Heap Inspector for PC

**Contents**

[Welcome 3](#_Toc376421604)

[System requirements 4](#_Toc376421605)

[Installing Heap Inspector 4](#_Toc376421606)

[Package contents 4](#_Toc376421607)

[Server Setup Guide 5](#_Toc376421608)

[1: Initialise and Shutdown 5](#_Toc376421609)

[2A: Sending allocations using low-level interception 5](#_Toc376421610)

[2B: Sending allocations manually 6](#_Toc376421611)

[Using multiple heaps 6](#_Toc376421612)

[Adjusting Heap Ranges 7](#_Toc376421613)

[Samples 8](#_Toc376421614)

[Samples\Hook 8](#_Toc376421615)

[Samples\Manual 8](#_Toc376421616)

[Samples\ReplaceNewDelete 8](#_Toc376421617)

[Samples\MultiThreadedHook 8](#_Toc376421618)

[Using the Microsoft Symbol Server 9](#_Toc376421619)

[Using a Custom Symbol Server 9](#_Toc376421620)

[Exporting to CSV 10](#_Toc376421621)

[General usage tips 11](#_Toc376421622)

[The Windows allocation routines 12](#_Toc376421623)

# Welcome

Welcome to Heap Inspector! Heap Inspector is a C++ memory analysis tool. It operates using a client-server model. Allocations are sent by the server over the network to a client. The client provides powerful statistics about all of the allocations. All of this is done in real-time.

HeapInspector mainly focuses on *memory consumption*, *memory leaks* and *memory fragmentation*.

Heap Inspector has been battle-tested on various commercial PC and PlayStation 3 products and it is used in Sony’s PhyreEngine. Nevertheless, it is very well possible that issues may arise. Please do not hesitate to give feedback through [info@heapinspector.com](mailto:info@heapinspector.com). The product will continue to evolve and your feedback will help in shaping the product.

This documentation does not yet include a lot of information on how to use the client. There is a demonstration video on [www.heapinspector.com](http://www.heapinspector.com) that shows most of the functionality of the client.

# System requirements

* Windows Vista / Windows 7 / Windows 8
* Visual Studio 2010 / 2012 / 2013
* 64 bit hardware/OS.

Heap Inspector for PC supports x86 applications (32-bit) and x64 applications (64 bit). Connection to the localhost (127.0.0.1) is supported only, remote network connections are not yet supported.

# Installing Heap Inspector

Heap Inspector is distributed with a common zip file and one or more platform zip files. The common zip file contains the Heap Inspector client and other files that are shared among platforms. The platform zip file(s) contain the platform specific server and sample code, along with a platform plugin for the client.

To install, simply unzip all files to the same location. Where necessary, choose to overwrite files or folders.

**Warning**: mixing zip files from different versions will cause conflicts. Heap Inspector will notify you if platform plugins are loaded that do not match the client’s version. Likewise, if you are trying to connect to a server that is incompatible (running on a different protocol), an error will be displayed.

# Package contents

When unzipped, the following directory structure should be displayed:

/Client/Release Contains HeapInspector.exe, the executable for the client.

/Client/Release/Plugins The platform plugins for Heap Inspector.

/Documentation Contains this documentation file and other documents.

/Server The HeapInspector server API.

/Samples A few simple samples that show how to setup HeapInspector.

In this package, a debug variant of HeapInspector.exe is provided in /Client/Debug. For performance reasons, it is not recommended that this executable is used. If there are any problems, running the debug variant may give additional information.

# Server Setup Guide

## 1: Initialise and Shutdown

* 1. Intialise /Shutdown: initialise as early as possibly in your main.
  2. Include HeapInspectorServer.h

Example:

#include "../../Server/HeapInspectorServer.h"

int main()

{

using namespace HeapInspectorServer;

Initialise(3000, WaitForConnection\_Enabled);

// ...

Shutdown();

}

1. You need to link with the Heap Inspector library from the correct Visual Studio version and the correct platform. The lib path is formatted as: *configuration\_platform\_visualstudio,* where *configuration* is either ‘Debug’ or ‘Release’, *platform* is either ‘win32’ or ‘x64’ and *visualstudio* is either ‘VS2010’, ‘VS2012’ or ‘VS2013’. All libraries are named HeapInspectorServer.lib. Example: \Debug\_x64\_VS2013\HeapInspectorServer.lib.
2. Also link with wsock32.lib if you aren’t already.

Next you can choose two techniques: intercepting all allocations at a low level, or manually sending allocations. These are described in the following sections.

## 2A: Sending allocations using low-level interception

HeapInspector for PC is capable of intercepting allocations at a very low level. It hooks into Windows’ HeapAlloc, HeapRealloc and HeapFree functions. Any malloc, new or Windows allocation function will eventually go to HeapAlloc (see chapter ‘The Windows allocation routines’). This means that you can even track DirectX allocations.

Using it is simple:

1. Include Server/PC\_Windows/HeapHooks.h
2. Call HeapInspectorServer::InitialiseHeapHooks and ShutdownHeapHooks.

Example:

#include "../../Server/HeapInspectorServer.h"

#include "../../Server/PC\_Windows/HeapHooks.h"

int main(int argc, char\*\* argv)

{

using namespace HeapInspectorServer;

InitialiseHeapHooks();

Initialise(3000, WaitForConnection\_Enabled);

// ...

Shutdown();

ShutdownHeapHooks();

return 0;

}

That’s all there is to it! This technique is demonstrated in the ‘Hook’ sample. For this technique it is highly recommended to use the Microsoft Symbol Server. See chapter ‘Using the Microsoft Symbol Server’ for details.

*Warning: a great deal of effort has gone into making intercepting on the lowest level possible. However, it remains extremely complicated to intercept on such a low level. Send any remaining problems to* [*info@heapinspector.com*](mailto:info@heapinspector.com)*. If hooking doesn’t work, you can try the different techniques like replacing new/delete. See the samples for details.*

## 2B: Sending allocations manually

Instead of intercepting on a very low level, it may be desirable to hook up your own memory manager or to replace new and delete. This section describes how this works.

The allocations/deallocations are wrapped in BeginAlloc/EndAlloc, BeginRealloc/EndRealloc and BeginFree and EndFree calls. The actual allocation or deallocation need to be performed between those API calls for thread safety.

Example:

void\* operator new(size\_t a\_Size)

{

Mutation mutation = BeginAlloc();

void\* mem = malloc(a\_Size);

EndAlloc(mutation, 0, mem, a\_Size, a\_Size);

return mem;

}

void operator delete(void\* a\_Pointer)

{

Mutation mutation = BeginFree();

free(a\_Pointer);

EndFree(mutation, 0, a\_Pointer);

}

The ‘Manual’ and ‘ReplaceNewDelete’ samples demonstrate this technique in detail.

## Using multiple heaps

If you are sending allocations manually, you can also support multiple heaps. Each HeapInspector Alloc, Realloc and Free API call contains a parameter that identifies the heap. This value is an identifier that can hold any value, as long as it identifies the same heap uniquely.

## Associating names with heaps

To let HeapInspector display a more descriptive name with your heap, you can associate a name to the heap identifier. Just simply call SetHeapName with the same identifier as you are passing to Alloc, Realloc and Free, and a zero terminated string for a name.

*Note: if there is no connection established yet, the name will not be send to Heap Inspector. The reason why Heap Inspector does not cache those names on the server side is because it can be tricky in initialization code to perform heap allocations. I am curious to hear if people find this annoying, blocking or anything. Please send feedback to* [*info@heapinspector.com*](mailto:info@heapinspector.com)*.*

# Samples

## Samples\Hook

The Hook sample demonstrates how to intercept allocations using the HeapAlloc hook as described earlier. This will trace all allocations in your application. Note that the new and delete operators will eventually call malloc and free, so these allocations are intercepted as well.

## Samples\Manual

The Manual sample demonstrates how to manually call the Server API. This is convenient for hooking up your own memory manager and for using multiple heaps.

## Samples\ReplaceNewDelete

The ReplaceNewDelete sample demonstrates how to overload new and delete and how to send those allocations to HeapInspector. It is recommended to use the Hook sample instead of ReplaceNewDelete, as ReplaceNewDelete will only trace a subset of the allocations. If using low-level hooking is problematic for some reason, replacing new and delete may still offer powerful tracing of memory allocations.

## Samples\MultiThreadedHook

The MultiThreadedHook sample is mostly a show- and testcase for HeapInspector’s ability to handle multiple threads and allocations during the static initialization phase. Please note that there aren’t any extra requirements for using HeapInspector with multiple threads.

# Using the Microsoft Symbol Server

Many of the items in the callstacks that are sent to Heap Inspector will be in Windows code. This is especially the case when using low-level interception of HeapAlloc. There may be many unknown symbols in your callgraph. Heap Inspector supports the use of the Microsoft Symbol Server. The Microsoft Symbol Server is a server that stores symbol information for many Microsoft products and versions of those products. This enables us to see proper callstack information, even within low-level Windows functionality.

To enable this functionality, go to Tools/Options. Select ‘PC\_Windows’ and check the ‘Use Microsoft Symbol Server’ checkbox. Fill in the directory where you want to store the symbols on your local drive and you’re done.

Use of the Microsoft Symbol Server requires an internet connection. At first use you are required to accept the Microsoft agreement (a popup dialog will come up). Downloading the symbols will take some time, so loading symbols may sometimes take longer.

# Using a Custom Symbol Server

Heap Inspector also supports the use of a custom symbol server. A custom symbol server can be used in a setup where an application is built by a build server. The build server then submits the symbols to the local symbol server and publishes the executable, for instance through source control.

From that point onwards it is possible for Heap Inspector to connect to the published executable and use the symbols from the local symbol server. This means you can profile your application at any time, even if you haven’t build the executable yourself.

# Exporting to CSV

At this moment it is possible to export data to CSV format. You can use this to perform your own analysis on the data. Excel can use this format, but you can also easily write your own tools to perform custom analysis. This option is available in the snapshots view.

When exporting a snapshot, all allocations that were alive at that moment in time are exported. For each block, the address, size requested, size received and timestamp are exported. Also, the callstack ID is exported.

The callstack ID is a unique ID that represents an entire callstack. So all the return addresses are collapsed into a single ID. This is a very memory efficient way to deal with callstacks, also for the CSV format. The indirection avoids a lot of redundancy. But also, this ID can be used in analysis to identify one location where an allocation occurred. The location is not only the function who called it, but also the entire path that lead to that function. This is extremely convenient for a callgraph analysis, for instance.

Naturally you would like to resolve the callstack ID to something that is readable for yourself, so you have the option to export the symbols as well. This can be exported to either separate files, or embedded into the same CSV file. An empty line will be printed inbetween.

The symbol information is stored in two passes again. The first table is a map from the callstack ID to a list of return addresses. These return addresses can be converted to symbol information using the symbol table. The symbol table can convert a return address to function name, line number and source file.

*This feature was requested by multiple people, but it is not a feature I am actively using. So I made a guess what would be the most appropriate information. I can imagine that the information is too verbose and that a simplified format would be more convenient. Please send feedback about the usability of this feature to* [*info@heapinspector.com*](mailto:info@heapinspector.com) *.*

# General usage tips

* The server can be run in release and debug configurations. Release configurations will give best performance, but debug will show more accurate callstacks.
* The allocation graphs and layout view can be panned and zoomed by using the mouse and the CTRL and ALT keys.
* The layout view is disabled by default. The layout view requires quite some CPU power and should only be enabled when necessary to prevent the server from slowing down. To enable it, press the “Auto refresh” button. Alternatively you can press the refresh button when necessary.
* Heap Inspector can store quite a long history of data by using compression. There is a buffer reserved for this data. If the buffer is full, chunks within the buffer will be dropped. The allocation graphs will display a dark-green line for the part that cannot be inspected in detail. To make the buffer larger, go to tools/options.
* You can control the refresh speed in tools/options. The higher the refresh speed, the more CPU power it will consume.

# The Windows allocation routines

In Windows, there are many allocation functions that all serve a different purpose. It is often not very clear how these functions interoperate. The information from various users on this subject on internet is not very trustworthy, so attached is an attempt to shed some light on the subject. It is not entirely complete, but this is the result from observing what happens when breakpoints are put in the low-level functions.

Heap Inspector intercepts allocations at the HeapAlloc and RtlAllocateHeap level.

malloc

operator new newGlobalAlloc

Alloc

HeapAlloc

RtlAllocateHeap

Client

LocalAlloc

GlobalAlloc

VirtualAlloc

= Calls

= May call

To get a good grip on what every allocator does, refer to the MSDN documentation.