathIsDirectory
- PathIsFileSpec
- PathIsPrefix
- PathIsRelative
- PathIsRoot
- PathIsSameRoot
- PathIsSystemFolder
- PathIsUNC
- PathIsUNCServer
- PathIsUNCServerShare
- PathIsURL
- PathMakePretty
- PathMatchSpec
- PathParseIconLocation
- PathQuoteSpaces
- PathUnquoteSpaces
- PathRelativePathTo
- PathRemoveArgs
- PathRemoveBackslash
- PathRemoveBlanks
- PathRemoveExtension
- PathRemoveFileSpec
- PathRenameExtension
- PathSearchAndQualify
- PathSkipRoot
- PathSetDlgItemPath
- PathStripPath
- PathStripToRoot
- PathMakeSystemFolder
- PathUnmakeSystemFolder

Registry functions

- SHDeleteEmptyKey
- SHDeleteKey
- SHEnumKeyEx
- SHDeleteValue
- SHQueryInfoKey
- SHQueryValueEx
- SHRegCreateUSKey
- SHRegDeleteEmptyUSKey
- SHRegDeleteUSValue
- SHRegGetUSValue
- SHRegEnumUSValue
- SHRegQueryUSValue
- SHRegSetUSValue

- SHEnumValue
- SHGetValue
- SHSetValue
- SHRegEnumUSKey
- SHRegOpenUSKey
- SHRegQueryInfoUSKey
- SHRegWriteUSValue
- SHOpenRegStream

13.6 Uninitialised Data Hooks

Memory Validator hooks all C++ constructors and destructors for which symbols are available in all DLLs that are loaded.

Each DLL that is loaded must be enabled for hooks.

13.7 Miscellaneous Memory Allocations

These functions allocate and free memory to manage miscellaneous functions in the WIN32 API.

⊟ Miscellaneous functions allocating memory

Ole32.dll

- CreateStreamOnHGlobal

Oleacc.dll

- AccessibleObjectFromPoint
- AccessibleObjectFromEvent

Kernel32.dll

- GetEnvironmentStringsA
- GetEnvironmentStringsW
- FreeEnvironmentStringsA
- FreeEnvironmentStringsW

- AllocateUserPhysicalPages
- FreeUserPhysicalPages

- AllocConsole
- FreeConsole

Advapi32.dll

- AllocateAndInitializeSid
- FreeSid

- FreeEncryptionCertificateHashList

- QueryUsersOnEncryptedFile
- QueryRecoveryAgentsOnEncryptedFile

13.8 LocalAlloc and GlobalAlloc Functions

Here are some hooked functions you may use that will indirectly allocate memory using LocalAlloc or GlobalAlloc

Functions using LocalAlloc

These functions use `LocalAlloc()` to allocate memory which must be freed with `LocalFree()`:

⊟ Advapi32.dll (LocalAlloc allocations)

- GetExplicitEntriesFromAclA
- GetExplicitEntriesFromAclW
- SetEntriesInAclA
- SetEntriesInAclW

- BuildSecurityDescriptorA
- BuildSecurityDescriptorW
- LookupSecurityDescriptorPartsA
- LookupSecurityDescriptorPartsW

- GetSecurityInfo
- GetNamedSecurityInfo

- ConvertStringSidToSidA
- ConvertStringSidToSidW
- ConvertSidToStringSidA
- ConvertSidToStringSidW

- ConvertStringSecurityDescriptorToSecurityDescriptorA
- ConvertStringSecurityDescriptorToSecurityDescriptorW
- ConvertSecurityDescriptorToStringSecurityDescriptorA
- ConvertSecurityDescriptorToStringSecurityDescriptorW

⊟ Kernel32.dll (LocalAlloc allocations)

- FormatMessageA
- FormatMessageW

- GetQueuedCompletionStatus

⊟ Crypt32.dll (LocalAlloc allocations)

- CryptDecodeObjectEx
- CryptEncodeObjectEx

- CryptGetKeyIdentifierProperty

- CryptUnprotectData

⊟ Setupapi.dll (LocalAlloc allocations)

- SetupGetFileCompressionInfoA
- SetupGetFileCompressionInfoW

Mprapi.dll (LocalAlloc allocations)

- MprAdminGetErrorString

Resutils.dll (LocalAlloc allocations)

- ResUtilDupString
- ResUtilExpandEnvironmentStrings
- ResUtilFindBinaryProperty
- ResUtilFindMultiSzProperty
- ResUtilFindSzProperty
- ResUtilFindExpandSzProperty
- ResUtilFindExpandedSzProperty
- ResUtilGetBinaryValue
- ResUtilGetMultiSzValue
- ResUtilGetSzValue
- ResUtilGetExpandSzValue
- ResUtilSetBinaryValue
- ResUtilSetMultiSzValue
- ResUtilSetSzValue
- ResUtilSetExpandSzValue

Functions using GlobalAlloc

These functions use `GlobalAlloc()` to allocate memory which must be freed with `GlobalFree()`.

Gdi32.dll (GlobalAlloc allocations)

- CMCreateDeviceLinkProfile
- CMCreateProfileA
- CMCreateProfileW

Mscms.dll (GlobalAlloc allocations)

- CreateDeviceLinkProfile
- CreateProfileFromLogColorSpace

Shell32.dll (GlobalAlloc allocations)

- CommandLineToArgW

Ole32.dll (GlobalAlloc allocations)

- CreateStreamOnHGlobal

13.9 Functions using CoTaskMemAlloc

These functions use `CoTaskMemAlloc()` to allocate memory which must be freed with `CoTaskMemFree()`:

Ole32.dll functions using CoTaskMemAlloc

- StringFromCLSID
- StringFromIID

- ReadFmtUserTypeStg

- CoQueryProxyBlanket
- CoQueryClientBlanket

- CoQueryAuthenticationServices

Shlwapi.dll functions using CoTaskMemAlloc

- SHStrDupA
- SHStrDupW

- StrRetToStrA
- StrRetToStrW

13.10 Net API Hooks

Netwapi.dll functions

- NetApiBufferAllocate
- NetApiBufferReallocate
- NetApiBufferFree

- NetStatisticsGet

- NetUseEnum
- NetUseGetInfo

- NetUserGetGroups
- NetUserGetInfo
- NetUserGetLocalGroups
- NetUserModalsGet

- NetWkstaGetInfo
- NetWkstaUserEnum
- NetWkstaUserGetInfo
- NetWkstaTransportEnum

- NetDfsGetInfo
- NetDfsEnum
- NetDfsGetClientInfo

- NetRemoteTOD

- NetGroupEnum
- NetGroupGetInfo
- NetGroupGetUsers

- NetQueryDisplayInformation

- NetShareGetInfo
- NetShareEnum
- NetShareEnumSticky

- NetServerDiskEnum
- NetServerEnum
- NetServerEnumEx
- NetServerGetInfo
- NetServerTransportEnum

- NetConnectionEnum

- NetSessionGetInfo
- NetSessionEnum

- NetGetDCName
- NetGetAnyDCName

- NetReplImportDirEnum
- NetReplImportDirGetInfo
- NetReplGetInfo
- NetReplExportDirEnum
- NetReplExportDirGetInfo

- NetMessageNameEnum
- NetMessageNameGetInfo

- NetLocalGroupGetMembers
- NetLocalGroupGetInfo
- NetLocalGroupEnum

- NetFileEnum
- NetFileGetInfo

- NetGetJoinInformation
- NetGetJoinableOUs

- NetScheduleJobEnum
- NetScheduleJobGetInfo

Part

14 Debug Information, Symbols, Filenames, Line Numbers

Depending on which IDE or compiler/linker combination the process to create debug information to ensure that you have symbols, filenames and line numbers is different.

This section shows you what to do to ensure you have symbols for your compiler and linker.

14.1 Visual Studio

Enabling debug information in Visual Studio has changed over the years depending on the version of Visual Studio you are using.

It's generally the same, but there have been some changes in recent versions that can cause confusion.

By default debug configurations create debug information, but for some versions of Visual Studio, release configurations do not create debug information.

You need to set both compiler and linker settings to get debug information. ***Setting just one or the other will not give you debug information you can use.***

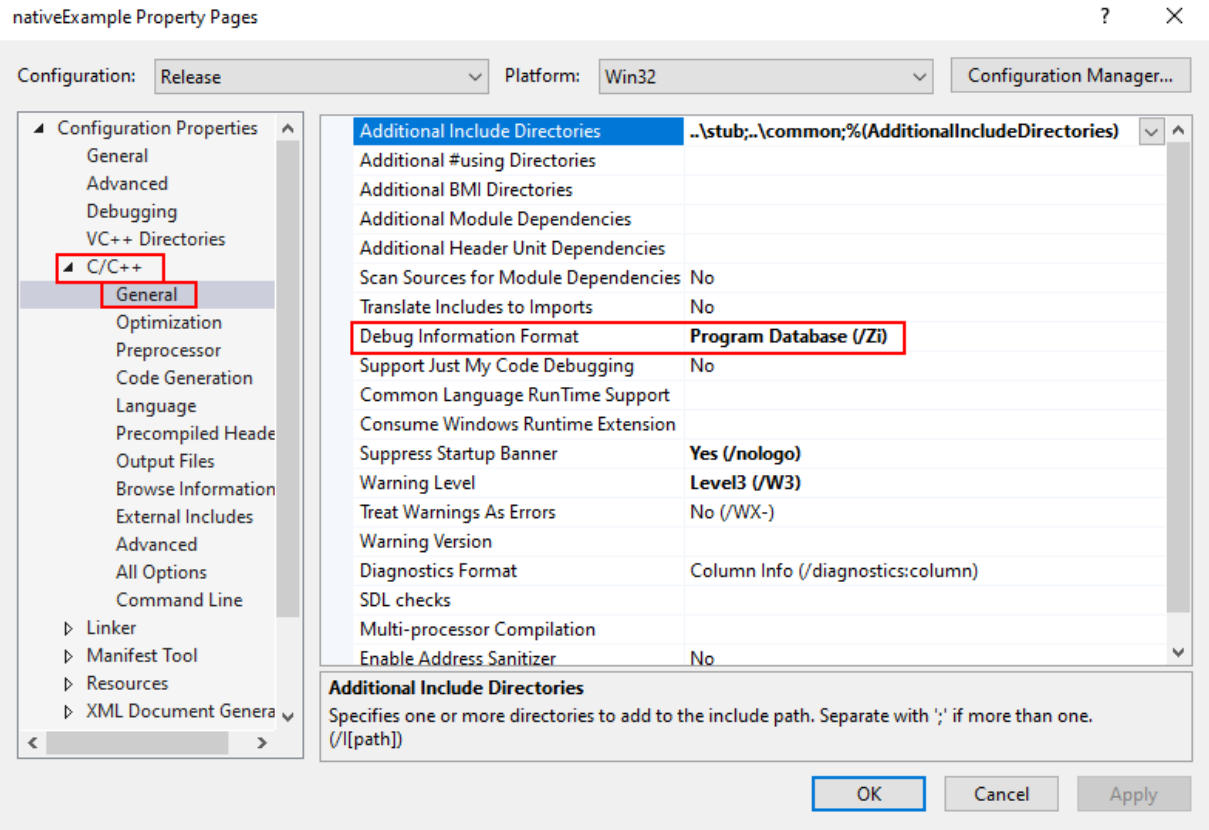
Configurations

In the help below we show you how to modify one configuration, for example Release | Win32.

You need to modify all configurations appropriately. Release, Debug, Win32, Win64 and any other configurations you are using.

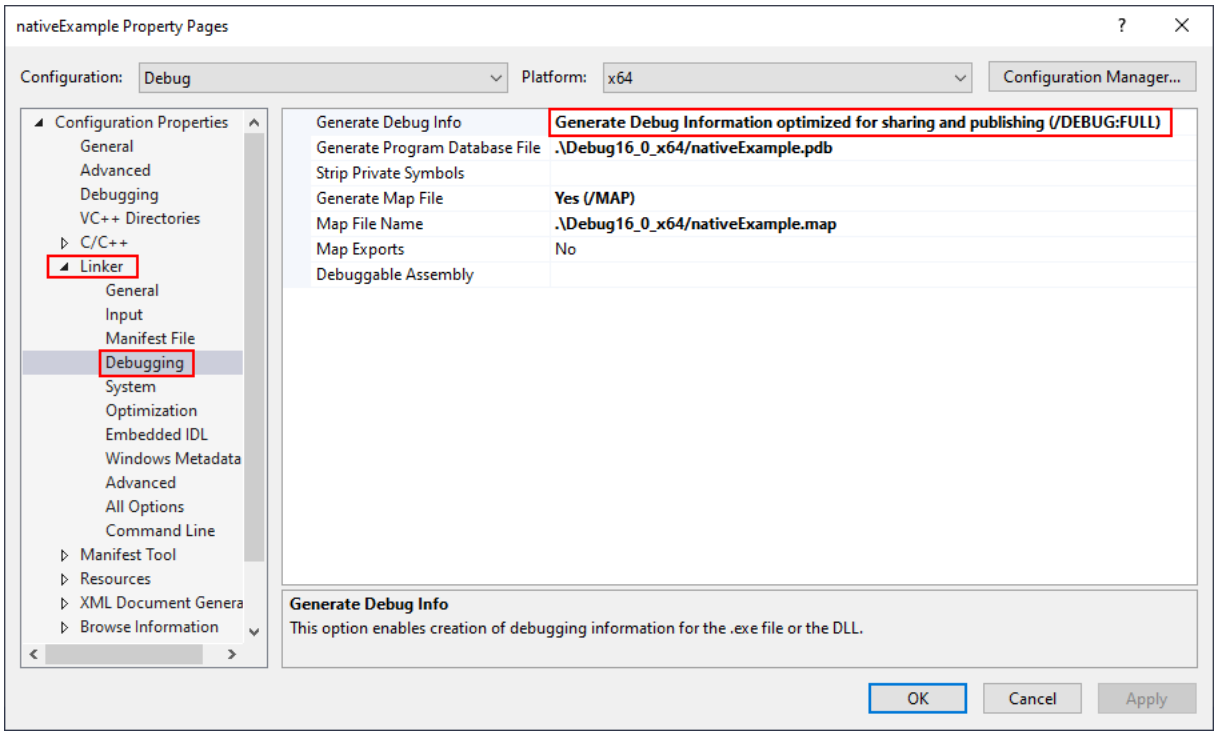
Visual Studio 2017 - 2021

Compiler Settings



Linker Settings

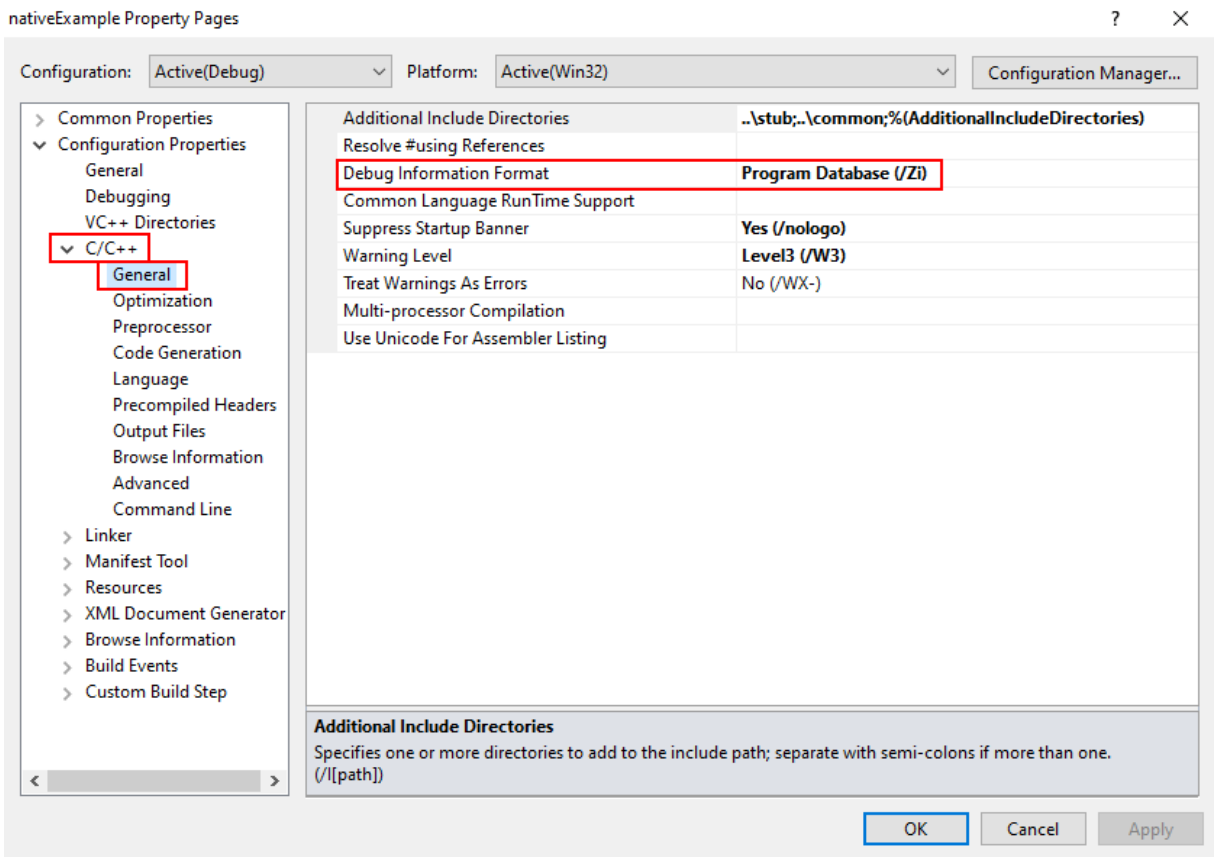
If you're building on a different machine to the machine you're working on (for example a build server), you should choose **/DEBUG:FULL**, not /DEBUG or /DEBUG:FASTLINK.



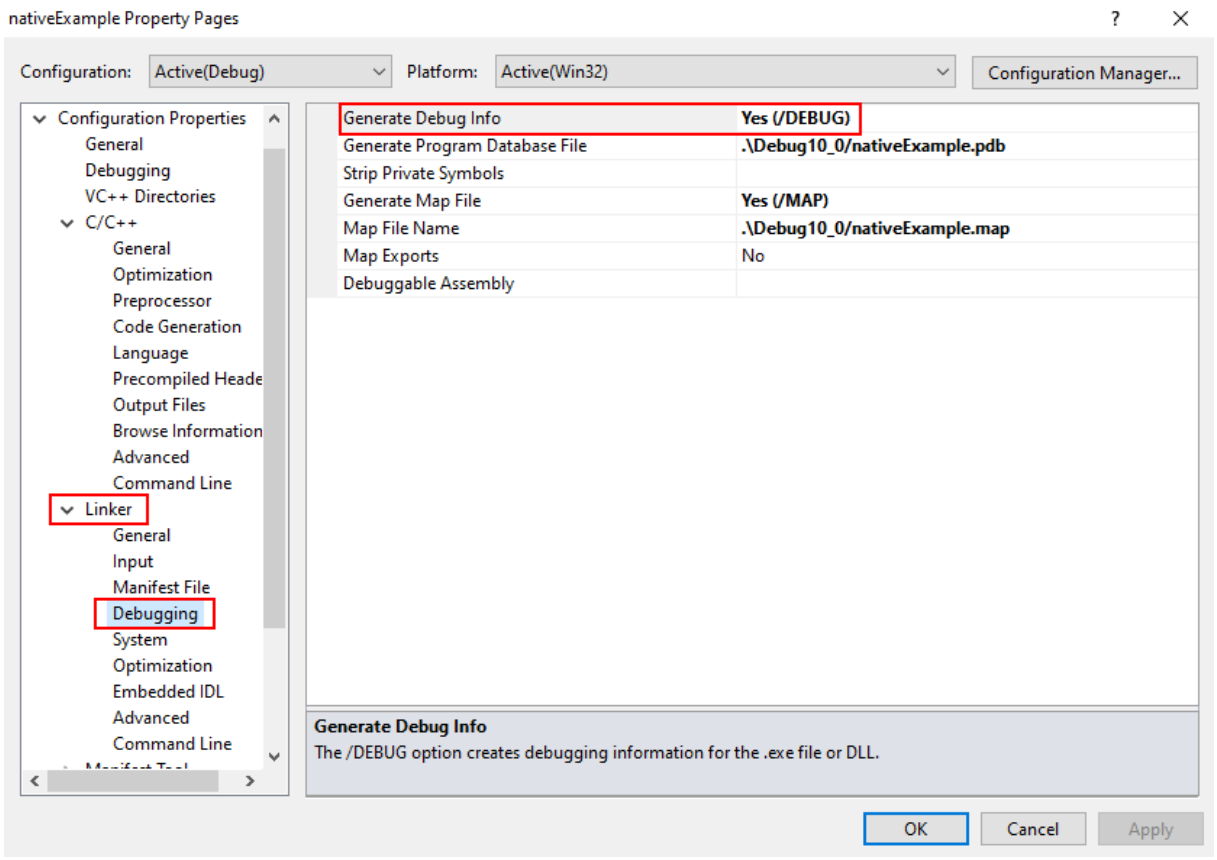
When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Visual Studio 2010 - 2015

Compiler Settings



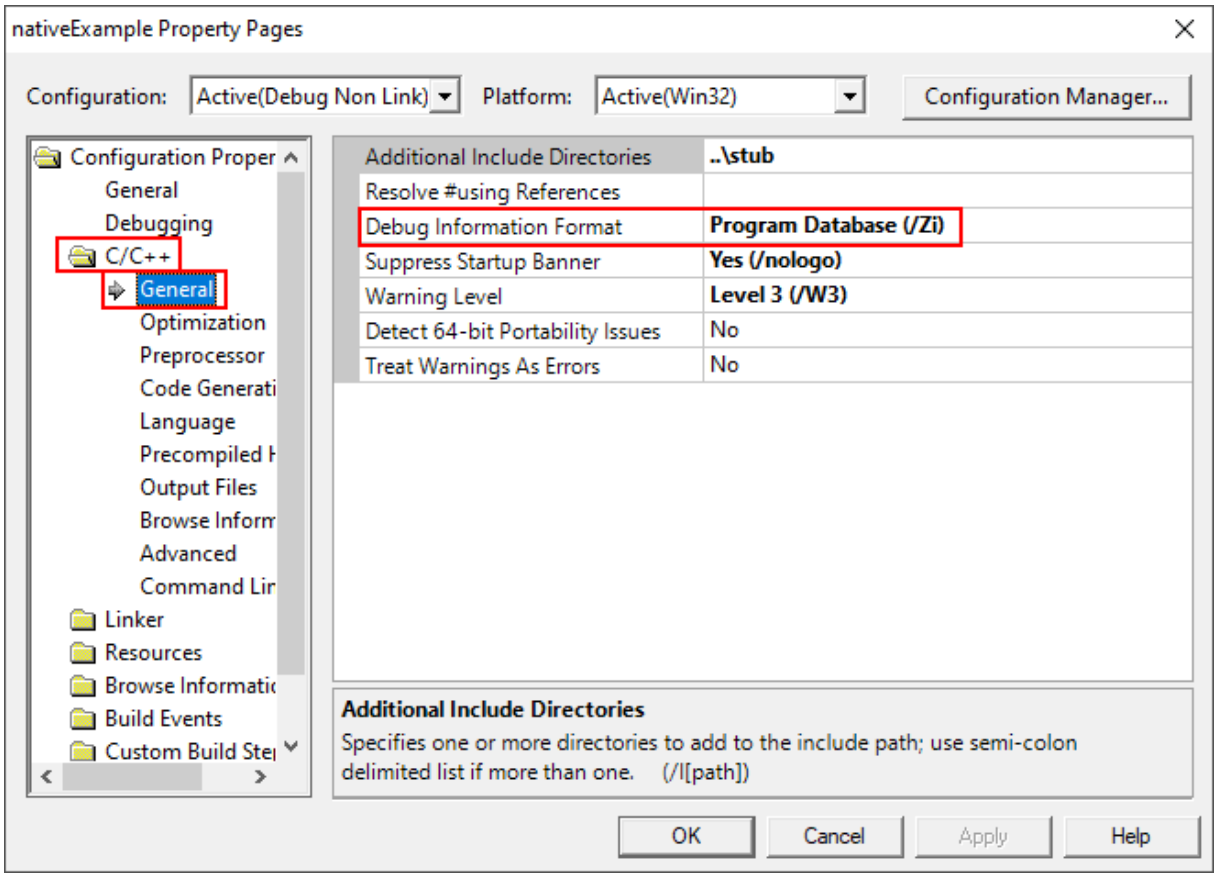
Linker Settings



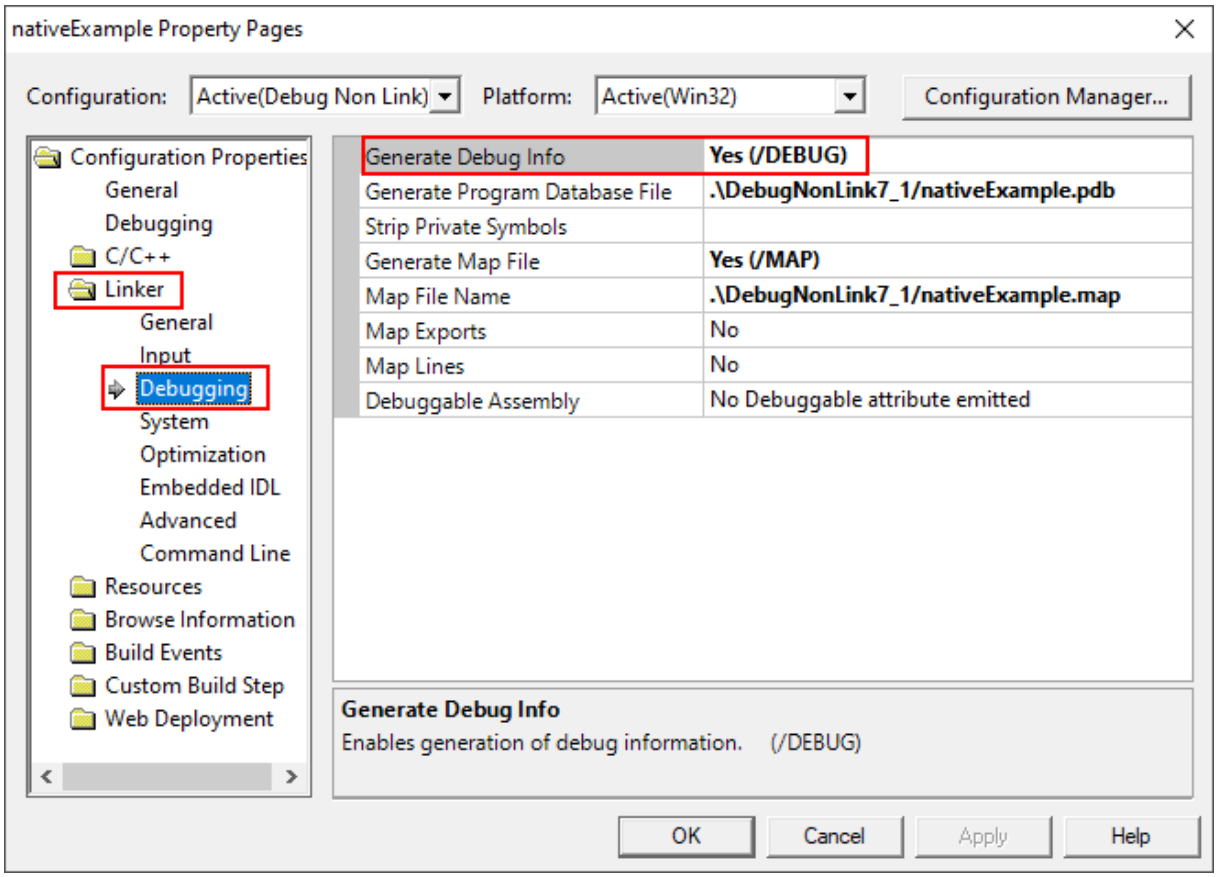
When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Visual Studio 2002 - 2008

Compiler Settings



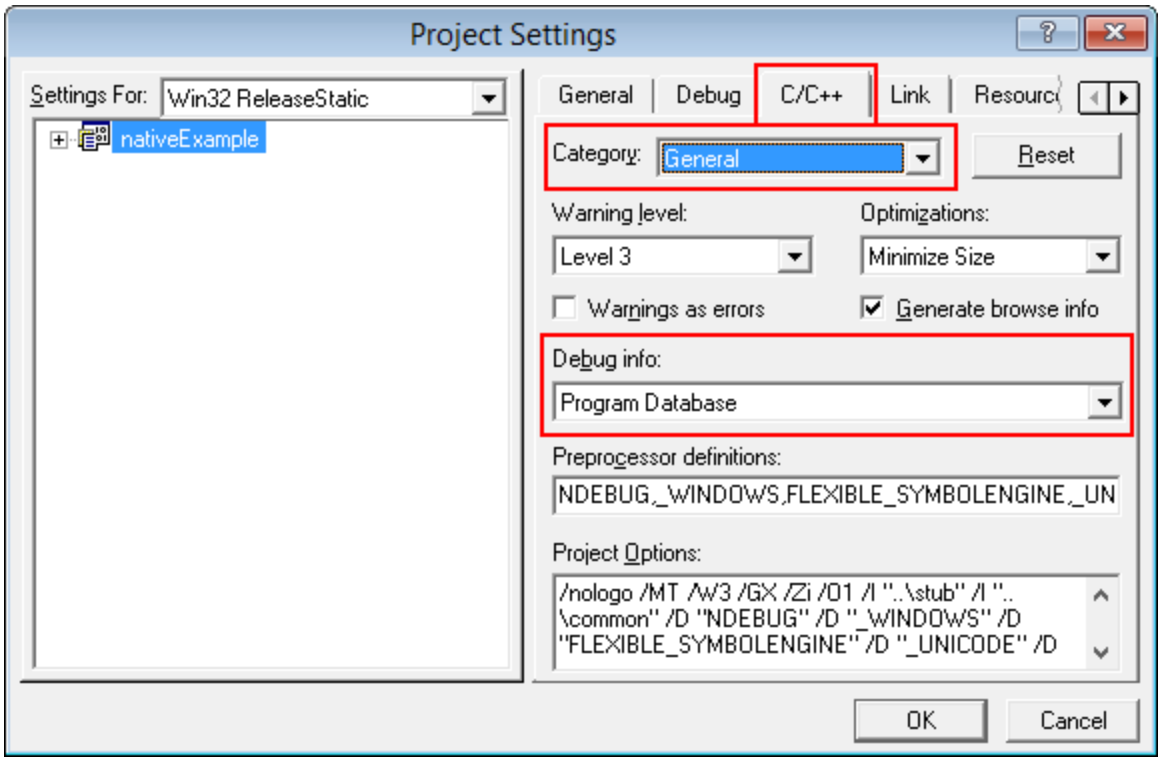
Linker Settings



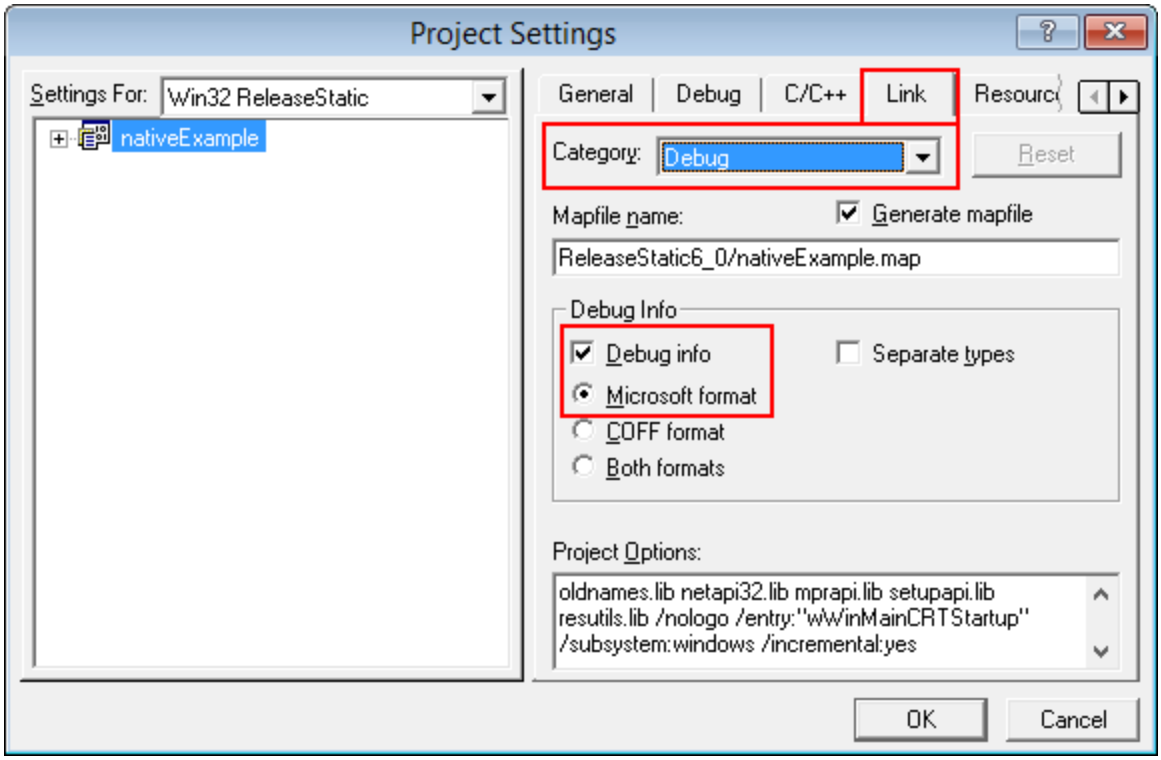
When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Visual Studio 6.0

Compiler Settings



Linker Settings



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

14.2 C++ Builder

Debug information can be provided using two methods.

- Debugging information (TDS or DWARF format)
- MAP files

Debugging Information

Debug configurations of C++ Builder projects automatically generate debug information that provides symbols, filenames and line numbers.

However the release configurations of C++ Builder projects do not automatically generate debug information. You need to configure that yourself.

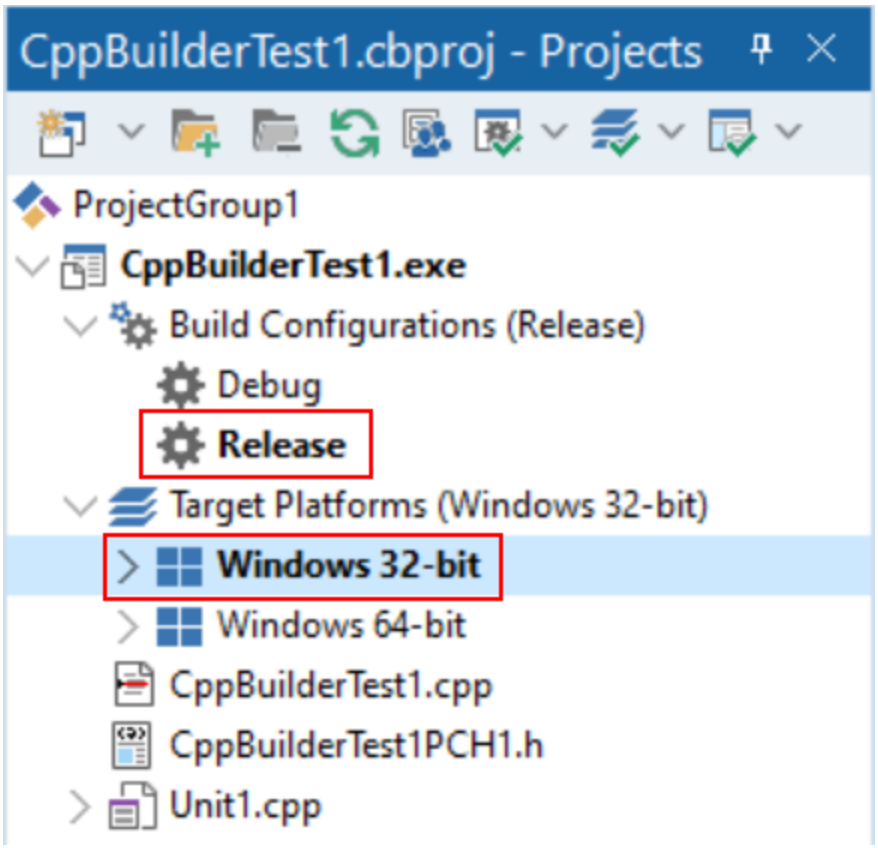
Here's how you do that. It's slightly different if you're building 32 bit applications rather 64 bit applications.

You need to set both compiler and linker settings to get debug information. Setting just one or the other will not give you debug information you can use.

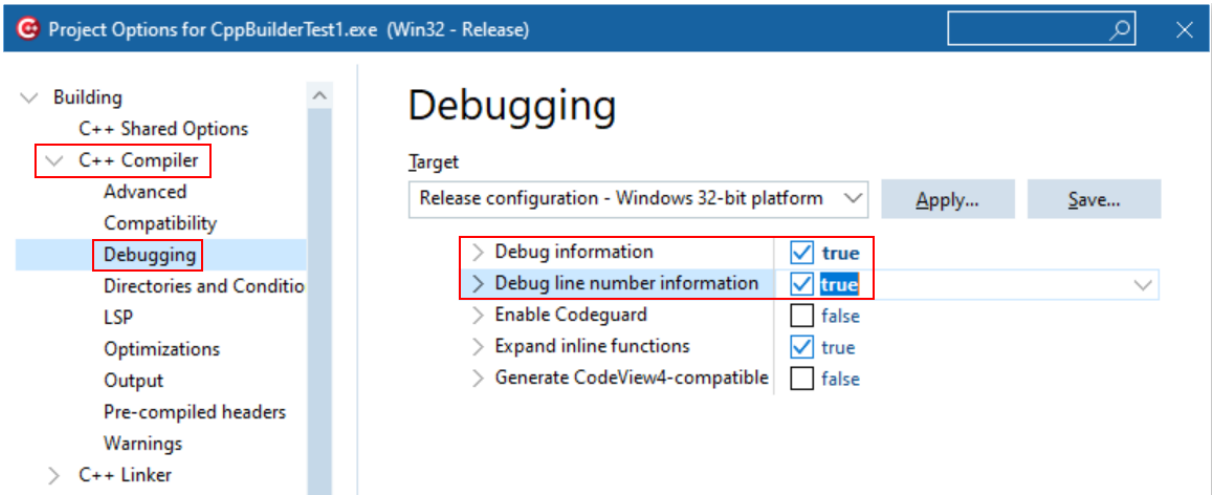
32 bit C++ Builder

Project Configuration

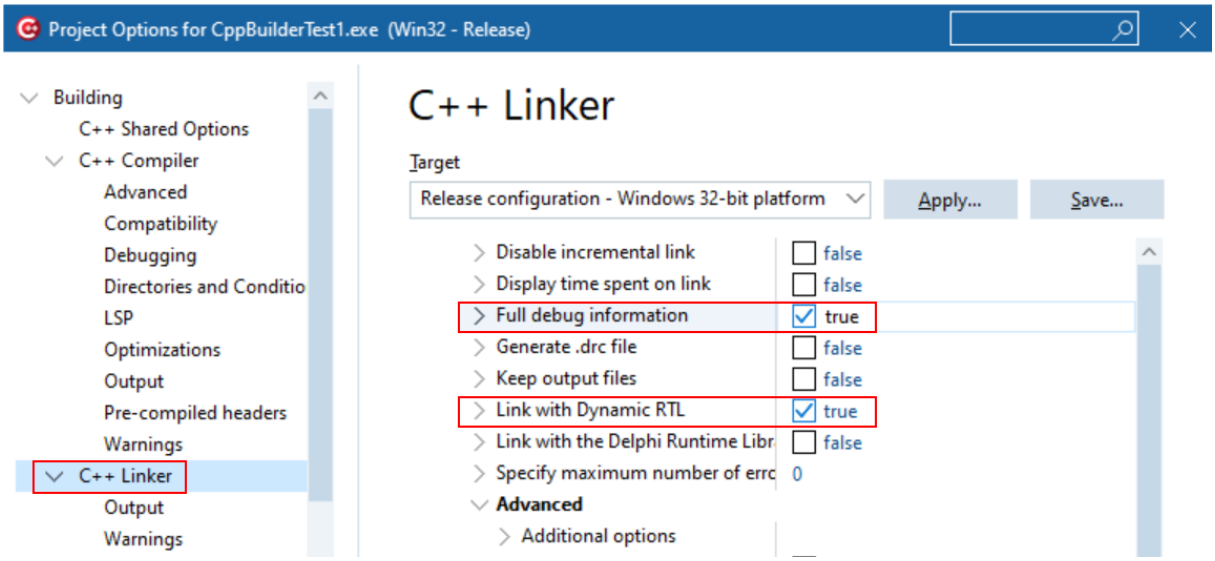
Change your project settings to target 32 bit builds.



Compiler Settings



Linker Settings

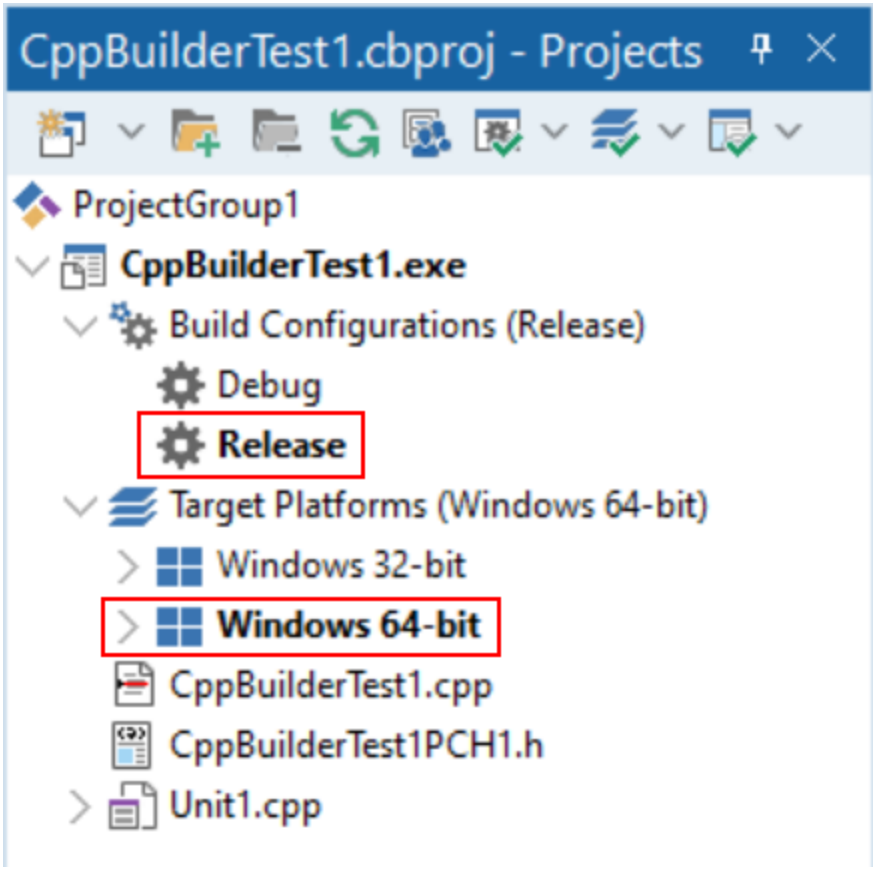


When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

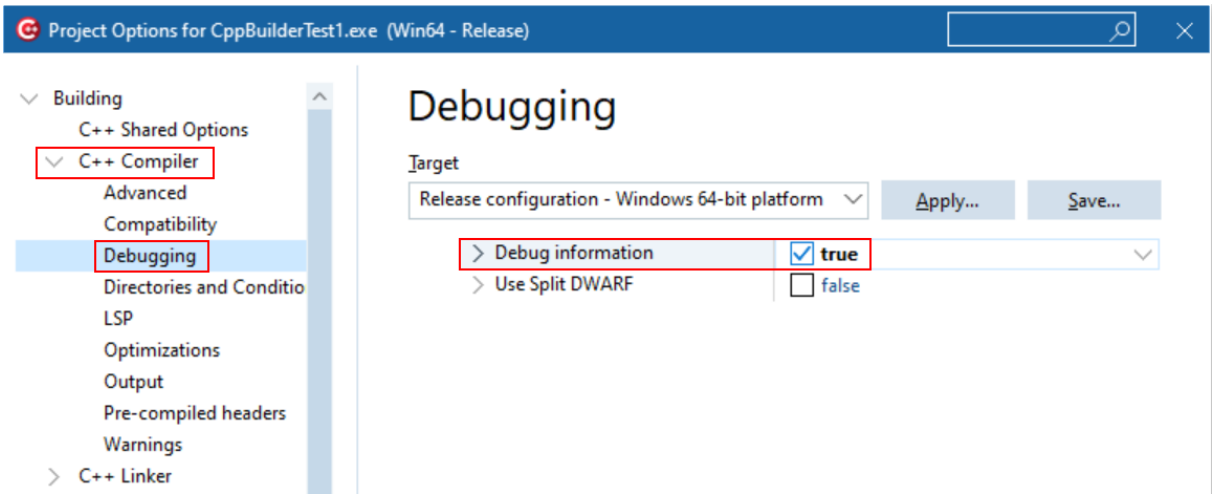
64 bit C++ Builder

Project Configuration

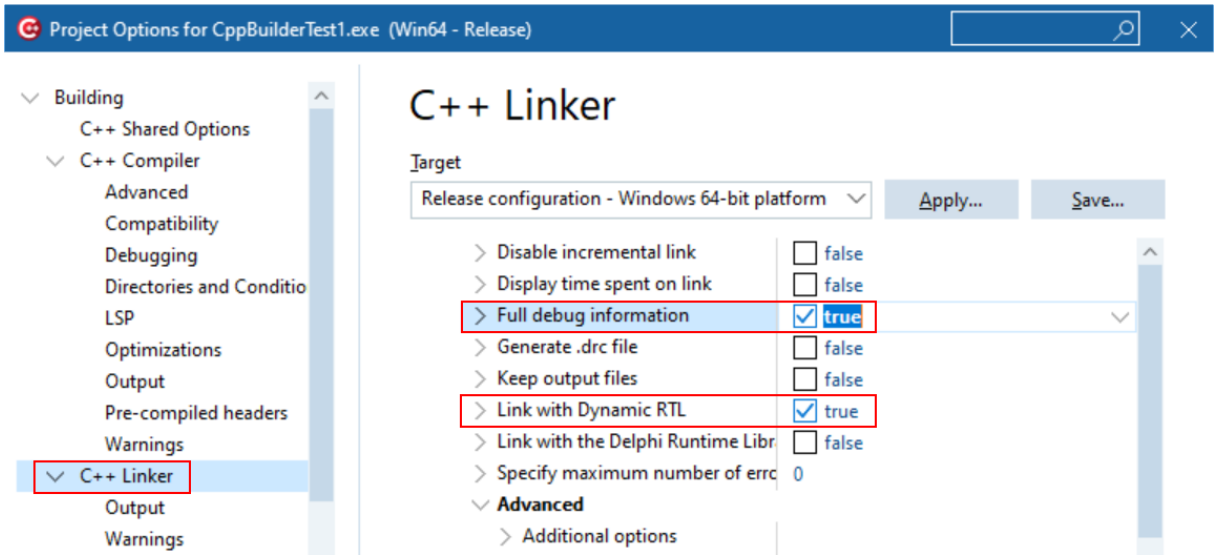
Change your project settings to target 64 bit builds.



Compiler Settings



Linker Settings



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

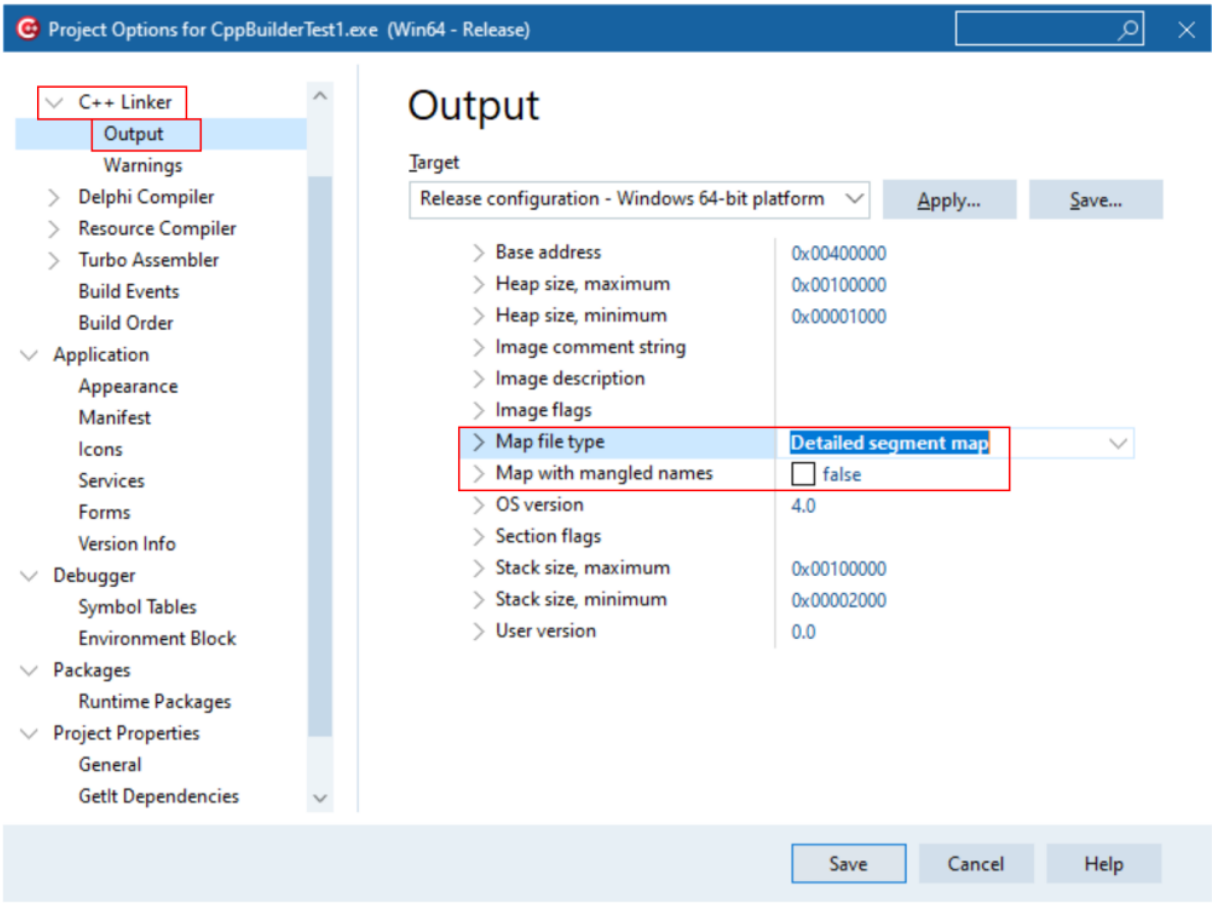
MAP files

MAP files are not generated by default. You need to enable the option to generate a detailed map file.

The method is the same for 32 bit and 64 bit C++ Builder.

Select the project configuration as shown in the Debugging Information section above, then modify the C++ Linker, Output settings.

Linker Settings



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Debugging Information or MAP files?

If you can create both debugging information and MAP files which should I use?

Memory Validator uses this information to provide symbols, filenames and line numbers in stack traces.

For this purpose it doesn't matter whether you use Debugging Information or MAP files.

14.3 Delphi

Debug information can be provided using two methods.

- Debugging information (TDS format)
- MAP files

Debugging Information

Debug configurations of Delphi projects automatically generate debug information that provides symbols, filenames and line numbers.

However the release configurations of Delphi projects do not automatically generate debug information. You need to configure that yourself.

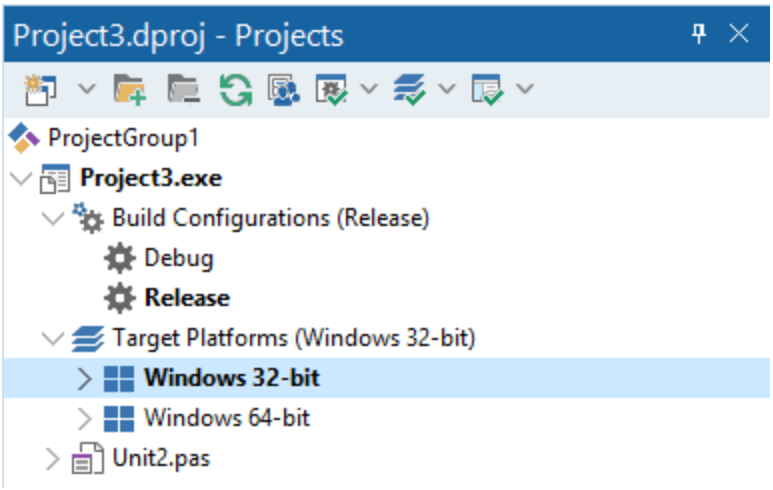
Here's how you do that.

You need to set both compiler and linker settings to get debug information. Setting just one or the other will not give you debug information you can use.

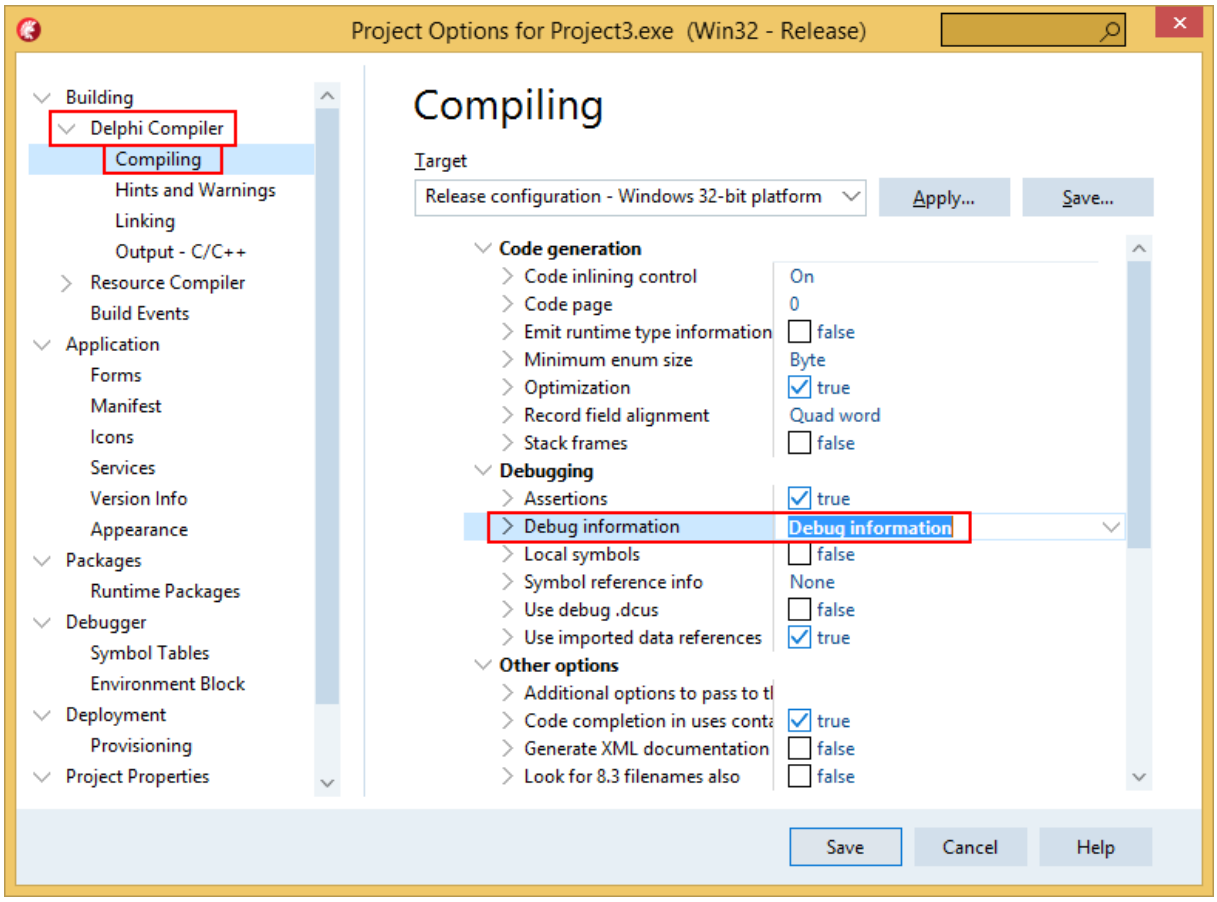
32 bit Delphi

Project Configuration

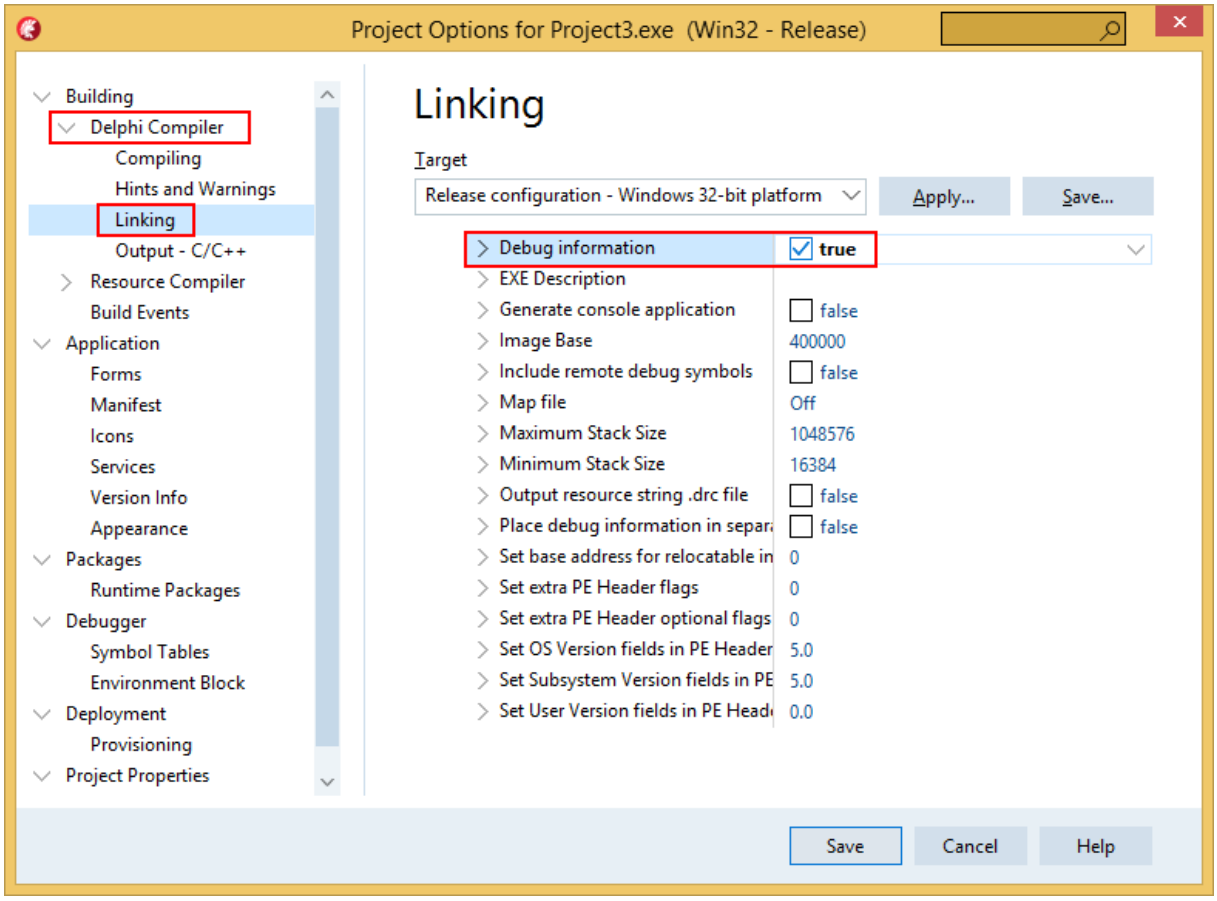
Change your project settings to target 32 bit builds.



Compiler Settings



Linker Settings

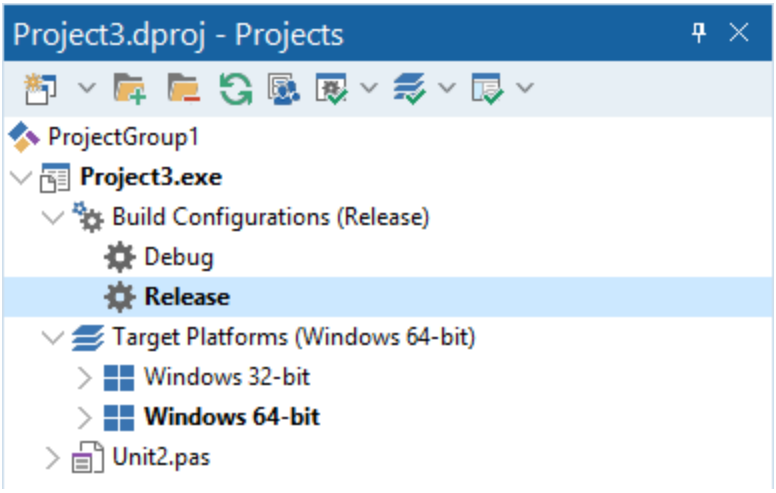


When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

64 bit Delphi

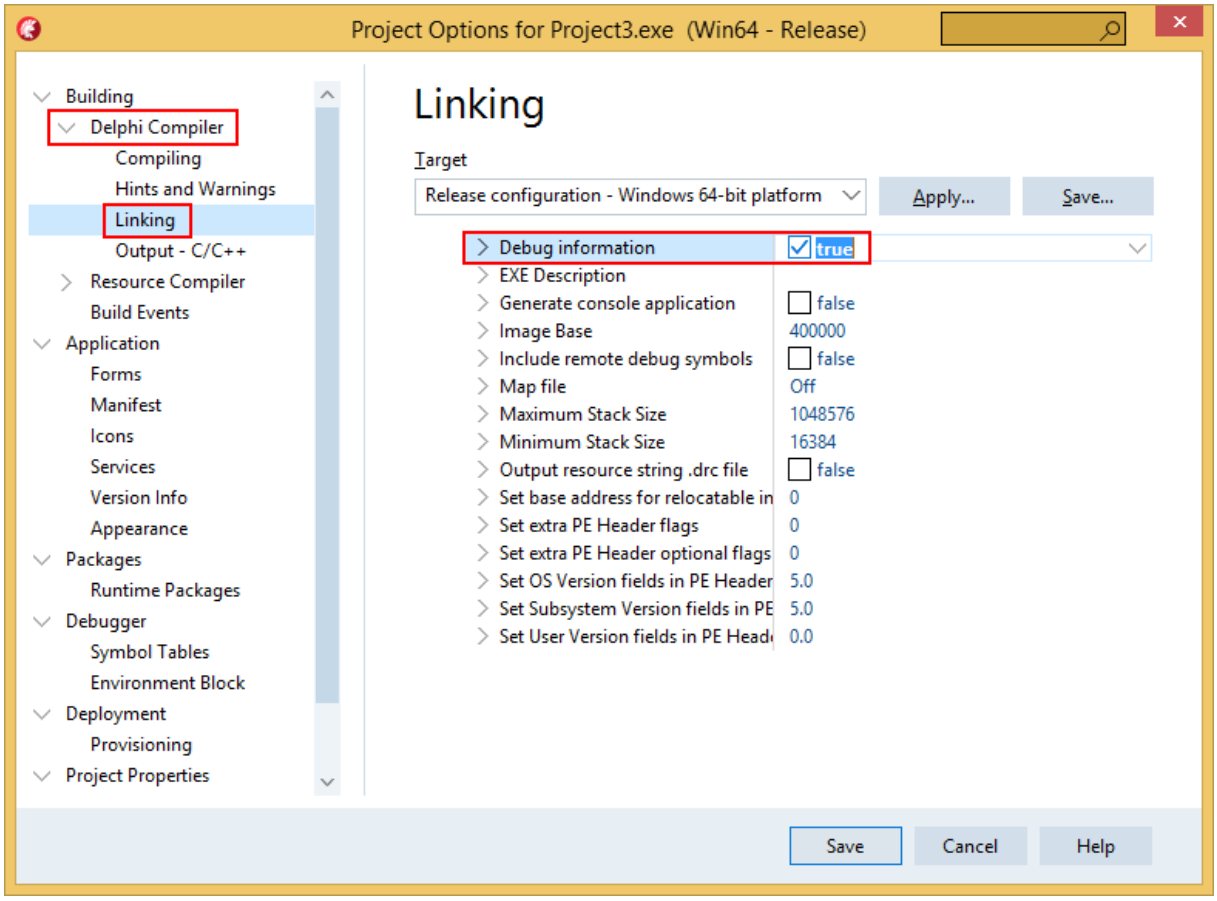
Project Configuration

Change your project settings to target 64 bit builds.



Compiler Settings

Linker Settings



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

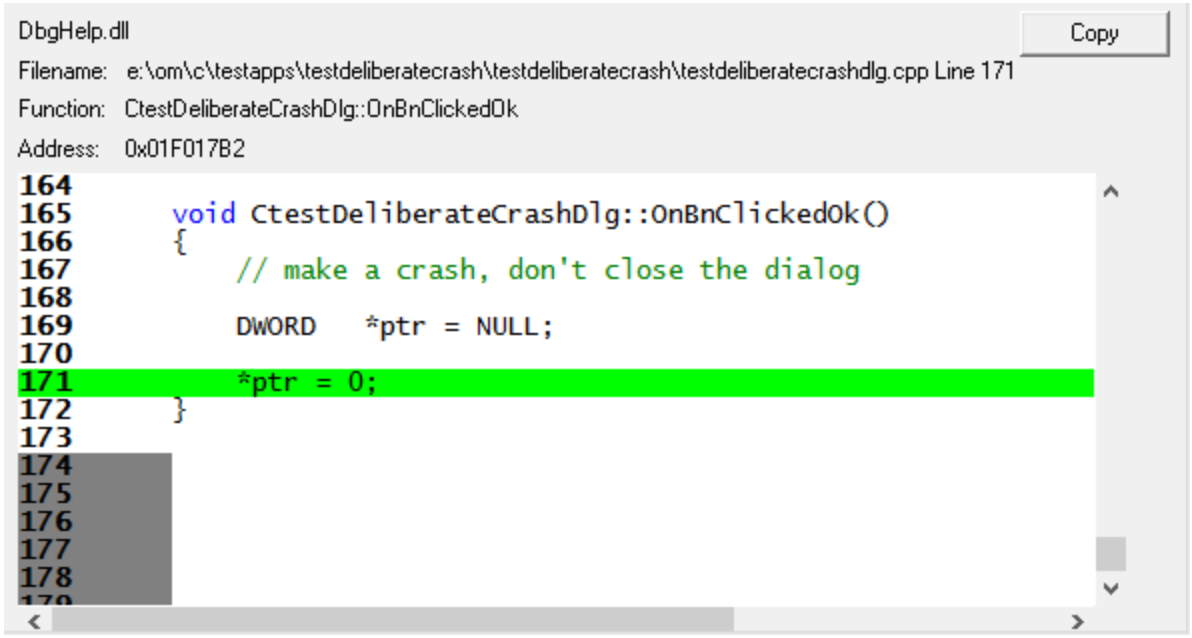
MAP files

MAP files are not generated by default. You need to enable the option to generate a detailed map file.

The method is the same for 32 bit and 64 bit Delphi.

Select the project configuration as shown in the Debugging Information section above, then modify the Delphi Compiler, Linking settings.

Linker Settings



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Debugging Information or MAP files?

If you can create both debugging information and MAP files which should I use?

Memory Validator uses this information to provide symbols, filenames and line numbers in stack traces.

For this purpose it doesn't matter whether you use Debugging Information or MAP files.

14.4 MingW, gcc, g++

The following compiler options are available if you are using MingW, gcc or g++.

-g
This is the default debug format. This will normally choose the DWARF symbol format.

-gdwarf
The DWARF symbol format.

-gstabs
The STABS symbol format.

-gCoff
The COFF symbol format. This does create a lot of unnecessary symbols, making symbol parsing slower.

14.5 Dev C++

Dev C++ uses the gcc and g++ compilers.

The following compiler options are available if you are using gcc or g++.

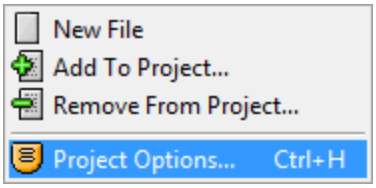
-g
This is the default debug format. This will normally choose the DWARF symbol format.

-gdwarf
The DWARF symbol format.

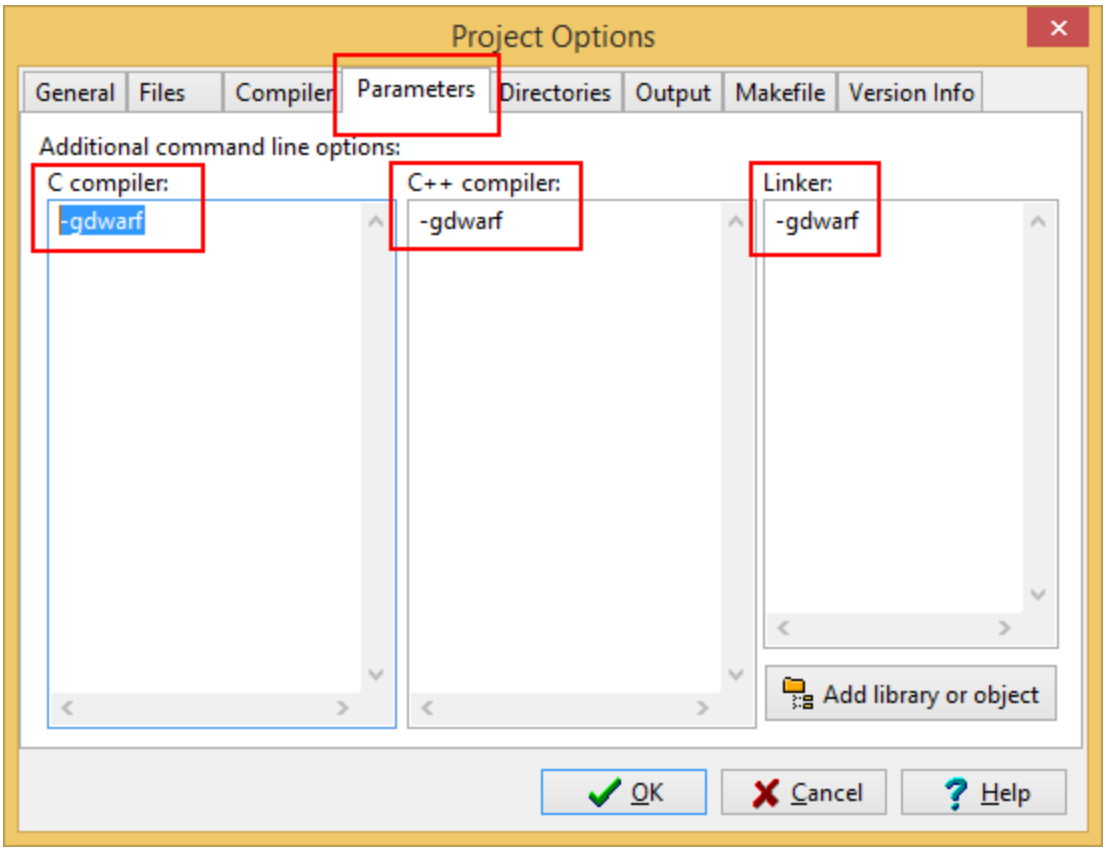
-gstabs
The STABS symbol format.

-gCoff
The COFF symbol format. This does create a lot of unnecessary symbols, making symbol parsing slower.

You can edit the compiler and linker options by choosing **Project Options...** from the **Project** menu.



Compiler and Linker options



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

14.6 Salford Software FORTRAN 95

Salford FORTRAN95 symbolic information is embedded in the .exe/.dll as COFF (Common Object File Format) information, with some proprietary extensions to Salford Software (which they have kindly shared with us).

Please consult the documentation for Salford FORTRAN95 to include debug information (including filenames and line numbers) in the COFF information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

14.7 Metrowerks

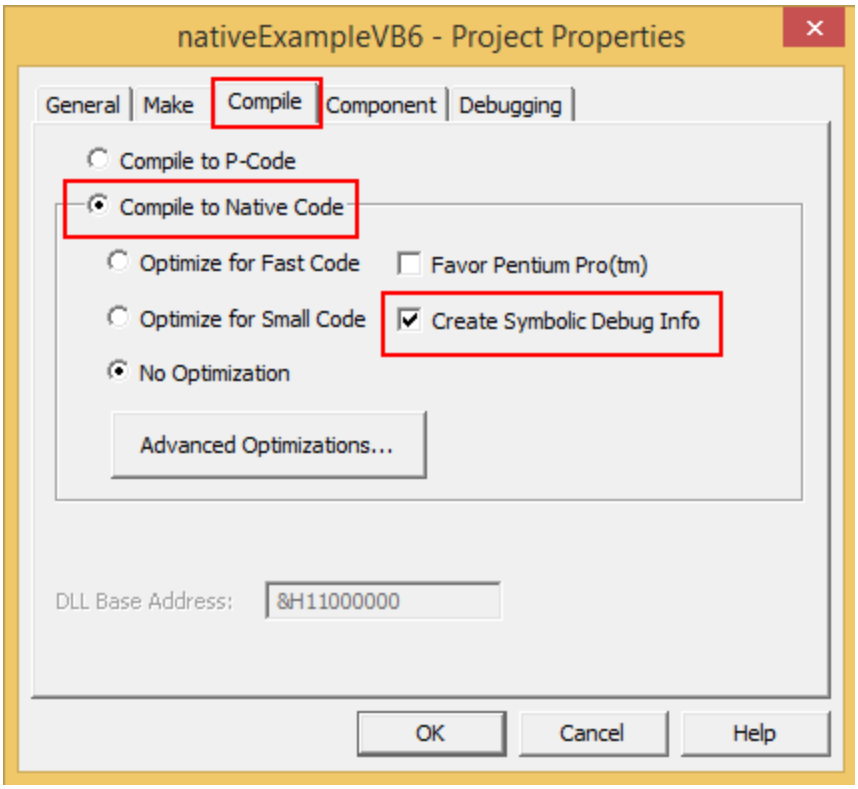
Metrowerks symbolic information is embedded in the .exe/.dll as CodeView information.

Please consult the documentation for CodeWarrior in order to include debug information (including filenames and line numbers) in the CodeView information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

14.8 Visual Basic 6

To get debug symbols for Visual Basic you need to open the **Properties** dialog box from the **Project** menu (you'll find it at the bottom of the menu).



When you have changed your project properties you need to build the application.

Go to the **File** menu and choose **Make <projectname.exe>**.

Part

15 Frequently Asked Questions

Here's a brief description about the type of question included in each of the following sections:

- General questions

  How Memory Validator works and how to do a few of the more common tasks.

- Not getting results

  Missing or unhooked data and not finding the data you expected.

- Seeing unexpected data

  The data you are finding looks wrong or is unexpected.

- Crashes and error reports

  Your program crashes with Memory Validator or Memory Validator itself has a problem.

- Performance

  Your program runs slowly and features that affect performance.

- DbgHelp

  Troubleshooting search paths for DbgHelp.dll, and finding or installing different versions.

- Extensions and tools

  Using the stublib extension facilities.

- System and environment

  Your environment on the machine you are using Memory Validator with.

- Does memory validator do...

  Common questions about the capabilities of Memory Validator.

15.1 General Questions

⊟ Do I need to use the CRT to detect memory leaks?

No, Memory Validator inspects all of

- C runtime heap (CRT)
- Win32 heaps (HeapCreate etc)
- GlobalAlloc() heap

- LocalAlloc() heap

Any of these heaps can be monitored for memory allocations and deallocations.

If using a custom heap manager of your own design (or a 3rd party design) you can provide information via the API so its memory can be tracked.

Why might Inject or Launch fail?

Not using CreateProcess

If you are launching your application with any option other than CreateProcess you are effectively using CreateRemoteThread to inject into the application you have just started running using CreateProcess.

The Inject and Wait for Application to Start functionality also use CreateRemoteThread to inject into an application.

As below, injection using CreateRemoteThread does not always work.

Common reasons for injection failure

- A missing DLL in your application

  Check your application is complete.

- The target application is a .NET application or .NET service

  Check your application or service is not written using .NET technology.

- A missing DLL in Memory Validator

  Check Memory Validator is installed correctly.

- The application may have started and finished before the DLL could be injected

  This only applies if you are *launching* the application.

- The application security settings do not allow process handles to be opened

- The application is a service and is running with different privileges than Memory Validator

  If the application being injected into is a service it is recommended that the service and Memory Validator are both run on the same user account. See help for Memory Validator for working with NT services.

Application Specific Reasons for Failure

A small percentage of applications/services will not allow any DLL to be injected into them.

The reasons for this are unknown, but our testing shows that the reason for failure to inject is a combination of application, operating system and hardware that causes an inconsistency during injection (we think it is a timing issue) that causes a failure.

Our tests show that on NT 4 about 1% of all applications fail to inject, 2% on Windows 2000 rising to 5% with Windows XP.

We expect that subsequent operating systems (Windows 2003 and Windows Vista) will have higher failure rates.

How do I name a thread?

Some features such as the Thread Filter can use thread names to make things a bit more readable.

Fom within your application you can provide a name for use by a debugger or debugging tool by using the Win32 RaiseException() API.

Add the function below to your application. This is based on an example from Microsoft. There are other examples
available on the web; some specify a buffer size of 8 characters and one terminator, others specify no strict buffer
size limit.

Show code

```
// This function is documented as being callable from outside of the thread
which is being
// named, however it appears that it works more reliably if called from within
the code of
// the thread being name, passing a threadId of -1 to indicate "current thread"

void nameThread(const DWORD           threadId,
             const char         *name)
{
   // You can name your threads by using the following code.
   // Memory Validator will intercept the exception and pass it along (so if
you are also running
   // under a debugger the debugger will also see the exception and read the
thread name

   // NOTE: this is for 'unmanaged' C++ ONLY!

   #define MS_VC_EXCEPTION 0x406D1388
   #define BUFFER_LEN         16

   typedef struct tagTHREADNAME_INFO
   {
          DWORD  dwType;       // must be 0x1000
          LPCSTR szName;       // pointer to name (in user address space)
                               // buffer must include terminator character
          DWORD  dwThreadID;   // thread ID (-1 == caller thread)
          DWORD  dwFlags;      // reserved for future use, must be zero
   } THREADNAME_INFO;
```

```
    THREADNAME_INFO      ThreadInfo;
    char                 szSafeThreadName[BUFFER_LEN];      // buffer can be any
size,
                                                            // just make sure it
is large enough!

    memset(szSafeThreadName, 0, sizeof(szSafeThreadName)); // ensure all
characters are NULL before
    strncpy(szSafeThreadName, name, BUFFER_LEN - 1);       // copying name
    //szSafeThreadName[BUFFER_LEN - 1] = '\0';

    ThreadInfo.dwType = 0x1000;
    ThreadInfo.szName = szSafeThreadName;
    ThreadInfo.dwThreadID = threadId;
    ThreadInfo.dwFlags = 0;

    __try
    {
           RaiseException(MS_VC_EXCEPTION, 0, sizeof(ThreadInfo) /
sizeof(DWORD), (DWORD*)&ThreadInfo);
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
           // do nothing, just catch the exception so that you don't terminate
the application
    }
}
```

After adding this function declaration you can call it from inside the thread procedure of any executing thread to name.

```
nameThread(-1, "example");
```

To name a thread from *outside* of the thread procedure pass the thread id instead of -1.

The example application shipped with Memory Validator demonstrates how to use nameThread.

How do I clear the symbol cache?

To clear Memory Validator's in-memory symbol cache, delete all sessions first:

- **Managers** Menu › **Session Manager** › **Delete All** › **Close**

Then flush the cache:

- **Settings** Menu › **Edit Settings** › **Data Display** › **File Cache / Subst Drives** › click **Flush Cache** button › **OK**

  Flush Cache disabled? Delete all the sessions *first*.

You may also want to disable the on-disk cache of PDB file symbols:

- **Settings** Menu › **Edit Settings** › **Advanced** › **Symbol Lookup** › deselect **Enable caching of symbol data** › **OK**

How is uninitialized memory tracked?

Memory Validator detects uninitialized memory debug C runtime heap allocations.

The debug C runtime heap initialises memory as follows

- all allocated memory with a signature byte of 0xCD
- any uninitialised stack variables with a signature byte of 0xCC.

Memory Validator hooks all constructors of C++ objects and at the end of the constructor examines the object.

Any uninitialized data bytes are reported to the user.

Note that 0xCD and 0xCC are valid data bytes but is unusual to be found them as a WORD (eg 0xCDCD) or as a DWORD (eg 0xCCCCCCCC).

Because of the chance of false detection, it is **up to the user** to determine if uninitialized data reports are correct.

The method for hooking the object's constructors is the same as is used to hook COM objects.

How are COM objects tracked?

COM objects are tracked by hooking the Win32 functions that *return* COM objects.

These objects are then queried for their `QueryInterface`, `AddRef` and `Release` function pointers.

Those functions are then hooked by rewriting the instruction stream using some proprietary code.

The hooks then examine the return values to detect the reference count or returned object.

Caveats:

Rewriting the instruction stream is not a generally recommended practice and the above method can sometimes fail, typically when the compiler optimises two functions to share some common code.

Memory Validator tries to detect when hooking will fail and refuses to hook any functions that it knows it cannot hook safely.

Some COM objects simply cannot be hooked safely - we have found that you can debug some COM objects using Memory Validator, and not others because of the internal structure of the COM objects.

About instruction stream rewriting:

The instruction stream rewriting concept has been around for years (early video games in the 1980s for the Commodore VIC 20 and Commodore 64 often rewrote their instructions on the fly to gain a

speed advantage) and is used, in varying forms, by some of the competing products and complimentary products to Memory Validator, and by Microsoft®.

I have an idea for a feature, can it be added to Memory Validator?

We have tried to add as many features to Memory Validator that we thought would be useful to our users.

In fact, every feature in Memory Validator has been used to solve problems and bugs for clients who consult us, and in our own business, so we know the features we have are useful.

However, maybe we overlooked a feature that you would find very useful, and which you cannot work out how to add to Memory Validator via an extension DLL.

We'll happily consider most ideas for new features to Memory Validator. But no Quake, FlightSim or Flappy Bird easter eggs though, sorry!

Please contact us to let us know your thoughts.

15.2 Not getting results

Memory Validator isn't collecting any data, why?

Memory Validator has many settings that control the collection and display of the data.

It is possible either the default settings or settings from a previous session are set to ignore the data you want collected.

For example, let's say you ran a session and turned off all the handle collecting functions in the Handle Allocation Hooks tab on the global settings dialog. Even with the hooks inserted into your target program the hooks would be disabled, and no information on handles would get displayed.

Here's some possible reasons for your data not being collected:

- **Program starting too quickly**

  Starting your program in **Normal** or **Idle** modes may mean part of the code in which you want to track allocations may get executed before Memory Validator has attached to the program.

  If this is happening use the Paused or Suspend modes to start the program.

- **Data collection is switched off**

  Check that the green data collect icon is *not* displayed on the session toolbar (it will be disabled if data is being collected).

Read further information on data collection.

- **Data collection hook groups not installed**

  The appropriate hook group for the data you wish to view may not have been installed in the target program when the target program was attached to.

  The hook groups installed are configured from the Collect tab on the global settings dialog.

  Enable the hook group and re-run your session.

- **The individual hooks for the function you are interested in may have been disabled**

  Check the hooks using the Memory Allocation Hooks, Handle Allocation Hooks, Buffer Manipulation Hooks tab on the global settings dialog.

  If the data hooks were installed, then the hooks will start collecting data once you enable the data hooks.

- **Data is being collected but not displayed**

  Check that the controls for the view you are using are set so that the appropriate data will be displayed.

  Some of the views have the data display controls as part of the view. The Memory tab has its control on the memory display settings dialog.

Other checks

Check that there are no filters (in Global, Session and Local filter groups) that could be suppressing the display of the data.

Check that the thread filter is not suppressing the display of the data.

Statically linked to C runtime libraries?

If your program is linked statically to the C runtime libraries, please read the before you start information.

Memory Validator isn't showing the data I expect to see, why?

A common reason for not seeing expected data is that the required hooks have been disabled.

Here's a few things to check for:

- the correct hooks to monitor the memory, handle and/or COM allocations are installed

  See the Collect tab of the global settings dialog.

- the relevant hooks are enabled

See the Hooks tab of the global settings dialog.

- the collected data is displayed - it might be being collected, but not displayed

  See the memory tab display settings dialog.

- global and session filters are not filtering the data from being displayed
- local filters (for the view being used) are not filtering the data from display
- the target program *really is* executing the code you think it is executing

I'm not seeing CRT allocations, but am seeing allocations for everything else. Why?

If you are not seeing allocations for the C runtime heap, but you are seeing allocations for all other heaps and for handles, check to see if you are linked to the statically linked C runtime libraries.

If your program is linked statically to the C runtime libraries, please read the before you start information.

➡ See also: Memory Validator isn't collecting any data, why? (above)

I can't see handles from LoadIcon(), LoadCursor() and LoadCursorFromFile(), why?

The functions `LoadIcon()`, `LoadCursor()` and `LoadCursorFromFile()` create *shared* Icons and cursors.

These objects are shared between multiple applications.

Since they can't be leaked, Memory Validator doesn't track them.

This also means that when your application has finished with them, they don't need to be destroyed.

The function I want is not hooked, why not?

We provide hooks for all the functions that we think are required to be hooked in order to give you the best tool we can to aid you in your software development tasks.

Inevitably, new APIs with new functions get released, and we continually update Memory Validator to reflect this.

However, if we missed an API or method which is important for a bug you're working on, please drop us a line!

If we think the API should be added to Memory Validator list of hooks, we'll make it happen and let you know either way.

I can't get stack traces in my release mode programs, what can I do?

Sometimes in release mode programs, callstacks of a useable length cannot be collected.

One reason for this is that high levels of compiler optimization have removed the stack frame instructions (push EBP prologue and POP EBP epilogue) and have not included adequate frame pointer omission data (FPO_DATA) in the PE file for the functions that have been optimized.

It can also happen in debug mode programs, that if extra data is placed on the stack during custom assembly calls or hooks, the DbgHelp `StackWalk()` function may fail to find the next EBP stack frame, halting the stack walk.

Memory Validator has advanced stack walking options that can walk callstacks even when DbgHelp `StackWalk()` can't.

Why can I see datatypes that give no search results?

When using the find memory dialog or the find object type dialogs you might occasionally see datatypes listed that don't produce any search results.

This is usually because the datatypes have been loaded from the file type cache.

To remove the cached datatypes you need to flush the datatype cache.

Why do some double delete callstacks show no allocation location?

Some double delete callstacks do not show the allocation location.

A few possible causes for this might be as follows:

- **Memory Validator was not collecting data when the memory was allocated**

  This could be because the memory was allocated before Memory Validator was attached to the process, or because data collection was turned off when the memory was allocated.

- **Information about the memory allocation has been discarded**

  This can happen if the **Discard stack traces for freed memory** check box is selected on the Allocation History page of the global settings dialog box.

- **The double delete is being recognized as an invalid memory location**

  Consider the following:

```
char   *ptr;
ptr = new char [3000];

// .. do some work

ptr = ptr + 10;   // pointer no longer points at the start of the memory
delete [] ptr;    // incorrect

// .. do some work

delete [] ptr;    // incorrect double delete
```

Not only is the first delete incorrect because of the altered pointer value, but there is a second delete with the same (incorrect) pointer value.

When Memory Validator looks for the associated *allocation* location, it will never find it because there are no allocations at the address "ptr + 10".

In this way, the double delete can be detected, but the allocation location can't be detected.

Why are my ordinal to symbol conversions not working?

If you have used the Ordinal Handling utility, you may still find the ordinal names have not been converted in symbol names,

Here's a couple of reasons:

- **You may not have enabled ordinal handling**

  Use the Source Parsing page on the global settings dialog to set the **Map Ordinals to function names** option

- **You haven't defined the ordinal to symbol conversion for the correct DLL in which the ordinals are defined**

  Double check the DLL for which you defined the conversions really include those ordinals.

  ➡ See the topic on Ordinal Handling

Why are some callstacks shorter than the depth I defined?

There are a couple of resons why the collected callstack may be shorter than you specified:

- **The complete callstack may simply have less entries than the specified depth**
- **Some of the items in the callstack may have been omitted to provide a more useful stack trace**

The second option sounds counter-intuitive, but here's an example - tracking a call to operator new.

For Debug builds `operator new` calls to `malloc_dbg()`, so the callstack will be something like (various functions omitted for clarity):

```
malloc_dbg()
operator new()
myFunctionThatCallsNew()
functionBlues()
functionJazz()
```

The callstack seen in this case would be:

```
myFunctionThatCallsNew()
functionBlues()
functionJazz()
```

Examining the source code to `myFunctionThatCallsNew()` would show the call to `operator new()`.

This is done so that program calls to `operator new()` are not confused with program calls to `malloc()` or `malloc_dbg()`.

In this example 2 items are removed from the callstack, but since only the specified depth of callstack was actually collected, it ends up shorter.

Memory Validator does try to correct for this by estimating the number of items that might be excluded before collecting the trace, but occasionally you'll still get a shorter trace than expected.

If you need longer callstacks collected for a particular bug you are investigating change the data collection callstack monitoring settings to collect more of the callstack, or even the complete callstack.

15.3 Not getting symbols, filenames, line numbers

Why does Memory Validator fail to load my symbols?

In a few cases Memory Validator will fail to load symbols for a DLL that you believe you have provided symbols for. This topic describes the possible causes. Please read the suggested course of action for each compiler.

Microsoft Visual Studio or Developer Studio

Symbols are defined in PDB files with the same name as the exe or dll to which it refers.

Memory Validator uses the Microsoft supplied DbgHelp.dll to perform all symbol handling activities.

Correct PDB name and location?

To ensure that the correct PDB is found to match a DLL the following must be true:

- The DLL and PDB file have the same name, except for the extension

  For example test.pdb matches for test.dll or test.exe.

- The first matching PDB file in the PDB search path has the correct checksum

  If DbgHelp finds a PDB file with a different checksum, loading symbols will fail but the search will still stop.

Verify that there are no PDB files with the same file name that are on the PDB search path, except for the PDB file you expect to be used.

You can check the DbgHelp symbol search path to troubleshoot symbol loading failures relating to the symbol search path.

Are compiler and linker producing symbols?

If DbgHelp is still failing to load your symbols, check the following:

- Your program is **compiled** to include symbol information
- Your program is **linked** to include symbol information

  Linker options are different to the compiler options

Running correct version of DLL?

Check that you are using:

- The **most recent** version of your DLL
- The **correct build** version of your DLL

  For example release DLL with release builds, debug DLL with debug builds

Checking for correctly loaded modules

When your application is running, check the modules being loaded by the application.

In Memory Validator, you can check the modules by using the Loaded Modules dialog, or by inspecting the Diagnostics tab.

You need to be sure that your application is not loading a different DLL with the same name from a different directory that is on the search path.

Correct version of DbgHelp.dll?

Try checking the version of DbgHelp.dll used by your Visual Studio installation and the version of DbgHelp.dll distributed with Memory Validator.

If the version used by Visual Studio is higher, it's possible Microsoft changed the PDB file format, making the symbols unreadable by Memory Validator.

To fix this:

- Copy the DbgHelp.dll from Visual Studio to the Memory Validator installation directory
- Remove any DbgHelp.dll from your application directory

  When Memory Validator launches an application it copies Memory Validator's DbgHelp.dll to the directory of the executable.

  This ensure that the DbgHelp.dll used is more recent than the default `system32\dbghelp.dll` which may not get updated.

  You need to find and remove these dlls - eg `c:\myapplication\debug\DbgHelp.dll` etc.

If all else fails...

Sometimes symbolic information will not load for unknown reasons.

In this circumstance, after trying the above suggestions, try changing the location in which symbols are sourced.

You could also try flushing and disabling the caching of symbols.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

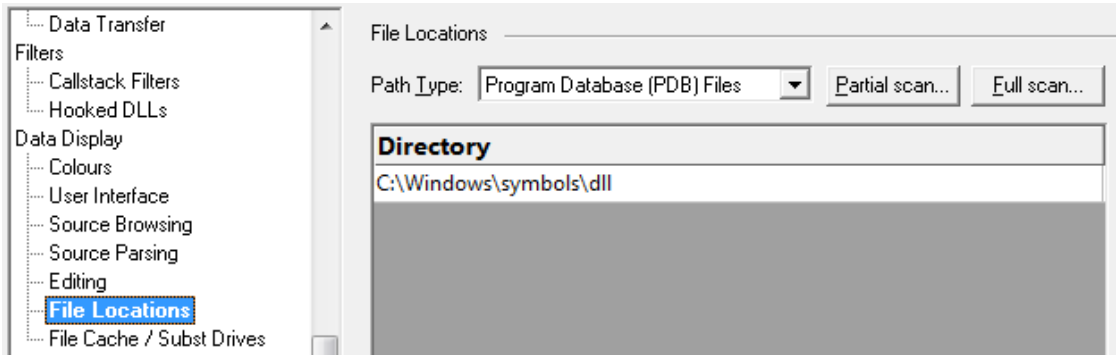
Visual Studio 2005 (8.0) and later versions

You may find that symbols for the `msvcr80.dll`, `msvcr80d.dll`, `mf80.dll`, `mfc80u.dll`, `mfc80d.dll` and `mfc80ud.dll` DLLs are not loaded.

The reason for this is that these symbols are stored in `c:\windows\symbols\dll` rather than with the DLLs themselves.

This is due to the Windows.NET Side-by-Side (WinSxS) DLL/assembly loading.

To resolve this, add the path **c:\windows\symbols\dll** to the list of paths for Program Database Files on the File Locations tab:



You may need to restart Memory Validator to get valid symbols for `MFC80(u)(d).dll` if you have already recorded a session for which you did not get symbols.

Alternatively follow the instructions in the question on how to clear the symbol cache:

Metrowerks CodeWarrior for Windows V8 / V9

Metrowerks symbolic information is embedded in the .exe/.dll as CodeView information.

Please consult the documentation for CodeWarrior in order to include debug information (including filenames and line numbers) in the CodeView information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

Salford Software FORTRAN95

Salford FORTRAN95 symbolic information is embedded in the .exe/.dll as COFF (Common Object File Format) information, with some proprietary extensions to Salford Software (which they have kindly shared with us).

Please consult the documentation for Salford FORTRAN95 to include debug information (including filenames and line numbers) in the COFF information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

MingW compiler

We recommend compiling your software with `-gdwarf` to create DWARF debugging information.

The `-gstabs` option is also supported, as is the `-gCoff` option, but -gCoff does create a lot of unnecessary symbols, making symbol parsing slower.

15.4 Seeing unexpected data

Why do I get <UNKNOWN> symbols?

Some callstacks may not have a symbol name and can display the value <UNKNOWN>.

There are several reasons this may happen:

- **The program you are monitoring has no debugging information**

  You'll need to enable debugging information in your program.

  Debugging information is controlled from the Linker Tab on your VIsual C++ project settings, and is available for Debug *and* Release builds.

- **The PDB files with the debug information can't be found**

  The program you are monitoring may have the debug information but Memory Validator can't find the debug information if it's stored in PDB files that are not in the current directory.

  Use the File Paths dialog to set where the PDB files can be found.

  If you don't have PDB files for a particular DLL, but do have MAP files, you can also set the location of these too.

- **A stack traces contain location is not present inside a DLL**

  This sometimes happens when hooks cause the program to jump to dynamically allocated memory holding the hook.

  These hooks will not have any debugging information referencing them.

- **The DLL has no filename and line-number data in the debugging information**

  This is the case for some release mode DLLs from Microsoft such as mfc42(u).dll and mfc80(u).dll.

  These only have symbol name information available, with filename and line-number removed.

  If you have this problem, you could try and get more up to date symbol information from Microsoft using the symbol server support page in the global settings dialog.

  Note that this will only work if the symbol server symbols *do actually contain* filename and line-number information - they might not!

If none of the above solves your problem and **all** symbols are still displayed as <UNKNOWN> please drop us a line. We have found that newer versions of Visual Studio sometimes change the debug information format and need a newer version of DbgHelp.dll. The version of DbgHelp.dll that is shipped with Memory Validator is compatible with Visual Studio.net and all previous versions of Visual Studio.

I get false reports of memory leaks from VariantCopy(), why?

The documentation for `VariantCopy()` states that the operating system does not know the reference count for any object in a Variant that has the VT_BYREF flag. As such VariantCopy() cannot modify the reference count in the destination and source arguments.

Since the reference count for these objects with the VT_BYREF flag is unknown, Memory Validator assumes a new object is deallocated prior to the copy (destination variant is cleared) and that a new object is allocated in the destination.

For the cases where the operating system does *not* do this, but simply adjusts reference counts, leaks will get reported where there are none.

The MSDN documentation indicates this can only happen for COM objects and not for BSTRs.

Memory Validator only tracks BSTRs in `VariantCopy()`. We've not *seen* false reports of memory leaks for `VariantCopy()` but this question is here in case you do!

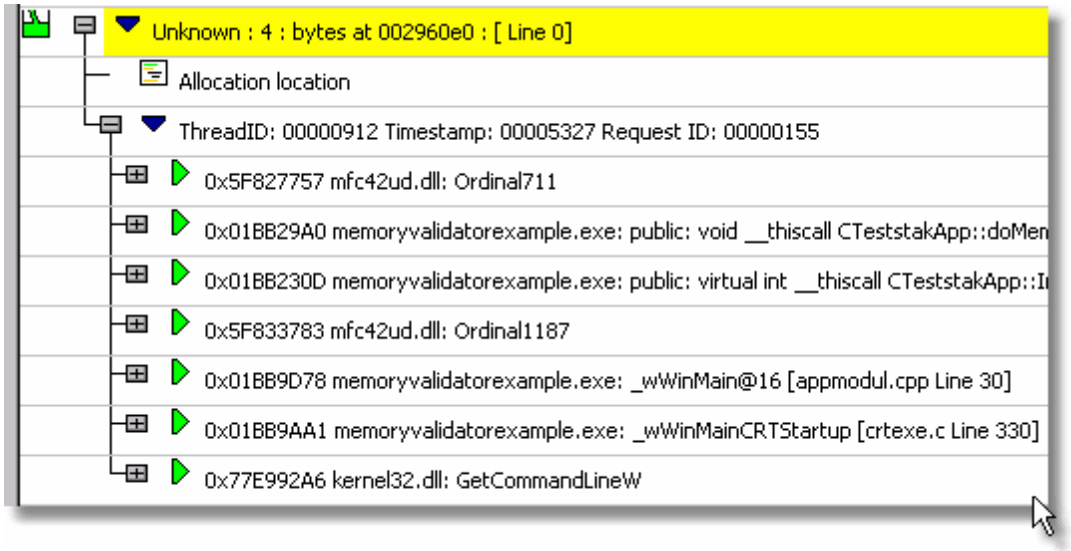
Some symbols are displayed as Ordinalxxx, why?

If Memory Validator can't find debug information for DLLs that have their functions exported as ordinal values, the functions are named `OrdinalXXX`, using decimal number of the function.

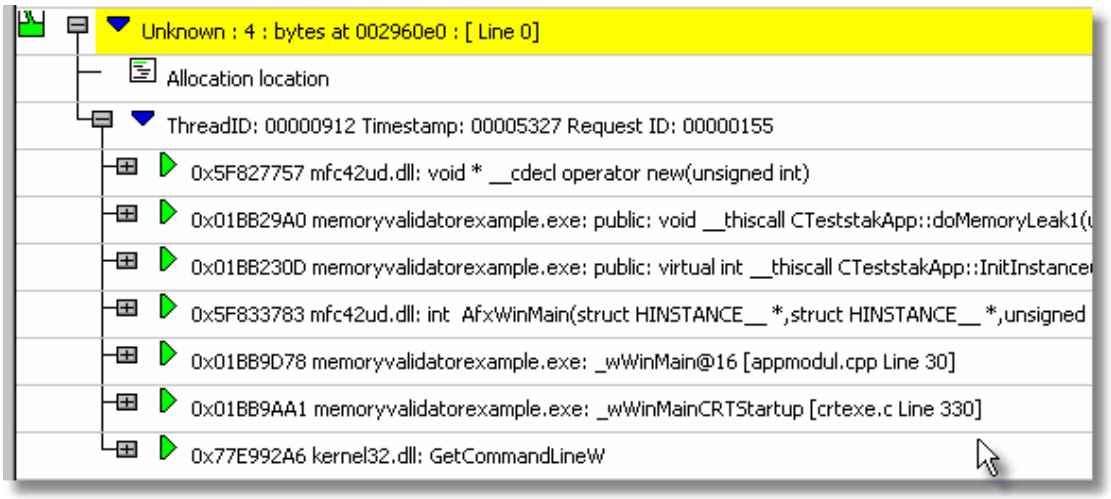
These ordinal values *can* be displayed as function names, but only if the linker definition (.def) file that refers to each relevant DLL is known.

You can define the Ordinal Handling, but don't forget to also select the **Map Ordinals to function names** check box on the Source Parser page of the global settings dialog.

Here's an example of Ordinal function names without debug information: `Ordinal711` and `Ordinal187`



And here's the same thing without debug information but after ordinal to symbol conversion, showing `__cdecl operator new()` and `int AfxWinMain()`



I'm seeing unusual trace data

The following information applies to older operating systems.

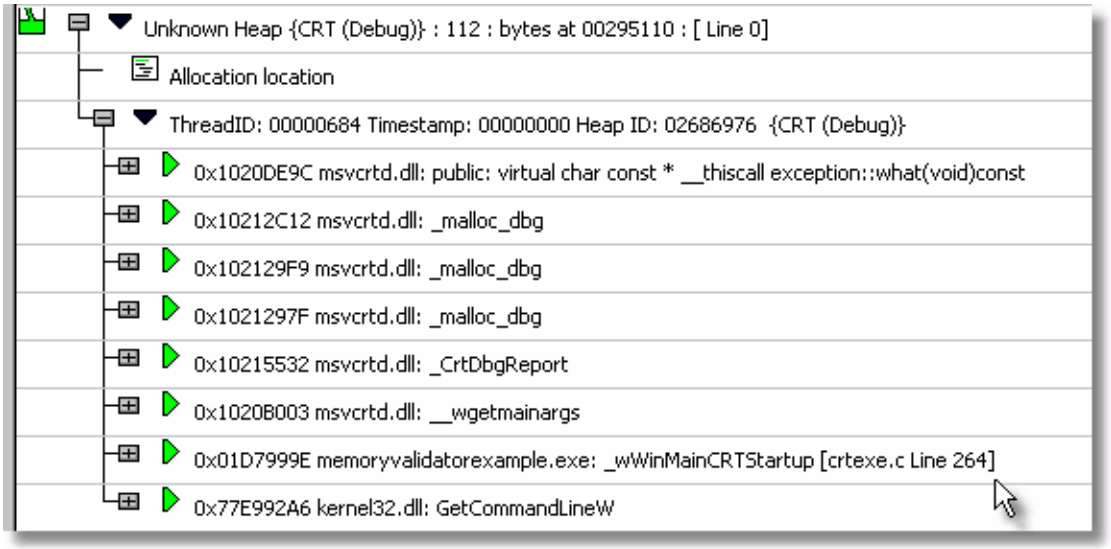
Some machines have been found to exhibit unusual behaviour during the startup phase of the target program.

The result is a few memory allocations from the CRT that appear to come from an exception handler.

We have seen this behaviour on a Windows 2000® machine that, for unknown reasons meant neither Memory Validator nor Visual Studio could not find debug information for msvcrtd.dll and mfc42ud.dll despite it being available.

We suspect that the debug information, although present, was in some way not the correct debug information for the respective DLLs. Probably this was a configuration issue for the machines, caused by incorrect installation of a service pack, platform SDK, or other SDK.

Here's an example of one of these unusual data traces:.



All of these data traces happen at program startup.

This behaviour has been observed on one Windows 2000® machine, configured as a laptop computer.

This information is provided now only so that users do not get confused by this extra data that is collected by Memory Validator in this circumstance.

Even if you see unusual traces like this, Memory Validator will otherwise still continue to perform correctly.

What is address 0x006d0065?

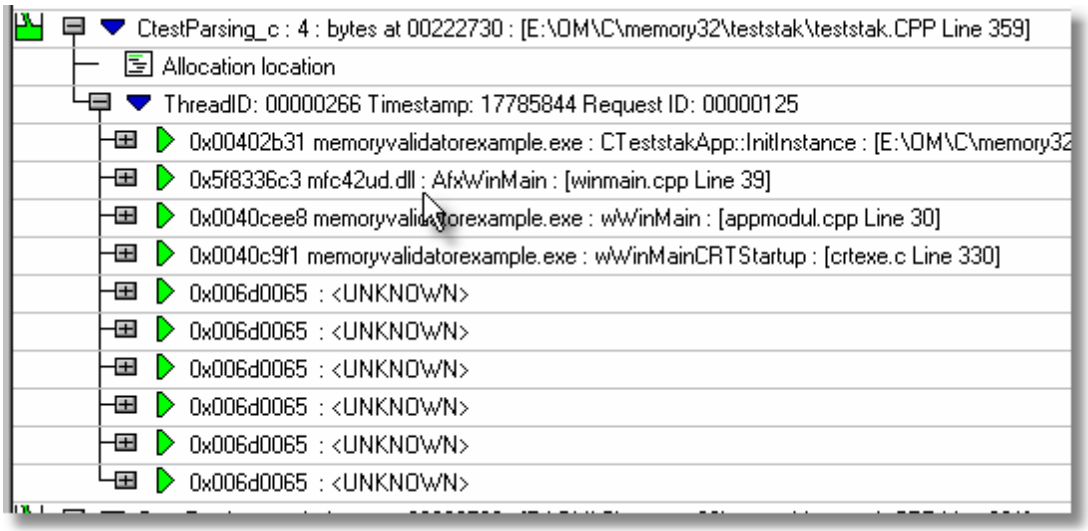
This is relevant to one version of the VisualStudio.net DbgHelp library that sometimes does not correctly identify the end of a stack walk.

When this happens, a stack trace can have numerous addresses of value 0x006d0065 tacked onto the end of it, even if the stack walking callback informs DbgHelp that the address is not valid.

This bug will not affect Memory Validator. All stack traces shown will be valid. A few stack traces may have extra data, but no data will be missing.

We don't filter these addresses out, in case a valid DLL does get loaded and uses this address space, producing symbols for this address.

Callstacks that contain this error look like this:



As you can see from the image, the program started at the UNICODE entry point wWinMainCRTStartup, so there should be no symbols (other than GetPriorityBoost or other kernel32.dll symbols) after this entry.

Subsequent versions of the DbgHelp.dll from Microsoft fix this bug.

15.5 Crashes and error reports

The program I'm trying to validate keeps crashing, why?

The following assumes your crash is one that only happens when using Memory Validator.

Here's a number of scenarios in which your program might crash:

- **COM Reference Counting hooks are installed**

  These hooks rewrite the instruction stream, which sometimes doesn't work because some object code produced by the optimising compiler, or hand written assembly code shares common routines.

  Try turning off the COM Reference Counting hooks and trying again.

- **Uninitialised Data hooks are installed**

  These also rewrite the instruction stream (see above).

  Try turning off Uninitialised Data hooks and trying again.

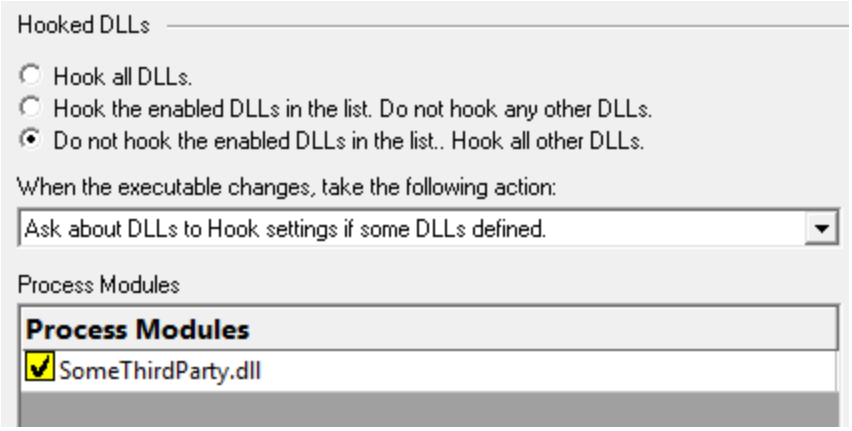
- **Stub or User Interface extensions are being used**

  Remove all Stub extensions and User Interface extensions and try again.

- **Third party DLLs are using system wide hooks**

  Some DLLs from third party vendors use system wide hooks and do not interact with Memory Validator and the target program very well.

  If you can identify such DLLs, prevent them being hooked by adding the DLL name to the Hooked DLLs page of the global settings dialog as in the example below.



- **Third party DLLs are using global hooks**

A global hook DLL from a third party vendor could be adversely affecting Memory Validator when hooking your program.

Read about handling global hooks on the Global Hooks page of the settings dialog.

Judging by multiple independent error reports, we believe there may be an incompatibility between Memory Validator and the global hooks that come with the Matrox G400 and the Matrox Millenium II PCI video cards released in the late 1990's.

- **There may be a bug in Memory Validator**

  It happens. We've tried to make Memory Validator as robust as possible, but bugs and new scenarios do occur.

  First, ensure that the crash never happens if you are not using Memory Validator.

  Second, check all the suggestions above.

  Then drop us a line sending details of the error and we'll try to reproduce the crash with a view to fixing any bugs found in as timely a manner as possible.

My program crashes in Release mode, but not in Debug, why?

Possible causes of release only crashes include:

- **Uninitialised data**

  In debug mode, uninitialized data on the heap is filled with the value 0xCD.

  This means that any code testing for TRUE or FALSE, always chooses TRUE when it reads uninitialized data in debug mode.

  In release mode, the potential for uninitialised data to contain 0x00 means there's a 1 in 256 chance the value is FALSE, leading to hard-to-reproduce bugs.

  Read more on how to detect uninitialized data using Memory Validator.

- **An MFC Message Map bug**

  In debug mode, errors made when typing function prototypes for functions called by the MFC message maps cause no problems.

  The same code, when run in release mode can corrupt stacks and crash.

  Read about how to detect MFC message map errors using Memory Validator.

- **Data Corruption**

  The debug and release CRT heaps are very different in structure.

  The debug CRT heap has a lot of extra data in it, whereas the release CRT heap is compact.

A data corruption bug in debug mode may go unnoticed and only show up in release mode, or vice versa.

Data corruption bugs are unpredictable, alter data values; damaging data structures used to hold your data; or damaging the integrity of the CRT heap.

Read more about finding data corruption using Memory Validator.

Memory Validator gives an Unrecoverable Error

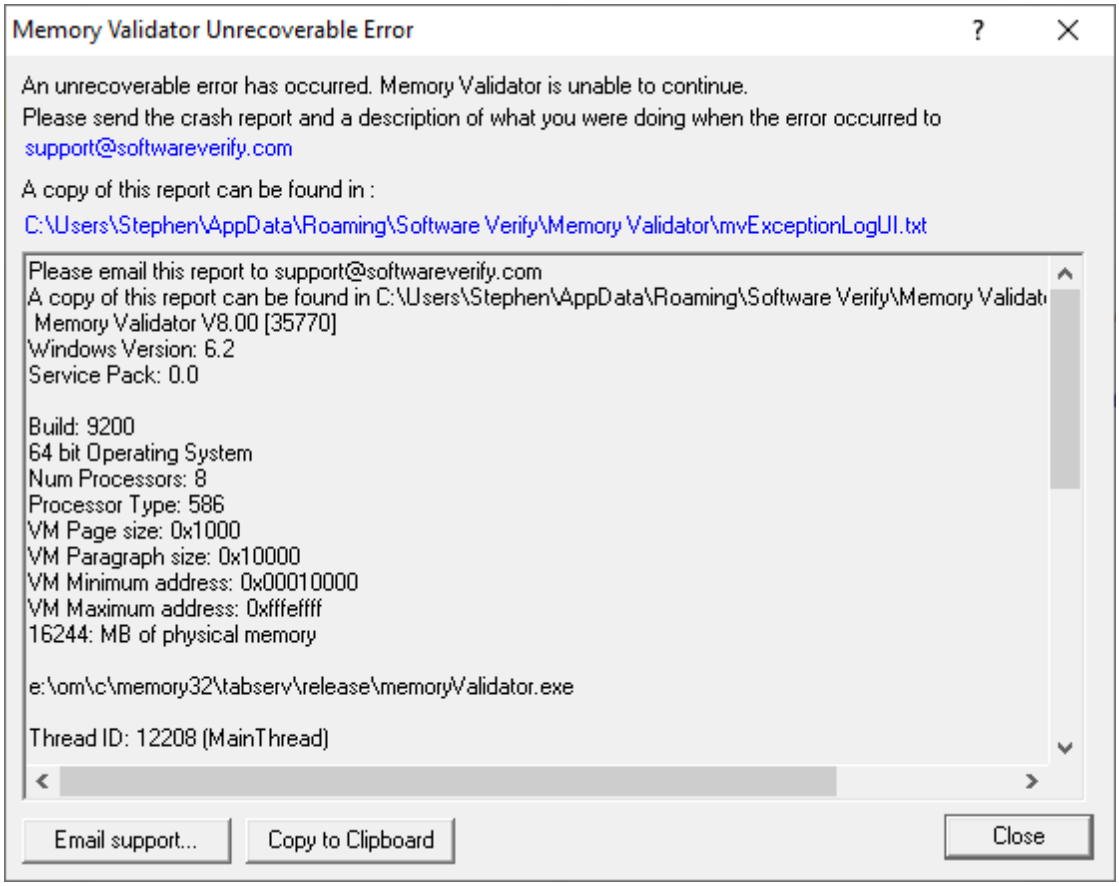
The Memory Validator Unrecoverable Error dialog is displayed when an unexpected internal error means Memory Validator cannot continue to execute.

A stack trace and register dump is shown and you can **Copy to Clipboard** so that the data can be sent to us with a description of the activities that caused the error.

We'll aim to fix any problems in as timely manner as possible.

The data shown in the dialog is also written to `c:\users\<username>\AppData\Roaming\Software Verify\Memory Validator\mvExceptionLogUI.txt` .

The picture below shows an artificial exception report for a stack overflow error.



mvExceptionLog.txt?

In the event of a crash in the Memory Validator user interface, the file `c:\users\<username>\AppData\Roaming\Software Verify\Memory Validator\mvExceptionLogUI.txt` contains information that identifies where Memory Validator was executing when Memory Validator crashed.

In the event of a crash in the target program, the file `c:\users\<username>\AppData\Roaming\Software Verify\Memory Validator\mvExceptionLog.txt` contains information that identifies where Memory Validator was executing when the target program crashed. This crash may have been caused by Memory Validator's instrumentation or by an error in the target application.

The file contains a stack trace and register dump and is the same information that is displayed in the Unrecoverable Error dialog when a crash occurred.

The file contains only the data for the most *recent* exception.

15.6 Performance

My program is running slowly with Memory Validator, why?

There are many reasons why your program may run slowly when used with Memory Validator, but they largely boil down to two types of problem:

Collecting too much unwanted data

There are many options to enable you to turn off data collection for data that is not important. some of which are turned off by default.

Turning off unwanted options prevents Memory Validator from spending time examining data you don't want collected:

- If not trying to isolate memory corruptions, turn off buffer checking
- If not trying to isolate uninitialized data, turn off uninitialized data detection
- If not trying to detect handle leaks, turn off collection of all handle related hooks
- If not trying to detect leaks in GlobalAlloc, LocalAlloc and HeapAlloc, turn off the matching memory hooks
- If not trying to detect CRT leaks, turn off CRT leaks
- If you don't need complete callstacks, collect only the part of the callstack that is interesting to you

  Depending how deep your programs callstacks get, this can have quite a dramatic impact on performance

Collecting data in a tight loop

If your program is still running slowly, it may well be because it's allocating many blocks of memory in a tight loop.

When this happens, Memory Validator gets swamped with the sheer volume and rate of data it needs to track, and symbols (for the callstack) to resolve.

When the program exits the tight loop, the program performance will return to more normal speeds.

Often this is a sign that the target program could be improved by redesigning its memory allocation strategy.

Examining the statistics on the objects view will give you an insight into the number and frequency of allocations being made.

Which data items have the greatest performance impact?

This depends largely on the target program, but some generalisations below are based on a variety of programs between 10,000, and 2,000,000 lines of code.

In order of greatest performance impact first:

- **Buffer overrun detection**

  If used, then this can have quite a big hit, but only If your program uses the C (and Win32 shell) string functions.

  For example a lot of string processing during startup can slow things down until the program is ready.

  However, buffer overrun detection is probably not used that much.

- **Uninitialized data detection**

  By its very nature this can have a high overhead, but is dependent upon the data it is examining.

- **Memory Allocation tracking** (CRT, Win32 heap etc)

  The memory allocation tracking has a low overhead unless there are large numbers of allocations in very tight loops.

- **COM object tracking**

- **Handle tracking**

  The handle tracking functions produce very little performance overhead even on very large programs.

We don't give a suite of % impact figures for each feature as they can be misleading, but in some typical examples, we found

- with all options enabled, a program launched and ran in 90 seconds
- with the uninitialized data and buffer tracking disabled it took 30 seconds

meanwhile:

- a competitor application to Memory Validator took 40 minutes and then usually failed!

At the end of the day, the performance change will be always in relation to the data generated by the target program, and no two programs are alike.

Even the size of the program is not a great indicator: for example we tested a 2,000,000 line CAD program and a 300,000 line web authoring program.

The larger program started up with Memory Validator in much less time simply due the nature of the work each of the programs was doing during startup.

Our suggestion:

If in doubt about the performance impact of Memory Validator we suggest you simply try it on your product and see.

We hope you'll be favourably impressed compared to our competitors!

15.7 DbgHelp

Why does Memory Validator fail to load my symbols?

In a few cases Memory Validator will fail to load symbols for a DLL that you believe you have provided symbols for.

This topic describes the possible causes. Please read the suggested course of action for each compiler.

Microsoft Visual Studio or Developer Studio

Symbols are defined in PDB files with the same name as the .exe or .dll to which it refers.

Memory Validator uses the Microsoft supplied DbgHelp.dll to perform all symbol handling activities.

Correct PDB name and location?

To ensure that the correct PDB is found to match a DLL the following must be true:

- The DLL and PDB file have the same name, except for the extension

  For example test.pdb matches for test.dll or test.exe.

- The first matching PDB file in the PDB search path has the correct checksum

  If DbgHelp finds a PDB file with a different checksum, loading symbols will fail but the search will still stop.

Verify that there are no PDB files with the same file name that are on the PDB search path, except for the PDB file you expect to be used.

You can check the DbgHelp symbol search path to troubleshoot symbol loading failures relating to the symbol search path.

Are compiler and linker producing symbols?

If DbgHelp is still failing to load your symbols, check the following:

- Your program is **compiled** to include symbol information
- Your program is **linked** to include symbol information

  Linker options are different to the compiler options

Running correct version of DLL?

Check that you are using:

- The **most recent** version of your DLL
- The **correct build** version of your DLL

  For example release DLL with release builds, debug DLL with debug builds

Checking for correctly loaded modules

When your application is running, check the modules being loaded by the application.

In Memory Validator, you can check the modules by using the Loaded Modules dialog, or by inspecting the Diagnostics tab.

You need to be sure that your application is not loading a different DLL with the same name from a different directory that is on the search path.

Correct version of DbgHelp.dll?

Try checking the version of DbgHelp.dll used by your Visual Studio installation and the version of DbgHelp.dll distributed with Memory Validator.

If the version used by Visual Studio is higher, it's possible Microsoft changed the PDB file format, making the symbols unreadable by Memory Validator.

To fix this:

- Copy the DbgHelp.dll from Visual Studio to the Memory Validator installation directory
- Remove any DbgHelp.dll from your application directory

When Memory Validator launches an application it copies Memory Validator's DbgHelp.dll to the directory of the executable.

This ensure that the DbgHelp.dll used is more recent than the default `system32\dbghelp.dll` which may not get updated.

You need to find and remove these dlls - e.g. `c:\myapplication\debug\DbgHelp.dll` etc.

If all else fails...

Sometimes symbolic information will not load for unknown reasons.

In this circumstance, after trying the above suggestions, try changing the location in which symbols are sourced.

You could also try flushing and disabling the caching of symbols.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

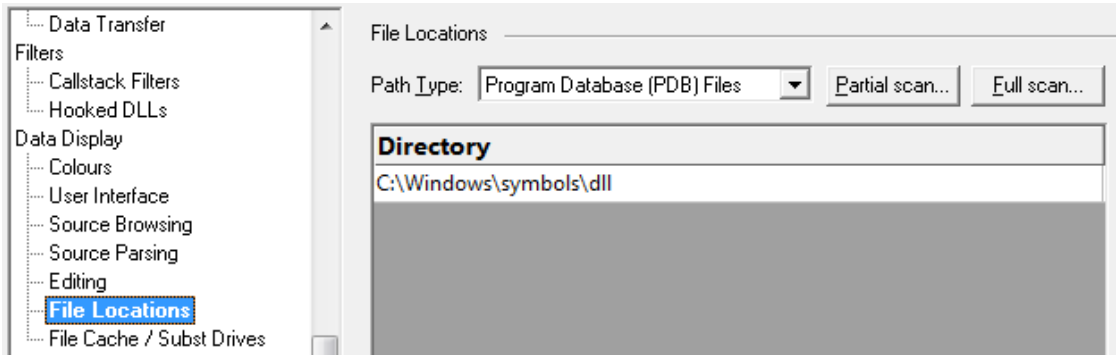
Visual Studio 2005 (8.0) and later versions

You may find that symbols for the `msvcr80.dll`, `msvcr80d.dll`, `mf80.dll`, `mfc80u.dll`, `mfc80d.dll` and `mfc80ud.dll` DLLs are not loaded.

The reason for this is that these symbols are stored in `c:\windows\symbols\dll` rather than with the DLLs themselves.

This is due to the Windows.NET Side-by-Side (WinSxS) DLL/assembly loading.

To resolve this, add the path **c:\windows\symbols\dll** to the list of paths for Program Database (PDB) Files on the File Locations tab:



You may need to restart Memory Validator to get valid symbols for `MFC80(u)(d).dll` if you have already recorded a session for which you did not get symbols.

Alternatively follow the instructions in the question on how to clear the symbol cache:

Metrowerks CodeWarrior for Windows V8 / V9

Metrowerks symbolic information is embedded in the .exe/.dll as CodeView information.

Please consult the documentation for CodeWarrior in order to include debug information (including filenames and line numbers) in the CodeView information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

Salford Software Fortran 95

Salford Fortran 95 symbolic information is embedded in the .exe/.dll as COFF (Common Object File Format) information, with some proprietary extensions to Salford Software (which they have kindly shared with us).

Please consult the documentation for Salford FORTRAN95 to include debug information (including filenames and line numbers) in the COFF information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

MingW compiler

We recommend compiling your software with `-gstabs` to create stabs debugging information.

The `-gCoff` option is also supported, but this does create a lot of unnecessary symbols, making symbol parsing slower.

Troubleshooting DbgHelp.dll #1

Memory Validator uses the Microsoft Debugging DLL, DbgHelp.dll, copying the correct private version to your application's directory as your program is started.

However, there are cases where your application can be started independently, and you must ensure that your application uses the correct DbgHelp.dll.

Diagnostic error messages appear on the Diagnostics tab as in the example below detailing which version of DbgHelp.dll was expected and what was actually loaded.

DbgHelp.dll version	C:\Program Files (x86)\Software Verification\C++ Performance Validator\pvExample\Debug9_0\dbghelp.dll
DbgHelp.dll version	DbgHelp.dll version loaded into target: 6.3.16.1
DbgHelp.dll version	DbgHelp.dll version expected: 6.11.1.404
DbgHelp.dll version warning	DbgHelp.dll loaded has a lower version number than the DbgHelp.dll that ships with C++ Performance Validator.
DbgHelp.dll version warning	This may cause failures when trying read debugging information (Symbols, Filenames, Line Numbers).
DbgHelp.dll version warning	DbgHelp.dll prior to 6.0 will not work properly. DbgHelp.dll 6.9 or better is preferred.
DbgHelp.dll version warning	For best results you need to ensure that C++ Performance Validator's DbgHelp.dll is found on the $PATH before the DbgHelp.dll that is being loaded.
DbgHelp.dll version warning	You can usually do this by putting the current directory ('.') at the start of your $PATH.

If you see any DbgHelp warning dialogs, or get diagnostic errors, ensure the correct DbgHelp.dll is used by:

- **Copy (don't move) DbgHelp.dll**

  from: the Memory Validator install directory

  to: the location of the application being tested (the same directory as the .exe).

  Rerun your test.

- **Try updating the versions of DbgHelp.dll** in:

  `c:\windows\system32`

  and

  `c:\windows\system32\dllcache`

  Accept any Windows permission warnings if you try to do this.

  Rerun your test.

If you *still* continue to have problems, please drop us a line via our support email.

Troubleshooting DbgHelp.dll #2

For versions of Memory Validator older than 3.48, see the next question. (Also, consider upgrading if you can!)

Memory Validator uses the Microsoft Debugging DLL, DbgHelp.dll, copying the correct private version to your application's directory as your program is started.

However, there are cases where your application can be started independently. For example, if you are linked to the Memory Validator API, you can attach to a running or newly started instance of Memory Validator.

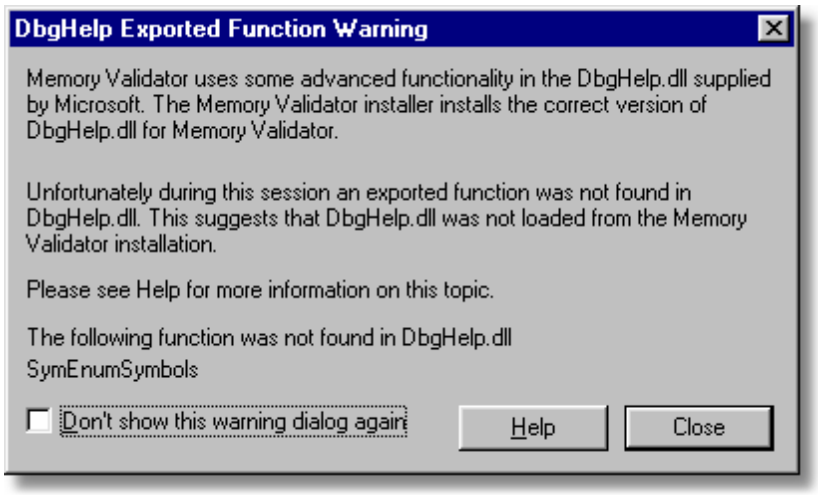
In these situations you must ensure that your application uses the correct DbgHelp.dll.

Installing the correct DbgHelp yourself

To help you detect when the wrong DbgHelp.dll is loaded, Memory Validator has some error messages and a warning dialog.

Diagnostic error messages appear on the Diagnostics tab as in the example shown below indicating which function could not be found in DbgHelp.dll and which feature was trying to use this function.

An optional warning dialog also gives the same information.



- **Help** › Displays this help topic
- **Don't show this warning dialog again** › stops further dialogs. Re-enable it from the Symbols Misc page of the global settings dialog.

If you see this warning dialog, or get diagnostic errors, ensure the correct DbgHelp.dll is used by:

- **Copy (don't move) DbgHelp.dll**

  from: the Memory Validator install directory

  to: the location of the application being tested (the same directory as the .exe).

  Rerun your test.

- **Try updating the versions of DbgHelp.dll** in:

  `c:\windows\system32`

  and

  `c:\windows\system32\dllcache`

  Accept any Windows permission warnings if you try to do this.

  Rerun your test.

If you *still* continue to have problems, please drop us a line via our support email.

How do I examine (and fix) the DbgHelp symbol search path?

It can be confusing to see why symbols fail to load for modules built with compilers that generate PDB files, eg: Microsoft, Intel.

There are typically three reasons for failure: the PDB file is...

- **missing**, for example it was not provided with the executable
- **in the wrong place**, so the the debugging library can't find it
- **the wrong version**, for example from a different build

The diagnostic tab

The Diagnostic tab of Memory Validator displays lots of messages that can help diagnose many problems.

To show only DbgHelp debug information, use the message filter drop down at the top of the diagnostic tab. This lets you examine where DbgHelp.dll looks for symbols.

Examine the output to see if it's finding the PDB file you think it should, and if it rejects the contents of any PDB file it finds.

Output for alternate modules is shown in alternating colorsets, and the messages are the exact same output from the DbgHelp.dll debugging stream.

Examples of examining the diagnostics

Below we show three examples using nativeExample.exe and nativeExample.pdb from our example application.

- **Correct symbol file found**

  DbgHelp first searches in various places looking for nativeExample.pdb

DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dll\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dll\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dll\symbols\exe\nativeExample.pdb - file not found

  Depending on your machine, there may be other search paths included.

  Finally nativeExample.pdb is found in the same directory as the .exe file of the target program

DbgHelp Search Info	DBGHELP: nativeExample - private symbols & lines
DbgHelp Search Info	C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\nativeExample.pdb
Loaded symbols	Loaded PDB symbols for:C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\nativeExample.exe

  DbgHelp loads private symbols and lines, (the alternative being that DbgHelp loads public symbols).

  Outcome:
  Success. Symbols are loaded.

- **Missing symbol file**

  As before, DbgHelp first searches in various places looking for nativeExample.pdb

But, nativeExample.pdb doesn't get found in the same directory as the .exe file of the target program.

DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\nativeExample.pdb - file not found

nativeExample.pdb never gets found on the search path.

SymSrv might then look for additional locations for nativeExample.pdb, but has no luck.

DbgHelp Search Info	SYMSRV: BYINDEX: 0x1
DbgHelp Search Info	C:\MicrosoftSymbols
DbgHelp Search Info	nativeExample.pdb
DbgHelp Search Info	BDCE0A5F65B645B6ABE47733C3D183C51
DbgHelp Search Info	SYMSRV: UNC: C:\MicrosoftSymbols\nativeExample.pdb\BDCE0A5F65B645B6ABE47733C3D183C51\nativeExample.pdb - path not found
DbgHelp Search Info	SYMSRV: UNC: C:\MicrosoftSymbols\nativeExample.pdb\BDCE0A5F65B645B6ABE47733C3D183C51\nativeExample.pd_ - path not found
DbgHelp Search Info	SYMSRV: UNC: C:\MicrosoftSymbols\nativeExample.pdb\BDCE0A5F65B645B6ABE47733C3D183C51\file.ptr - path not found
DbgHelp Search Info	SYMSRV: RESULT: 0x80070003

DbgHelp might find some COFF symbols in the executable, however these don't contain filename or line number information.

DbgHelp Search Info	DBGHELP: mvExample - coff symbols

Finally all options are exhausted.

DbgHelp Search Info	DBGHELP: nativeExample - no symbols loaded

Outcome:
Failure. The PDB file could not be found. Some default symbols are loaded but are not of much use.

Resolution:
Check the File Locations PDB paths to ensure that all the possible paths for PDB files are listed.

- **Incorrect symbol file**

As before, DbgHelp first searches in various places looking for nativeExample.pdb

This time, nativeExample.pdb *does* get found in the same directory as the .exe file of the target program.

DbgHelp tries to load the symbols but fails - the checksum inside the PDB file does not match the module.

This might be because the symbols are for a different build of the software, or it's an incorrectly named PDB file belonging to another program.

DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\nativeExample.pdb - mismatched pdb
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\symbols\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\nativeExample.pdb - mismatched pdb
DbgHelp Search Info	DBGHELP: Couldn't load mismatched pdb for C:\Program Files (x86)\Software Verify\Memory Validator x86\examples\nativeExample\ReleaseNonLink10_0\nativeExample.exe

Finally all options are exhausted.

DbgHelp Search Info	DBGHELP: nativeExample - no symbols loaded

Outcome:
Failure. A PDB file was found, but it was not the right one.

Resolutions:
Double check the PDB is the correct one for the build you are running.
When copying builds from another machine (or from a build server), make sure to copy the correct PDB as well.
Check the File Locations PDB paths to ensure that all the possible paths for PDB files are listed.
Check the order of those PDB paths in case there are multiple paths resulting in the wrong PDB being found first.

15.8 Extensions, services and tools

Including stublib.h in my project doesn't compile. Why?

You may encounter problems when including `stublib.h` in order to link directly with Memory Validator.

Include path problems

Ensure that your project **C preprocessor include paths** reference both of the **stub** and **stublib** subdirectories in the installation directory of Memory Validator.

For example, if Memory Validator is installed in:

```
C:\Program Files (x86)\Software Verify\C++ Memory Validator
```

Then add the following paths for *all* configurations; Debug, Release, etc:

```
C:\Program Files (x86)\Software Verify\C++ Memory Validator\stub
C:\Program Files (x86)\Software Verify\C++ Memory Validator\stublib
```

Compiler errors

If you include `stublib.h`, your project must have included `windows.h` first, (or see below for an alternative).

If you fail to include `windows.h` then `stublib.h` will refer to some none-existent datatypes, causing compiler errors similar to the ones shown below.

Here's an example program that will not compile:

```
#include "stdafx.h"
#include "stublib.h"

int main(int argc, char* argv[])
{
   return 0;
}
```

See the compiler errors from the above code

```
--------------------Configuration: testMV_allEnum - Win32
Debug--------------------
Compiling...
testMV_allEnum.cpp
c:\program files\software verification\memory validator\stub\allenum.h(70) :
error C2146: syntax error : missing ';' before identifier 'lRequest'
c:\program files\software verification\memory validator\stub\allenum.h(70) :
error C2501: 'LONG' : missing storage-class or type specifiers
c:\program files\software verification\memory validator\stub\allenum.h(70) :
error C2501: 'lRequest' : missing storage-class or type specifiers
c:\program files\software verification\memory validator\stub\allenum.h(71) :
error C2146: syntax error : missing ';' before identifier 'reserved3'
c:\program files\software verification\memory validator\stub\allenum.h(71) :
error C2501: 'DWORD' : missing storage-class or type specifiers
c:\program files\software verification\memory validator\stub\allenum.h(71) :
error C2501: 'reserved3' : missing storage-class or type specifiers
c:\program files\software verification\memory validator\stub\allenum.h(73) :
error C2143: syntax error : missing ';' before '*'
c:\program files\software verification\memory validator\stub\allenum.h(73) :
error C2501: 'BYTE' : missing storage-class or type specifiers
c:\program files\software verification\memory validator\stub\allenum.h(74) :
error C2501: 'dde_pbData' : missing storage-class or type specifiers
```

To fix this problem simply include `windows.h` before `stublib.h`

```
#include "stdafx.h"
#include <windows.h>      // new line to fix compile errors
#include "stublib.h"

int main(int argc, char* argv[])
{
   return 0;
}
```

Can't include windows.h?

If including `windows.h` is not an option, you can just define the following types:

```
#define LONG long
#define DWORD unsigned long
#define BYTE unsigned char
#define HANDLE void *
```

What do I do if I cannot use svlMVStubService.lib?

You may find that you can't use **svlMVStubService.lib / svlMVStubService_x64.lib** because your linker doesn't understand the format of the lib file.

If that happens you can use the code below to compile the two functions that would be provided by those libraries.

See the header file

```
#ifndef _SVL_MVSTUB_SERVICE_H
#define _SVL_MVSTUB_SERVICE_H

#include "svlServiceError.h"

// IMPORTANT.
// If you use svlMVStub_LoadMemoryValidator() to load svlMemoryValidatorStub.dll in
// application, you must also use svlMVStub_UnloadMemoryValidator() to unload the D
// your application being closed down. Failure to do so will almost certainly resul
// It does not matter how the application is closed down, you must ensure that you
// svlMVStub_UnloadMemoryValidator() to unload the DLL if you have loaded it.
//
// The DLL prepares itself in different ways and shuts itself down differently depe
// it is:-
// a) Directly linked to the application for use with the API or injected with Memo
//    When the DLL is used in this manner to DLL expects to oversee and manage the
//    shutdown.
// b) Loaded by using svlMVStub_LoadMemoryValidator().
//    When the DLL is used in this manner to DLL expects to be removed prior to app
//    and the behaviour of the DLL is undefined once you enter the program shutdown
//
//     This difference in behaviour is intentional and is done to allow the use of
//     services.

#ifdef __cplusplus
extern "C" {
#endif

SVL_SERVICE_ERROR svlMVStub_LoadMemoryValidator(serviceCallback_FUNC callback,
                                                void                *userParam);

SVL_SERVICE_ERROR svlMVStub_UnloadMemoryValidator();

#ifdef __cplusplus
}
#endif

#endif
```

See the implementation file

```
#include "svlMVStubService.h"

#include <windows.h>
#include <tchar.h>

//-NAME-----------------------------------
//.DESCRIPTION.........................
//.PARAMETERS..........................
//.RETURN.CODES........................
//-----------------------------------------

static HMODULE   hModule = NULL;

//-NAME-----------------------------------
//.DESCRIPTION.........................
//.PARAMETERS..........................
//.RETURN.CODES........................
//-----------------------------------------

typedef void (*ENABLE_STUB_SYMBOL_FUNC)();

SVL_SERVICE_ERROR svlMVStub_LoadMemoryValidator(serviceCallback_FUNC callback,
                                                void                *userParam)
{
   SVL_SERVICE_ERROR   errCode = SVL_OK;

   if (hModule == NULL)
   {
      hModule = LoadLibraryW(L"svlMemoryValidatorStub.dll");     // change this to
      if (hModule != NULL)
      {
         // DLL loaded, set the service callback function

         SETCALLBACK_FUNC   setCallbackFunc;

         setCallbackFunc = (SETCALLBACK_FUNC)GetProcAddress(hModule, "apiSetService
         if (setCallbackFunc != NULL)
         {
            (*setCallbackFunc)(callback, userParam);
         }

         // now start the profiler

         PROC *p;

         p = GetProcAddress(hModule, "startProfiler");
         if (p != NULL)
             (*p)();
```

```
            // now turn on provision of symbols by the stub

            ENABLE_STUB_SYMBOL_FUNC   enableSymbolFunc;

            enableSymbolFunc = (ENABLE_STUB_SYMBOL_FUNC)GetProcAddress(hModule, "apiEn
            if (enableSymbolFunc != NULL)
            {
               (*enableSymbolFunc)();
            }
            else
            {
               errCode = SVL_FAILED_TO_ENABLE_STUB_SYMBOLS;
            }
         }
         else
         {
            errCode = SVL_LOAD_FAILED;
         }
      }
      else
      {
         errCode = SVL_ALREADY_LOADED;
      }

      return errCode;
}

//-NAME-----------------------------------
//.DESCRIPTION.........................
//.PARAMETERS..........................
//.RETURN.CODES........................
//---------------------------------------

typedef void (*UNLOAD_FUNC)();
typedef HANDLE (*GET_STUB_HEAP_FUNC)();

SVL_SERVICE_ERROR svlMVStub_UnloadMemoryValidator()
{
   SVL_SERVICE_ERROR   errCode = SVL_OK;

   if (hModule != NULL)
   {
      // get the stub heap before we shut down the DLL

      HANDLE         hStubHeap = NULL;
      GET_STUB_HEAP_FUNC   getHeapFunc;

      getHeapFunc = (GET_STUB_HEAP_FUNC)GetProcAddress(hModule, "apiGetInternalMVst
      if (getHeapFunc != NULL)
      {
         hStubHeap = (*getHeapFunc)();
      }

      // get the unload stub function
```

```
        UNLOAD_FUNC   unloadFunc;

        unloadFunc = (UNLOAD_FUNC)GetProcAddress(hModule, "apiShutdownMemoryValidator
        if (unloadFunc != NULL)
        {
            (*unloadFunc)();

            // get the function

            HMODULE   hModule;

            hModule = GetModuleHandleW(L"svlMemoryValidatorStub.dll");
            if (hModule != NULL)
            {
                // unload the stub

                FreeLibrary(hModule);

                // destroy the stub's heap (which was still in use whilst FreeLibrary()

                if (hStubHeap != NULL)
                    HeapDestroy(hStubHeap);
                else
                {
                    if (errCode == SVL_OK)
                        errCode = SVL_FAIL_TO_CLEANUP_INTERNAL_HEAP;
                }
            }
            else
            {
                errCode = SVL_FAIL_MODULE_HANDLE;
            }
        }
        else
        {
            errCode = SVL_FAIL_UNLOAD;
        }

        hModule = NULL;
    }
    else
        errCode = SVL_NOT_LOADED;

  return errCode;
}
```

15.9 System and environment

How do I create a Power User on Windows XP?

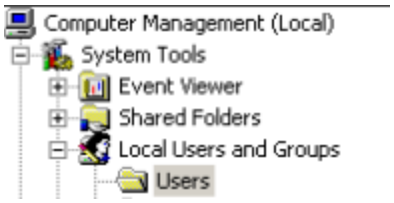
Windows 2000 and Windows XP Pro allow Power User accounts that stop short of full Administrator permissions.

To make an existing user (say **Test User**) a Power User do the following:

- **Start Menu** › **Right click** on **My Computer** › **Manage**

  The Computer Management window appears

- On the left, expand **System Tools** › **Local Users and Groups** › **Users**



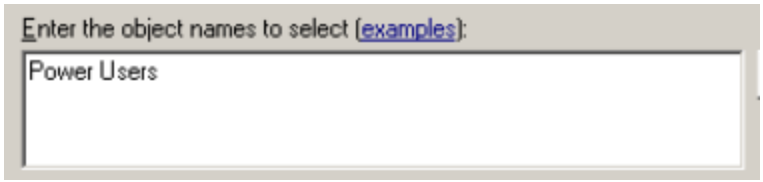
- On the right, select and **Right click** on **'Test User'** › **Properties**

  The User Properties dialog appears

- Select the **Member Of** tab › **Add...**

  The Select Groups dialog appears

- In the bottom box, **type** `Power Users` › **OK**



- In the user properties dialog select **Users** › **Remove** › **OK**
- Close **Computer Management**

Your **Test User** is now a member of the Power Users group - and probably not really a 'Test' User any more!

What file extensions does Memory Validator use?

Most configuration data is stored in the registry, but some information is file-based such as settings, coverage, and filter data.

Memory Validator uses the following extensions:

Session Export and Session Save

- **html** HTML export files
- **xml** XML export files
- **mvm** Session files for 32 or 64 bit Memory Validator
- **mvm_x64**

Settings, Filters, Coverage

- **mvs** Settings for 32 or 64 bit
- **mvs_x64**
- **mvx** Hooked DLLs
- **mvf** Filter files
- **mvc** Coverage files
- **mvd** Coverage files

Ordinal To Symbol Conversion

- **def** Linker definition files
- **ord** Ordinal data files

Program Launch, Extensions

- **dll** Extension DLLs
- **exe** Program files

Source Code

- **cpp** C++
- **c** C
- **h** C and C++
- **cxx** C++
- **hxx** C++
- **hpp** C++

15.10 Does Memory Validator do...

⊟ Does Memory Validator track everything at the same time?

It can, but not everything is tracked by default.

Memory Validator can be configured to track as little or as much as you wish,

For example, if you are only interested in the handles returned from `GetDC()` and passed into `ReleaseDC()` you can configure that.

Or if you want to know about every handle allocated and deallocated, every memory allocation and deallocation, all COM objects, all memory errors etc, you can do that too.

However, bear in mind that the more data items being tracked, the more the performance will be affected.

➡ See also the questions on Performance.

Does Memory Validator hook delay loaded functions?

Yes, delay loaded functions can be hooked and are done so by default.

- Set whether delay loaded DLLs are hooked via the Hooked DLLs page of the global settings dialog
- Set whether delay loaded functions are hooked on the Collect settings page.

Does Memory Validator work with NT Services?

Absolutely. There is a help section on working with NT Services.

Does Memory Validator use Thread Local Storage?

Yes, thread local storage is used in the stub part of the program that is injected into the target program.

Memory Validator needs to use thread local storage so that it can keep track of per-thread related data whilst instrumenting your application.

Windows NT provides each application with at least 64 thread local storage slots.

How many TLS slots are used?

Memory Validator uses 1 of these slots, leaving at least 63 available for your program to use at the same time as Memory Validator.

Does Memory Validator support the boot.ini /3GB switch?

Memory Validator supports applications linked with the /LARGEADDRESSAWARE switch (Microsoft linkers and compatible).

When run on a computer with the /3GB switch added to the boot.ini command line, these applications can access 3GB of application addressable memory rather than 2GB.

Only certain Operating Systems support this boot.ini feature:

- Windows Server 2003 family
- Windows XP Professional Edition
- Windows 2000 Datacenter Server

- Windows 2000 Advanced Server
- Windows NT 4.0 Enterprise Edition

➔ Read more about the /3GB switch and Large Address Aware applications.

Part XVI

16 Installing Floating Licensing

How to install floating licences

Floating licences float globally. Your team members in an office on the other side of the world can share a floating licence with you.

If you have floating licences install the software on all machines in your business unit that wish to use the software.

For an overview of how floating licences work, please read this.

Floating licence server

The floating licence server is managed by Software Verify.

No server to setup, no licences to misconfigure. All the things that are bad about floating licences, we've removed all that.

If you need to acquire a licences or release a licence, see the Floating Licences tab.

Floating licence help

If you have problems with the floating licences please contact support@softwareverify.com

If you need to purchase additional floating licences for a new floating licence please visit https://www.softwareverify.com/purchasing/.

If you need to purchase additional floating licences to add to an existing floating licence please contact sales@softwareverify.com.

Part XVI

17 Copyright notices

17.1 Udis86

This software uses the library svlUdis86.dll and svlUdis86_x64.dll. These libraries are modified binary versions of the open source disassembler udis86.

udis86 was hosted at http://udis86.sourceforge.net/
udis86 is currently hosted at https://github.com/vmt/udis86 although the current distribution (at the time of writing) appears to be missing some files required to compile.

The 1.7.0 version of the udis source code contains this copyright notice: Copyright (c) 2005, 2006, Vivek Mohan
The 1.7.2 version of the udis source code contains this copyright notice: Copyright (c) 2002-2009 Vivek Thampi

These copyright notices appear to conflict and the latter copyright notice completely ignores the claims set forth in the 1.7.0 copyright notice.

In accordance with the license terms in the 1.7.2 software we include this binary license.

```
* 1.7.2 Copyright (c) 2002-2009 Vivek Thampi
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without modification,
* are permitted provided that the following conditions are met:
*
*     * Redistributions of source code must retain the above copyright notice,
*       this list of conditions and the following disclaimer.
*     * Redistributions in binary form must reproduce the above copyright notice,
*       this list of conditions and the following disclaimer in the documentation
*       and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
* ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
* ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Index

- . -

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- J -

- K -

- L -

- M -

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Z -