Elephant Memory Manager SDK v1.76

Elephant Memory Manager is a sophisticated allocation replacement for your company with advanced features to help you increase productivity where it counts.

3/28/2016

Introduction

The content of this document is subject to change.

Welcome to the development kit of Jury Rig Software Ltd's Elephant Memory Manager. Elephant is probably the most advanced memory allocation system you will ever use to help you track memory use and bugs quicker than ever before on any platform you need.

Hopefully this document will cover everything. Contact support@juryrigsoftware.com if it doesn't answer any questions you may have.

Elephant has been designed by game programmers for game programmers. So if something doesn't meet your needs please contact us. Elephant has been in heavy use by us and partners and we have tried our best to have tested every possible configuration possible but bugs may exist so again if you think you have found a bug contact us.

Changes since last revision

Version 1.76

- Elephant Memory Manager
    - Added the ability to list the callstacks when reporting
        - Windows platform will show the correct function name
- Goldfish
    - Goldfish LiveView allows you to display the total and switch between memory size and allocation counts

Version 1.75

- Elephant Memory Manager
    - Fixed transferring JRS Map file names could get corrupted.

Version 1.74

- Elephant Memory Manager
    - Changed the way the default port for Liveview is set. Allows the ability to connect to multiple apps via different ports.
    - Fixed warnings to do with heap allocation systems.

Version 1.73

- Elephant Memory Manager
    - Added the ability to pass in External Id's when allocating and freeing memory.

Version 1.72

- Goldfish
    - Mapfile reading error fix for some GCC's.

Version 1.71

- Elephant Memory Manager
    - Free block bin allocation error where size could go to wrong bin fixed.
    - ReportAllocationsMemoryOrder has a flag to disable/enable reporting of free blocks to debugger output.
- Goldfish
    - Fixed error where compare window could reference out of range index.
    - Added largest free fragment size to the Overview window.

Version 1.70

- Elephant Memory Manager
    - Fixed potential lock issue with Reclaiming (still experimental).
    - Added MemoryManagerPlatformFunctionNameFromAddress to platform specific

        - Elephant can now transfer the method names eliminating the need for a map file.
- Goldfish
    - Various UI improvements
    - Supports JuryRigSoftware Map files.
    - LiveView will transfer the Function names if the platform supports it.

Version 1.61

- Elephant Memory Manager
    - Fixed potential threading issue with resizing heaps and freeing memory with cMemoryManager::Free

Version 1.60

- Elephant Memory Manager
    - Reclaim (experimental).
    - Create Goldfish captures from the application.
    - Memory Manager Added functions/methods
        - ReportAllToGoldfish
        - ReportContinuousStartToGoldfish
        - ReportContinuousStopToGoldfish
        - Reclaim
    - cHeap Added functions/methods
        - Reclaim
    - Fixed
        - Continuous logging.
        - NI Heap information transfer to Goldfish.
        - Pools not freeing correctly in all circumstances.
        - Memory pointer addresses not always reporting correctly in 64bit.
- Goldfish
    - Ability to step through different types of allocations.
    - Compare performance speed increased.
    - Various import crash bugs.

Version 1.51

- Elephant Memory Manager
    - Bug fix for Windows versions in CLO implementation (would fail on some processors because the result is undefined).

Version 1.50

- Elephant Memory Manager
    - Bug fix in resizable mode for aligned allocations that could fail under some circumstances.
    - Added cHeapNonIntrusive.
    - Examples

        - Custom System memory allocator example
        - Non Intrusive Heap Example
- Goldfish
    - Support for Non Intrusive Heap.
    - Right click on the Overview, MapFiles or Continuous will give you the option to delete it.
    - Sorts size, largest first.
    - Option added (default) to automatically map the only map file loaded to Overviews and Continuous files.

Version 1.41

- Elephant Memory Manager
    - Fixed rare out of memory bug with alignments during a heap resize in resizable mode.

Version 1.40

- Elephant Memory Manager
    - Destroy heap can work in any order for resizable heaps.
    - Fixed crash bug with resizable heaps under certain circumstances.

Version 1.35

- Elephant Memory Manager
    - Prevented CreateHeap creating a heap with a user memory address when it was self managed.
    - Added a check to ensure the dynamic resize heap memory was correctly aligned.

Version 1.34

- Elephant Memory Manger
    - Fixed an issue when resizing heaps where it may release a thread lock twice (platform/threadlock implementation dependant).

Version 1.33

- Elephant Memory Manager
    - Fixed an issue with Post Init Enhanced Debugging. To many Free operations would block and could not progress.
    - Fixed an issue in the reporting where the aligned size would include a free size resulting in incorrect values (introduced in v1.30).

Version 1.32

- Elephant Memory Manager
    - Fixed a Resizable Memory issue that in some situations could result in an invalid internal pointer.
    - Improved internal allocation minimum size performance.

- Documentation
    - Added Resizable Memory limitation issue.
    - Added documentation replacing the default system allocation callbacks.

Version 1.31

- Elephant Memory Manager
    - Deadlock fixed with continuous logging and Goldfish termination.
    - Added the ability to prevent initialization of Elephant until Goldfish has connected.
        - Makes it possible to grab every allocation via Goldfish with Continuous view.
    - Fixed an OS resource leak in resizable mode when destroying a Heap.
- Goldfish
    - Improved memory and performance usage of LiveView Continuous grabbing.
        - Now saves the continuous logging file dynamically and number of previous operations visible has been reduced.
    - Added 'Immediate Continuous' check that will start grabbing a continuous view when a connection occurs.
    - When opening a file to analyse the default directory is the same one used to store LiveView grabs by default.
    - Improved information when loading Continuous Logging files so you know the current state (it can take a while for multi GB files).
- Documentation
    - Added Goldfish Getting Started Guide

Version 1.3

- Elephant Memory Manager
    - Added documentation to help with adding an additional platform.
    - Added dynamic Resizable heap mode
    - Removed Elastic Memory mode
    - New methods
        - cHeap
            - GetAddressEnd
            - GetResizableSize
            - GetNumberOfLinks
        - cMemoryManager
            - GetSystemPageSize
    - New Heap Detail Value
        - uResizableSize.
    - Added Resizable Mode example.
    - Added VS 2012 example projects.
- Goldfish
    - Changes to handle resizable heaps.
    - Various small bug fixes.

Version 1.22

- Elephant Memory Manager
  - Changed log file separator from , (comma) to ; (semicolon). This avoids potential problems with certain C function names and parsing.
  - Fixed incorrect logging issues for 64bit values for the LLVM compiler
  - Fixed one or two error messages that incorrectly handled 64bit values.
- Goldfish
  - Changed File Options to read Load and not New.
  - Fixed a problem with slight change in MS MAP file formats.
  - Fixed a problem with name demangling not always demangling MS symbols.
  - Fixed a crash that could occur when switching between Continuous and Overview views and selecting a previously selected value.

Version 1.21

- Elephant Memory Manager
  - Crash in Sentinel builds fixed.
    - A crash was located in cHeap::CheckForErrors. This is a result of the real size change in 1.20 and would occur when a free block exists between 2 allocations but was too small to exist.

Version 1.2

- Elephant Memory Manager
  - Support for more compilers and platforms.
  - Allocation size is now byte rather than rounded to nearest value.
  - Memory Manager Added functions/methods
    - IsAllocatedFromAttachedPool
    - GetPoolFromAllocatedMemory
    - Realloc
    - SetMallocDefaultHeap
    - SizeofAllocationAligned
  - Heap added function/methods
    - GetSizeOfLargestFragment
    - IsOutOfMemoryReturnEnabled
    - GetTotalAllocations
    - GetMaxAllocations
    - GetTotalFreeMemory
    - GetPool
  - Pool added function/methods
    - GetSize
    - GetAllocationSize
    - HasNameAndCallstackTracing
    - HasSentinels
    - GetUniqueId

    - GetAddress
  - Changed functions
    - InitializeLiveView
    - InitializeEnhancedDebugging
    - InitializeContinuousDump
    - Initialize
    - InitializeSmallHeap

- Goldfish
  - Large interface changes.
  - Added Hierarchical view
  - Added Liveview continuous data view

Known Issues

- Goldfish
    - Fails with Out Of Memory exception in some circumstances (ironic we know)

Copyright Notice

Getting Started

The following section shows you how to get Elephant up and running quickly. Elephant is easy to set up and get working with your project quickly even with some of the more advanced features. Basic set up will take only minutes. The advanced section will help you get the most out of Elephant and set up advanced memory schemes.

Elephant Terms

Elephant has two main areas, the Memory Manager (cMemoryManager) and Heaps(cHeap, cHeapNonIntrusive). These definitions may or may not be different to other uses so we will clarify them here. Pools are a sub set of Heaps.

The Memory Manager controls the overall system while all the main memory operations are done in a Heap. The Memory Manager keeps track of all the Heaps you create.

A Heap is a block of memory that performs the main allocation operations and is of a fixed size. Elephant allows one or many heaps at any one time and even specialist heaps to exist for areas of memory like VRAM.

A Pool is a sub set of a Heap. It only allows fixed size allocations to occur but is very fast and suited for frequent, small allocation where speed is critical.

Setting Up The Build Environment

Memory Manager comes with two header files and different types of library. The only header you need to concern yourself with is JRSMemory.h. If you are using Pools you also need JRSMemory_Pools.h. You also need to link your executable it to the right library file. There are both Debug and Release library files and variations to help with debugging provided with an additional Master library file for when you finish your game offering best performance. You can link any of these libraries at any time in your project so you can use the Release library with your debug build if you wish.

JRSMemory_Debug.lib is your default library to include for debugging. It performs basic validation checks and ensures basic memory pitfalls are caught.

JRSMemory_Debug_NAC.lib (Name And Callstack) is the same as the standard debug library but each allocation and free allows you to add a name to the allocation. This might be the function or the file name or it may be just a custom name of your choice. Every allocation and free also tracks 8 levels of callstack which can be properly viewed in Goldfish.

JRSMemory_Debug_S.lib (Sentinel) has extra checks during allocation and free to ensure than the memory was not corrupted in anyway. This is the most basic sentinel check Memory Manager provides without affecting performance too much. You can specify further flags on each heap to test more at the cost of performance.

JRSMemory_Debug_NACS.lib (Name And Callstack with Sentinel) does all the above. It adds much more overhead than the other versions but will allow everything to be tracked.

Replace Debug with Release and you have the compiler optimised versions. NAC, S and NACS versions are also included.

JRSMemory_Master.lib has had everything stripped out that can be to give maximum performance. This means debug checking and some of the standard error checks have been completely removed and will give you no information in a crisis. Use this library for when speed is crucial and when mastering.

Elephant Memory Manager is platform agnostic and can provide the source code. Speak to our tech support for information on compiling it on the platform you need if we don't support it already.

Other Settings

Ensuring your project creates a MAP file is important if you wish to do full callstack debugging with GoldFish. This enables the tool to match every allocations stack with the correct one in your program.

Note: MSVC can remove the information needed to generate a callstack with optimizations so you may need to disable Frame Pointer Optimisation on the x86 platforms. Callstacks cannot be retrieved without this disabled in optimised builds. In MSVC this is the \Oy- flag.

Initializing Memory Manager

Trial

During Elephant trial period you must initialize Elephant with the supplied licence key. To do so you must call before you do anything else:

```
        cMemoryManager::InitializeTrial();
```

Getting Started

The main class of Memory Manager is cMemoryManager. This manages the global pool of memory needed for most of the Heaps you will be working with. cMemoryManager is a singleton class preventing more than one co-existing and preventing problems.

First main function you need to call is Initialize(). Initialize accepts three arguments to enable a few different situations but for our example we will tell Memory Manager to use 512MB or as near to that as possible. Here we use the default arguments for the remaining two. This is a fixed size. Elephant will never go above this limit.

```
cMemoryManager::Get().Initialize(512 << 20);
```

Something to note here is that all 512MB may not always be available due to internal issues like alignment or for some platforms not even be physically possible. Memory Manager will internally keep reducing this amount to find the maximum possible. See the SDK documentation for cMemoryManager::Initialize if you do not desire this feature.

You can put Elephant into resizable mode. This consumes memory as you use it. It may be slightly slower as it needs to go to the system level every time but is the best mode when you do not know your memory limitation. See Resizable Mode for more information and cMemoryManager::Initialize Function.

The call in this situation also creates you a default heap as large as possible.

Now, if you want to allocate memory straight away move right on to the Allocations section as you are good to go. Since you have chosen Elephant you will want to know how to set it up for at least basic error reporting.

Adding Support For Errors and Logging

There are three types of errors and logging you may wish to catch. The first is a basic software breakpoint for when something bad happens like freeing an allocation twice. The second is some sort of warning telling you that that is what has happened. It is recommended to always have these enabled except for final master. The third is writing data to a file which is useful for when you need Goldfish to view your memory in a friendlier manner. This one isn't needed if you use Live View as it enables the same information to be transferred externally. It does however allow you to specify an exact location for capturing memory. One small thing to pay attention to is that logging of text to a TTY window when you have many, many allocations can be really slow. This is out of our hands unfortunately so you may want to go and make some coffee with full logging enabled.

```
void MemoryManagerLogText(const i8 *pText)
{
	printf(pText);
	printf("\n");			// Dont forget the return
}

void MemoryManagerErrorHandle(const i8 *pError, u32 uErrorID)
{
	// This will perform a debug break on x86 under MSVC.
	__asm int 3;
}

void MemoryManagerWriteToFile(void *pData, int size, const char
*pFilePathAndName, bool bAppend)
{
	// Standard C file output.  Note that it always closes the file and
rarely creates a new one.
	FILE *fp;

	fp = fopen(pFilePathAndName, (bAppend) ? "ab" : "wb");
	if(fp)
	{
		fwrite(pData, size, 1, fp);
		fclose(fp);
	}
}
```

Now we have these functions let's look at our initialization code again. Elephant requires knowledge of these functions before it is initialized. Let's do that here.

```
cMemoryManager::InitializeCallbacks(MemoryManagerLogText,
MemoryManagerErrorHandle, MemoryManagerWriteToFile);
cMemoryManager::Get().Initialize(512 << 20);
```

Voila, those 2 lines and the above functions will give you the logging and error catching. The next section covers Heaps, you can skip that section if you wish but knowledge will help you set up the more advanced memory schemes you may want to use.

Live View

As mentioned the best way to capture information is with Live View. See the advanced section for more but if you want to get going now (and why wouldn't you) add this before the Initialize call.

```
cMemoryManager::InitializeLiveView();
```

Heaps

Heaps are a fundamental to Memory Manager. A Heap is an area of memory that Memory Manager uses to store its allocations but keep it separate from other memory areas. Each heap is thread safe which will offer large potential performance benefits in multithreaded applications. Unlike the default system allocation calls which Elephant replaces, each thread can safely allocate and free memory to multiple heaps without costly contention issues.

Memory Manager has added functionality that enables you to customise each Heap to your specific needs. You may want a Heap to only allocate memory in 4k sizes and 128byte alignments for example or you may want to prevent a Heap from error checking or just track down one particular allocation.

The biggest feature however is that Heaps do not have to come from the main memory pool. Some platforms have a system memory and separate memory for VRAM for example. This is often a problem because you have a memory allocator for system ram but either no allocator or a very poor one for VRAM costing you development time.

Types of Heap

Elephant Memory Manager supports two types of Heap. Intrusive and Non Intrusive.

Intrusive heaps store any book keeping information next to the allocation. This allows easy bug tracking and will work on any memory that the CPU has direct access to. Intrusive is supported with cHeap and is the recommended Heap to use for most systems.

Non Intrusive heaps store the book keeping information in a cHeap and no details in the target memory. This method of allocation is ideal for areas of memory that the CPU cannot directly access or are very slow at accessing. Elephant Memory Manager supports this with cHeapNonIntrusive.

Creating a Heap

Creating a Heap is simple if you just require a default type. The most common defaults are already set up for you for this task. We do however need to initialize Memory Manager differently. The first example showed how to create one large area of memory and by default created a Heap be of the same size. Depending on the memory scheme you wish to employ this may cause problems.

This time we initialize Memory Manager as before but this time telling it to create no default heap.

```
cMemoryManager::Get().Initialize(512 << 20, 0);       // 2nd Parameter
specifys the default heap size.

cHeap::sHeapDetails details;

cHeap *pNewHeap = cMemoryManager::Get().CreateHeap(1 << 20, "MyMainHeap",
&details);
```

These two lines create a new Heap called 'MyMainHeap' that is 1MB in size. The Heap will not allow allocations of 0bytes (which is perfectly valid with malloc) unless you specify otherwise and several other restrictions. Once you have familiarized yourself with Heaps and allocations you should read the advanced section on Heap customization and allocation overhead.

This Heap also becomes Memory Managers new default heap. This will mean Elephants cMemoryManager::Malloc function will allocate directly to this heap. If you create another Heap using the method just shown that will become the new default Heap.

What about allocating for VRAM?

As mentioned this is one of the major advantages offered by Elephants Heap system. By default Heaps are managed by the memory manager. It is possible to create heaps pointing to any memory that you have direct access to. On a console VRAM or some form of slow memory for other storage is often available but no standard allocation scheme exists manage that area. To use a Heap for these sorts of memory you need to tell Memory Manager you want to manage it yourself.

This requires the alternative cMemoryManager::CreateHeap method which allows you to pass in a memory pointer. It's slightly more complex to use but not by much. The structure used to pass the details into cMemoryManager::CreateHeap and creating this heap is also your route to customizing the Heap to allocate in a way that you desire.

```
cHeap::sHeapDetails details;

details.bHeapIsSelfManaged = true;

cHeap *pVRAMHeap = cMemoryManager::Get().CreateHeap(<MyPointerToVram>,
<MyVramSizeTo16Bytes>, "VRAMHeap", &details);
```

That is all there is to it. You will have both a system memory heap and a VRAM heap that you can use. Remember that a self managed heap may only have allocations made to it directly via cHeap::AllocateMemory. cMemoryManager::Malloc will never use these heaps.

Note: For areas of memory where a CPU has no direct access or performance penalty please use CreateNonIntrusiveHeap

It should be noted that although Heaps themselves are thread safe, creating, destroying and resizing a heap are not. See the FAQ for why this is.

Creating a Non Intrusive Heap

Creating a Non Intrusive Heap requires a standard Heap to store the entire book keeping overhead. Assuming you have created a heap (see Creating a Heap) already then you can create a Non Intrusive Heap like this:

```
cHeapNonIntrusive::sHeapDetails details;

cHeapNonIntrusive *pNewNIHeap =
cMemoryManager::Get().CreateNonIntrusiveHeap(<MySize>, p_cHeapPtr, "NI
Heap", &details);
```

And you may now allocate directly to this heap.

What about allocating for VRAM?

As mentioned Non Intrusive Heaps have important benefits for VRAM and other areas of memory not accessible directly with the CPU or areas that will cause severe performance problems.

In these situations, you can create a non-intrusive heap directly with a memory address (or value representing the start of memory) and Elephant will organise what goes where by returning valid pointers that can then be accessed via other methods.

This is done like so:

```
cHeapNonIntrusive::sHeapDetails details;

cHeapNonIntrusive *pNewNIHeap =
cMemoryManager::Get().CreateNonIntrusiveHeap(<MyVRAMAddress>, <MySize>,
p_cHeapPtr, "NI Heap", &details);
```

Destroying Heaps

Heap destruction is an easy task but there is one rule for Heaps that are managed by Memory Manager. Self managed heaps do not have these restrictions.

A Heap may only be destroyed in the reverse order that it was created.

Put simply it means the last Heap you created must be the first one you destroy. Elephant will warn if you perform it out of order.

To destroy a Heap do the following:

```
cMemoryManager::Get().DestroyHeap(<Pointer to a cHeap>);
```

The Heap will now be removed. By default any memory left allocated in the Heap will report an error helping you track any leaks. When you destroy a heap all the memory allocated with it will also be destroyed. See the advanced section for more information on how to allow destruction of a Heap with memory remaining without causing a warning to be triggered.

Additional

Locking (Preventing Operations)

All heaps can be locked and prevented from having any allocation or free operations taking place. This enables precise control over when memory should be allocated or not and finer control on performance. See cHeap::EnableLock Function and cHeapNonIntrusive::EnableLock Function.

Threading

All heaps are thread safe by default. However, you may disable this on Heap creation. This will improve performance but you should ensure that multiple operations do not overlapped and cause problems.

Reporting

Sometimes the best reporting does not come from Goldfish. It comes from Elephant. At any time, you can use the reporting functions to give you statistics and full dumps.

Allocations

The main reason you are using Elephant is to allocate memory and there are two ways to achieve this. The main way is to allocate memory through a Heap which allows you to know exactly which area you are requesting memory from. The other is with the generic cMemoryManager::Malloc method. This will allocate memory from the last create heap that is fully managed. It's more of a convenience function really allowing you to focus on development.

Freeing memory is just as simple. Again two methods exist for this, the direct way via cHeap or the generic way with cMemoryManager::Free.

Below initializes Memory Manager with one default Heap and also a VRAM Heap and demonstrates allocating and freeing memory using the two different methods. The first will allocate 533bytes and use the default alignment of that heap. The second allocates directly from a Heap, VRAM in this example, and aligns it to 4k.

```
cMemoryManager::Get().Initialize(512 << 20);

cHeap::sHeapDetails details;

details.bHeapIsSelfManaged = true;

cHeap *pVRAMHeap = cMemoryManager::Get().CreateHeap(<MyPointerToVram>,
<MyVramSizeTo16Bytes>, "VRAMHeap", &details);

void *pData = cMemoryManager::Get().Malloc(1024);

void *pVRAMPtr = pVRAMHeap->AllocateMemory(65535, 4096);

// Free the memory now

cMemoryManager::Get().Free(pData);

pVRAMHeap->FreeMemory(pVRAMPtr);
```

Note that for freeing an allocation you can use cMemoryManager::Free for self managed Heaps as well. Elephant checks the memory address to ensure the allocation you are freeing was allocated from the correct Heap.

cHeap::AllocateMemory and cMemoryManager::Malloc also allow you to add additional flags and debugging aids like a specific name to associate with the allocation. See the SDK documentation for more details.

Reporting

Memory Manager offers the ability to provide information for either the whole system or on a Heap by Heap basis.

During runtime you can query each heap and get information based on the number of allocations made, how much memory is consumed etc in a full detail. Some of these functions can be time consuming so should be queried infrequently. This can all be logged to the TTY or if you desire it written to a file for use later in Goldfish to view and aid any potential debugging.

All the reporting methods take the form of Report<X>. Some allow you to supply a file name and log the desired data for use in Goldfish or allows processing by yourself.

The most basic function is cMemoryManager::ReportStatistics. This will give you detailed information but not about each individual allocation contained within Elephant.

At runtime, you also may wish to have a constant update of information consumed by Memory Manager. You can get access to the number of allocations made and how much memory is consumed very quickly for use on a frame by frame basis.

Reclaiming

Note: Reclaiming is in an experimental state.

When using the resizable option Heaps can be extremely large, potentially larger than the system ram. By default, the OS then starts using page memory and performance can suffer. Even after you have freed memory, heap sizes still consume the memory given by the OS. Performance often returns because the paging occurs less but it may affect other systems performance.

Reclaiming is a method of forcing Elephant to give this back to the OS. You can call Reclaim on cMemoryManager or on each heap individually.

Reclaiming can only return the blocks of memory that have been alloced by the heap during a resize. By default, this is in 32MB blocks. Most OS's do not allow these blocks to be split so that is the limitation that has been adopted by Elephant as well. This means that badly fragmented heaps may not return any memory at all to the OS if for example it could not find a while 32MB block that was free.

Pools

Pools are an addition to Elephant Heaps and have several advantages mainly in regards to speed and allocation overhead reduction. Pools also feature many of the same features to help track any potential problems quickly. Pools also have some disadvantages as well though, all documented in the advanced section.

Memory operations are much quicker than from a Heap. All the allocations come from a fixed area of memory, just like with a Heap, but they must all be of the same size. The advantage being that a Pool, depending on type, can make these allocations with zero overhead making it ideal for small allocations.

Each Pool is linked to a Heap to allow Elephant to have a few advantages over tracking each allocation and ensures memory is freed from the correct Pool. It also allows situations regarding situations where the Pool may run out of space for allocations. With customization, you can direct any allocations that do not fit to be allocated from the Heap instead.

There are currently two types of Pool in Elephant documented in the advanced section.

Creating a Pool

Creating a pool is a bit like creating a Heap. You need to pass a few required arguments and either rely on defaults or customise each Pool to your exact needs.

You must always tell a heap the maximum number of allocations it can hold and the size of each element.

The example below creates a pool that holds 32K of C 'int' types.

```
sPoolDetails DetPoolDetails;
cPool *pPoolDet = cMemoryManager::Get().CreatePool(sizeof(int), 32678,
       "SamplePool", &DetPoolDetails, NULL);
```

The sample creates the pool with default parameters and will now allow you to allocate and free memory from it.

Pool Allocation

Memory operations within a Pool follow similar patterns to the Heap operations. Memory can be allocated and freed and stored with a unique name if the user requires. Some Pools also allow detailed memory tracking about every operation also but it depends on the type. A cPool is the default type and allows this.

The following example shows how to allocate and free from the previously created Pool.

```
void *pData = pPool->AllocateMemory();
pPool->FreeMemory(pData);
```

Destroying a Pool

Removing a Pool from Elephant is quick and simple. Elephant will check for errors by default just like a Heap to aid in debugging if any exist.

```
cMemoryManager::Get().DestroyPool(pPool);
```

Goldfish

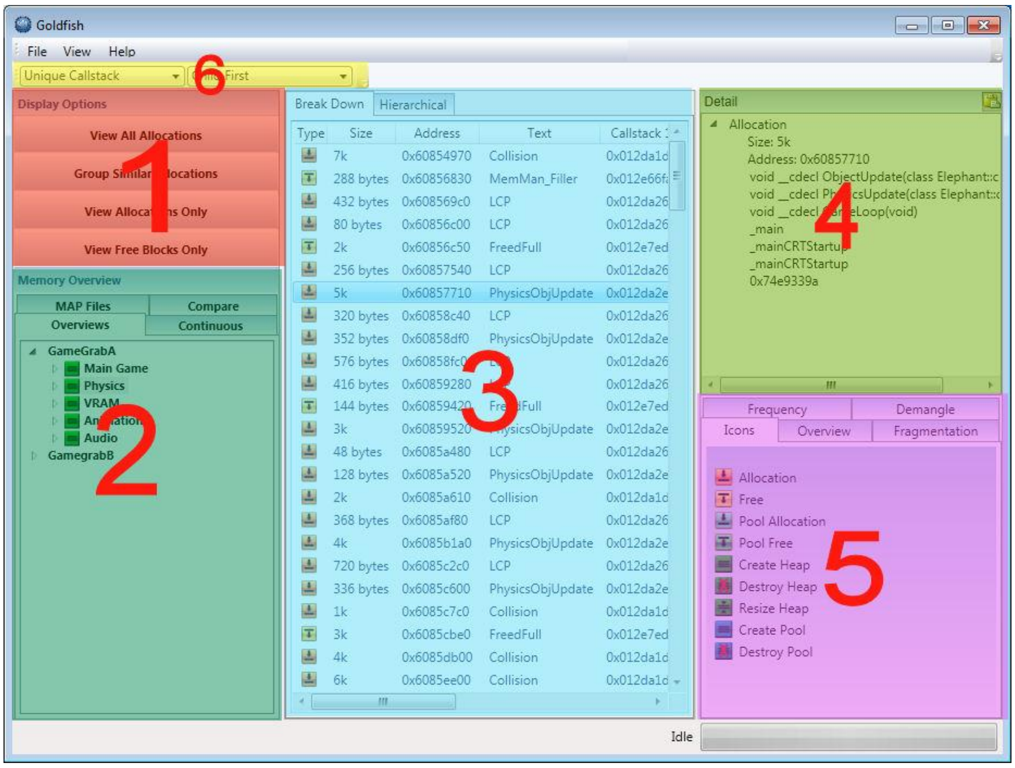
What is Goldfish?

Goldfish is a tool to help you examine the details of Elephant Memory Manager in a clear and concise manner in order to track down memory leaks and the cause of fragmentation. With Elephant Memory Manager's Continuous Logging feature it can also show you memory consumption over time and compare memory dumps against other memory dumps to isolate any differences.

Goldfish also allows you to examine your programs code and global variable memory consumption by using MAP file. It will tell you where your memory is going before you ever allocate any memory.

Main Window

On opening Goldfish you will be presented with the main interface which is split into separate parts.



Type	Size	Address	Text	Callstack
⤓	7k	0x60854970	Collision	0x012da1d
⊤	288 bytes	0x60856830	MemMan_Filler	0x012e66f
⤓	432 bytes	0x608569c0	LCP	0x012da26
⤓	80 bytes	0x60856c00	LCP	0x012da26
⊤	2k	0x60856c50	FreedFull	0x012e7ed
⤓	256 bytes	0x60857540	LCP	0x012da26
⤓	5k	0x60857710	PhysicsObjUpdate	0x012da2e
⤓	320 bytes	0x60858c40	LCP	0x012da26
⤓	352 bytes	0x60858df0	PhysicsObjUpdate	0x012da2e
⤓	576 bytes	0x60858fc0	LCP	0x012da26
⤓	416 bytes	0x60859280	LCP	0x012da26
⊤	144 bytes	0x60859420	FreedFull	0x012e7ed
⤓	3k	0x60859520	PhysicsObjUpdate	0x012da2e
⤓	48 bytes	0x6085a480	LCP	0x012da26
⤓	128 bytes	0x6085a520	PhysicsObjUpdate	0x012da2e
⤓	2k	0x6085a610	Collision	0x012da1d
⤓	368 bytes	0x6085af80	LCP	0x012da26
⤓	4k	0x6085b1a0	PhysicsObjUpdate	0x012da2e
⤓	720 bytes	0x6085c2c0	LCP	0x012da26
⤓	336 bytes	0x6085c600	PhysicsObjUpdate	0x012da2e
⤓	1k	0x6085c7c0	Collision	0x012da1d
⊤	3k	0x6085cbe0	FreedFull	0x012e7ed
⤓	4k	0x6085db00	Collision	0x012da1d
⤓	6k	0x6085ee00	Collision	0x012da1d

```
Allocation
    Size: 5k
    Address: 0x60857710
    void __cdecl ObjectUpdate(class Elephant::c
    void __cdecl PhysicsUpdate(class Elephant::c
    void __cdecl GameLoop(void)
    _main
    _mainCRTStartup
    _mainCRTStartup
    0x74e9339a
```

1. The Display panel allow for selection and filtering of memory items in the main view (3). When some options are not available with certain memory views they will be greyed out.
2. The Memory Overview shows you the four main areas of memory that can be monitored.
3. This is the Memory Display panel. It shows you the details depending on what you have selected in the Memory Overview panel.
4. The Detail panel shows you more detail about any the selected item in the Memory Display panel.
5. The Overview panel will give extra details on the whole memory area.
6. Options for various rules.

Various Memory Overviews

Goldfish allows you to view details of three different aspects of memory usage and a 4th to help you locate bugs. Two of these are directly related to the debugging output of Elephant and the 3rd is to help with isolating memory used by your code directly.

Overviews

Overviews are the main type of memory views you are likely to be using. These sort of views are created by Elephant using the cMemoryManager/cHeap::ReportAll or the cMemoryManager/cHeap::ReportAllocationsMemoryOrder functions.

A better method of creating these is to use Live View.

Continuous

Continuous memory views show what you application does over a certain time period. For example, you can monitor the memory allocation and deallocation over the period of several frames. Using this view you can also monitor memory high and lows and detect if any allocations are occurring when they shouldnt.

Map Files

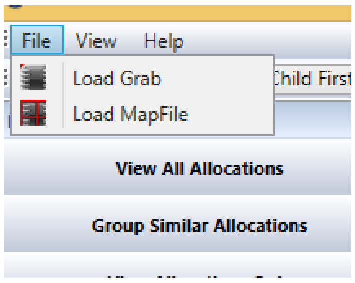
This view allows you to view your program structure in detail by loading in a MAP file. The other advantage is that by combining a MAP file and Elephant overview or continuous views (using the NAC or NACS library files), you can link each memory block to the actual function name they originated from.

Compared Overviews

Compared overviews are views which allow you to compare two Overviews and find out what memory changed between the two. This sort of view is very helpful in helping you track down memory leaks between Overview dumps.

Loading the data

Loading one of the three types of data is easy. Go to the File menu and select the type you wish to load. If you load the wrong type Goldfish will warn you and proceed no further.



Loading can sometimes take a while and there is a progress bar in the bottom right of Goldfish that will keep you updated on its progress.

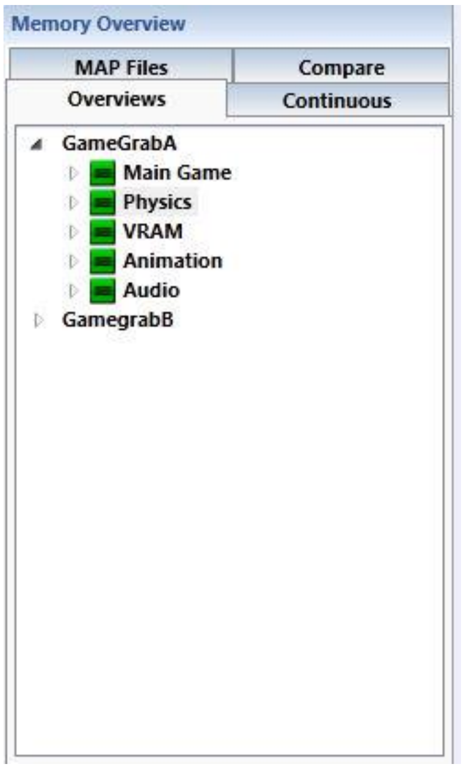
Once loaded the information will be added to the Memory Overview or Continuous panel and may prompt you with a window asking if you wish to link any loaded MAP files to a Continuous or Overview file loaded in. If the platform supports the transfer of the function names from LiveView it will try and load the matching MAP file where applicable. See the 'Linking MAP files' section for more information.

Goldfish will break each item you load into its own area so if you want to compare how your memory usage has changed between the last 10 iterations it will allow this.

Memory Overview

The Memory Overview panel displays and breaks down the overview, continuous, compared and MAP's into smaller chunks to help visualization of the memory used by your application. Every time one is loaded it is put into this panel.

From here you can select from the break down to view the information you want to see which will often be viewed in the Memory Display.



Overviews

Overviews will be broken down into the individual heaps giving you an overview of where the heap exists, its size, the number of allocations and how many fragments (free blocks) and the total amount of fragmented memory. In the image you can see the Generic heap doesn't have many free blocks but these are quite large.

In this example there are two Overview files loaded and their name is displayed in brackets next to **Memory**. Selecting memory will display the entirety of Elephants memory dump to be searched through. Selecting the individual heap (here called **Generic**) will just display the heap breakdown.

Continuous

The Continuous section shows any continuous files loaded. Instead of heaps the break down comes from user markers you may insert. Here a marker has been added to detect any allocations that happen per frame. Selecting **Continuous Log (x)** will show the entirety and selecting one of the markers will show just that part tracked.

Map Files

Map files get broken into two parts, one for functions and one for data. The count says how many items were picked up and for data the total size global variables etc take up. Selecting **Data** or **Functions** will show the data or code in your executable.

Compared Overviews

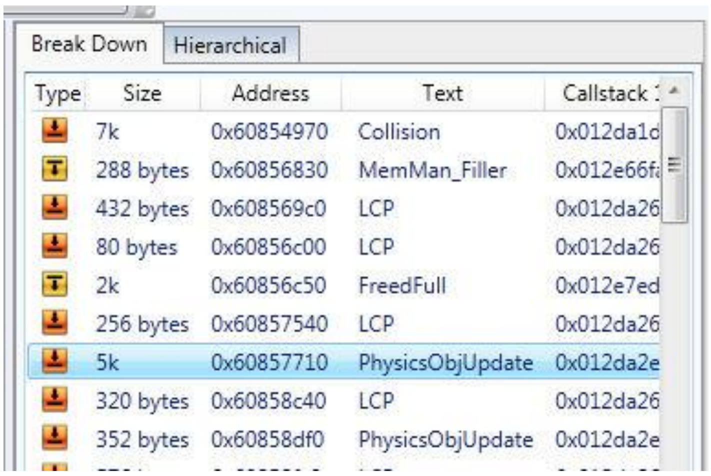
Compared Overviews are generated by comparing two Overviews. Compared Overviews are broken down almost the same as standard Overviews with the difference that they will also show you any allocation differences between the two. Unique in A shows how many allocations are different in Overview A verses the allocations in Overview B.

Memory Display

The main window you will be working with is the Memory Display window. For all views except MAP files the information displayed will mostly be the same except for some small differences.

Overview, Continuous and Compared Views

The main window will look a little like the image below. It is made up of columns showing what operation occurred whether it was an allocation, free, new heap etc. Other columns show the various details related to that operation like its memory address, size etc. Some columns will only be visible depending on the type or the option selected in the Display panel.



Type	Size	Address	Text	Callstack
🟧	7k	0x60854970	Collision	0x012da1d
🟨	288 bytes	0x60856830	MemMan_Filler	0x012e66f
🟧	432 bytes	0x608569c0	LCP	0x012da26
🟧	80 bytes	0x60856c00	LCP	0x012da26
🟨	2k	0x60856c50	FreedFull	0x012e7ed
🟧	256 bytes	0x60857540	LCP	0x012da26
🟧	5k	0x60857710	PhysicsObjUpdate	0x012da2e
🟧	320 bytes	0x60858c40	LCP	0x012da26
🟧	352 bytes	0x60858df0	PhysicsObjUpdate	0x012da2e

The column on the left will always show the operation. The icons represent different values depending on what operation. They are:

Icon	Operation
🟧	Allocation
🟨	Free
🟨	Reallocation
🟩	New Heap
🟩	Removed Heap
🟩	Resized Heap
🟩	Pool allocation
🟩	Pool free
🟦	New Pool
🟦	Removed Pool

Other columns show various other details they are:

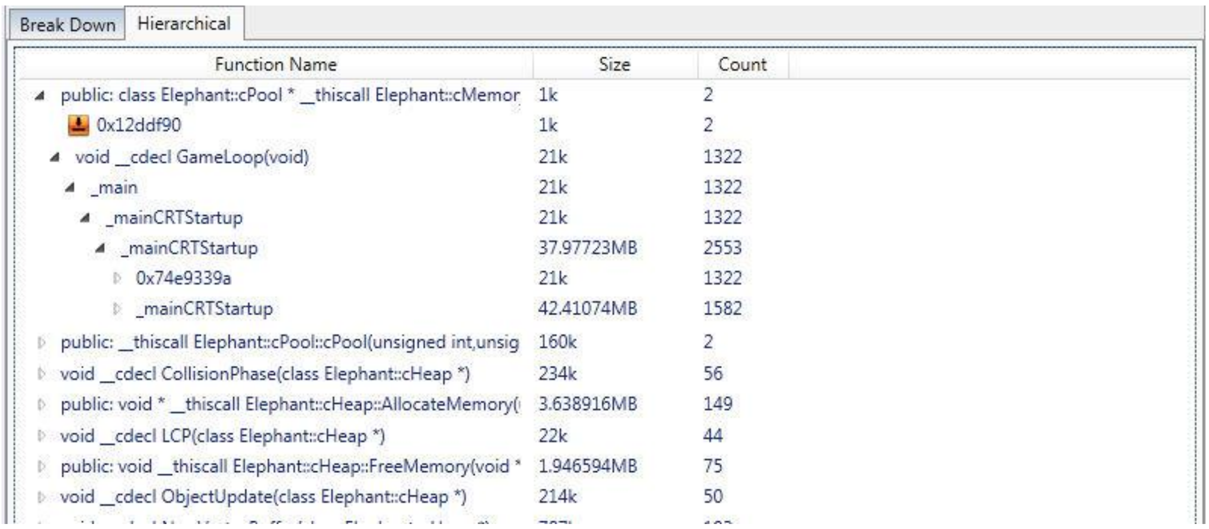
Name	**Description**
Address	This is the address in memory of the allocation or free block.
Size	This is the size in bytes of the allocation or free block of memory.
Unique ID	Each allocation or free block has a unique ID. You can track or find memory with this. See advanced.

Text	Text that you specified on the memory allocate or free function argument. Must be using NAC or NACS library.
Callstack <N>	Callstack level stored with the allocation or free. Must be using NAC or NACS library.
Number	The number in order the operation occurred.
Flags	Flag associated with the operation.
Unique To Table	Ticked if the operation occurred in the second table compared. Unticked otherwise. Only in compared views.
Alignment	The alignment the user specified for the allocation.
Allocations	Total number of allocations in the grouped view.
Grouped By	What the allocations are grouped by.

Hierarchical Views

Overviews and Continuous views offer the ability to see where how functions and the callstacks in your program work together.

You can view from the parent or the child down by using the options panel in the top left.

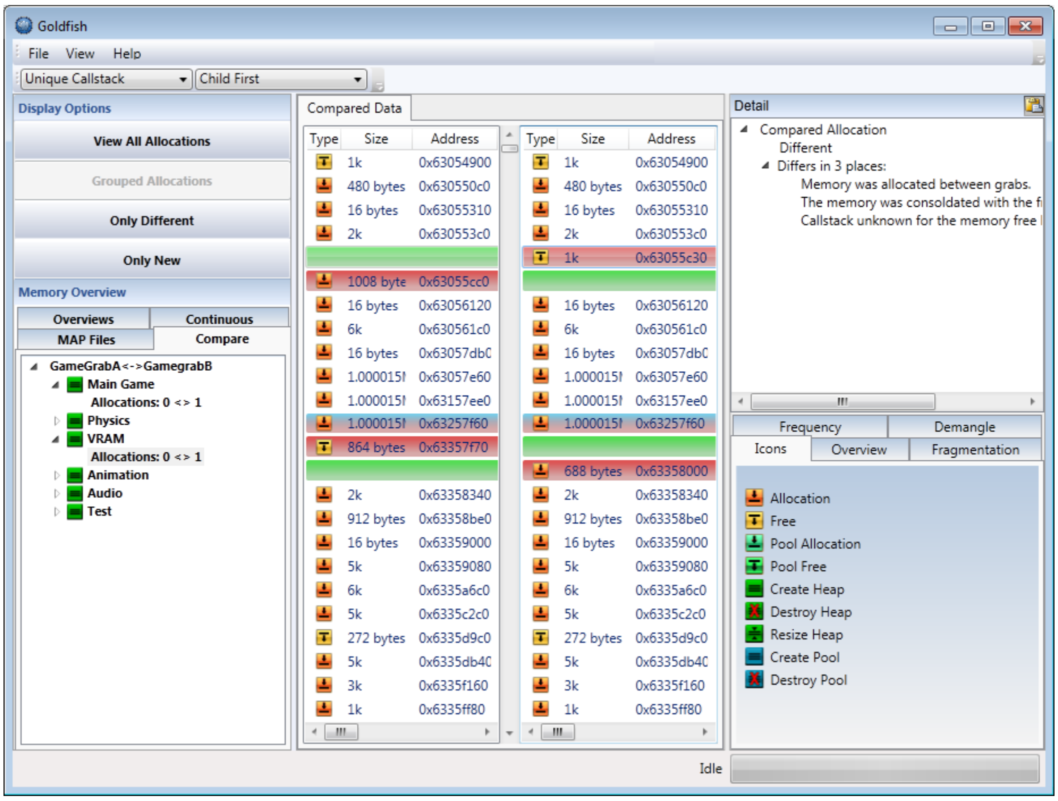


```
                 Function Name                                      Size          Count
◢ public: class Elephant::cPool * __thiscall Elephant::cMemor       1k            2
     0x12ddf90                                                      1k            2
  ◢ void __cdecl GameLoop(void)                                     21k           1322
    ◢ _main                                                         21k           1322
      ◢ _mainCRTStartup                                             21k           1322
        ◢ _mainCRTStartup                                           37.97723MB    2553
          ▷ 0x74e9339a                                              21k           1322
          ▷ _mainCRTStartup                                         42.41074MB    1582
▷ public: __thiscall Elephant::cPool::cPool(unsigned int,unsig      160k          2
▷ void __cdecl CollisionPhase(class Elephant::cHeap *)              234k          56
▷ public: void * __thiscall Elephant::cHeap::AllocateMemory(        3.638916MB    149
▷ void __cdecl LCP(class Elephant::cHeap *)                         22k           44
▷ public: void __thiscall Elephant::cHeap::FreeMemory(void *        1.946594MB    75
▷ void __cdecl ObjectUpdate(class Elephant::cHeap *)                214k          50
```

A shows the memory address of the code where an allocation takes place. Several maybe occur one function. Other functions show, depending on the mode, the functions above or below that are being called.

The size and count can be confusing. The values show the totals for the allocations with a unique callstack included from that function. For example, a program that does 10 allocations in total from malloc will show malloc as allocating 10 times with a total size of 2k (128bytes a time). If 2 different functions call malloc an equal number times they would show a count of 5 and a total allocated size of 1k.

Compared Views



- GameGrabA<->GamegrabB
  - Main Game
    - Allocations: 0 <> 1
  - Physics
  - VRAM
    - Allocations: 0 <> 1
  - Animation
  - Audio
  - Test

Type	Size	Address	Type	Size	Address
⤒	1k	0x63054900	⤒	1k	0x63054900
⤓	480 bytes	0x630550c0	⤓	480 bytes	0x630550c0
⤓	16 bytes	0x63055310	⤓	16 bytes	0x63055310
⤓	2k	0x630553c0	⤓	2k	0x630553c0
			⤒	1k	0x63055c30
⤓	1008 byte	0x63055cc0			
⤓	16 bytes	0x63056120	⤓	16 bytes	0x63056120
⤓	6k	0x630561c0	⤓	6k	0x630561c0
⤓	16 bytes	0x63057db0	⤓	16 bytes	0x63057db0
⤓	1.000015I	0x63057e60	⤓	1.000015I	0x63057e60
⤓	1.000015I	0x63157ee0	⤓	1.000015I	0x63157ee0
⤓	1.000015I	0x63257f60	⤓	1.000015I	0x63257f60
⤒	864 bytes	0x63357f70			
			⤓	688 bytes	0x63358000
⤓	2k	0x63358340	⤓	2k	0x63358340
⤓	912 bytes	0x63358be0	⤓	912 bytes	0x63358be0
⤓	16 bytes	0x63359000	⤓	16 bytes	0x63359000
⤓	5k	0x63359080	⤓	5k	0x63359080
⤓	6k	0x6335a6c0	⤓	6k	0x6335a6c0
⤓	5k	0x6335c2c0	⤓	5k	0x6335c2c0
⤒	272 bytes	0x6335d9c0	⤒	272 bytes	0x6335d9c0
⤓	5k	0x6335db40	⤓	5k	0x6335db40
⤓	3k	0x6335f160	⤓	3k	0x6335f160
⤓	1k	0x6335ff80	⤓	1k	0x6335ff80

- Compared Allocation
  - Different
  - Differs in 3 places:
    - Memory was allocated between grabs.
    - The memory was consoldated with the f
    - Callstack unknown for the memory free i

⤓	Allocation
⤒	Free
⤓	Pool Allocation
⤒	Pool Free
🟩	Create Heap
🟥	Destroy Heap
🟩	Resize Heap
🟦	Create Pool
🟥	Destroy Pool

The compared view shows in detail 2 different overviews. It does this by highlighting the values that have changed. The display panel changes to help filter the changes. Selecting an operation will display what has changed since the last grab and what has likely happened since the last time.

Map views

Map views display your MAP file in a couple of ways. One shows your functions used in the code and the other shows you the data. As normal the Memory Display window will break down the details for you.

Function View

The code view breaks all your functions down, detailing their size and location. This view is also helpful in tracking down crash locations in the code which QA may pass on to you in the absence of any other debugging information. The Memory Display windows will have the following columns:

Name	Description
Address	The starting address of the function.
End Address	The end address of the function up to but not including this address.
Size	Size in bytes of the function.
Text	The demangled name of the function.

Sometimes you will notice that functions either may not exist or share the same memory address. This is either because the function has not been called and stripped from the build or fully inlined. At other times the compiler may have determined it is the same as other functions and may have merged them together to save memory.

Data View

The data view shows you have much memory comes from the DATA and BSS sections of your program. These sections are used by the program to store any data that must exist for the length of the programs life such as globals, statics, strings, run time type information and virtual function tables. All of these can reduce the amount of memory available on start up and this view can help track down any large items you may not need.

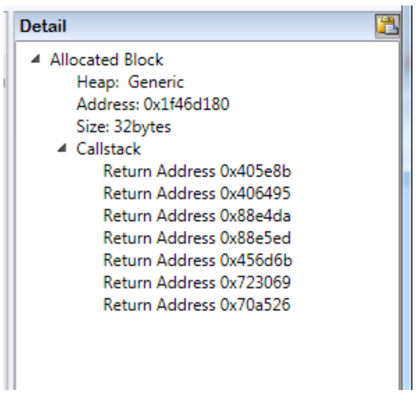
Like the Function View the columns are as follows:

Name	**Description**
Address	The starting address of the data.
End Address	Up to but not including the end address of the data.
Size	Size in bytes of the data.
Text	The name or type of the data.

Detail Panel

The detail panel is linked to the Memory Display window. Whenever you select an item from the Memory Display window more advanced details will be shown in this window such as a full callstack.

The callstack starts as near to where the allocation was made as possible and works its way up the stack.

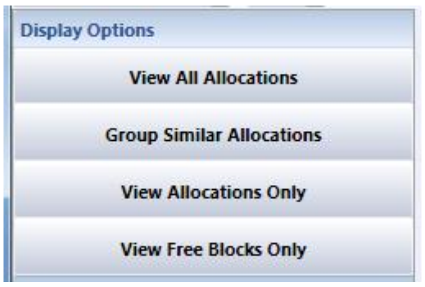

Detail
- Allocated Block
  - Heap: Generic
  - Address: 0x1f46d180
  - Size: 32bytes
  - Callstack
    - Return Address 0x405e8b
    - Return Address 0x406495
    - Return Address 0x88e4da
    - Return Address 0x88e5ed
    - Return Address 0x456d6b
    - Return Address 0x723069
    - Return Address 0x70a526

If an Overview, Continuous or Compared View has been linked to a MAP file the callstack will be changed to display the function names as well. In the case where several functions have been resolved to the same address these will open up displaying alternative functions that they may have been.

In the top right hand corner is a button that allows you to copy the view to the clipboard as text. This can then be passed on in emails or put into bug databases.

Display Options Panel

The display panel offers different ways to filter and view the memory. These will be enabled and disabled depending on which type of memory type you are viewing. MAP files will have this panel totally disabled for example while a Continuous view will only allow a couple to be selectable.



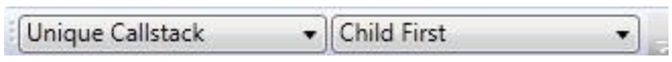
View All Allocations

This is the default view and available for Overview, Continuous and Compared Views. It will show every allocated and free area in memory tracked by Elephant at the time functions to log the information was called.

Group Similar Allocations

This view is probably the best view to see where memory allocated from similar locations has come from. Clicking on this will group those allocations together and this can be changed depending on which location you want to filter. By default this is the lowest Callstack tracked. This view is only really useful if you use the NAC or NACS libraries.

After selection this option brings the grouping toolbar into use. The combo box on the left allows you to choose how items are selected. The Unique Callstack option groups the operations by the combined callstacks, regardless of how many. Other options are based on individual callstack levels and the text stored with each option.



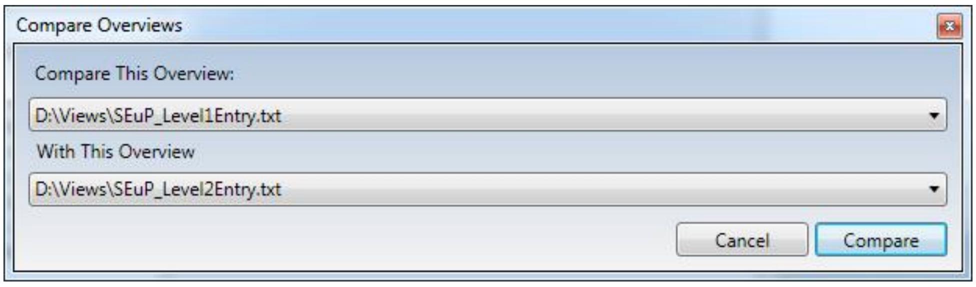
View Allocations Only
This view only shows the allocations in the list and filters out anything that isn't an allocation.

View Free Block Only
Filters out anything that isn't a free block in memory.

Comparing Overviews
Comparing Overviews is useful when tracking down memory leaks in particular.

For example you may know your games main menu leaks a bit of memory going into the first level. The main menu has to remain in memory at all times but has to reinitialize some values on return from the level. So to help you track it you call cMemoryManager::ReportAll as soon as you enter the level, exit to the main menu then load the level again. With these two Overviews you could look through every allocation in Goldfish or get Goldfish to do the hard work for you.

Load the two Overview files as normal and in the View menu select 'Compare Overviews'. You will be presented with the following dialog box which allows you to select which Overviews to compare.


Compare Overviews

Compare This Overview: D:\Views\SEuP_Level1Entry.txt

With This Overview: D:\Views\SEuP_Level2Entry.txt

The dialog allows you to select the base Overview to compare, otherwise known as Overview A to another Overview, Overview B. The Overviews must match the same memory locations and other aspects in order to be compared. Goldfish will warn you if this is not possible.

Once you have made your selection click 'Compare' and Goldfish will create a compared view.

Finding the unique allocations
Once you have a compared view you will notice the Memory Display window has a new column, Unique In Table. This shows you which Overview the allocation is unique in, ticked for Overview B and un-ticked for A.

You can filter the views by using the Unique toolbar. There are 3 unique states:

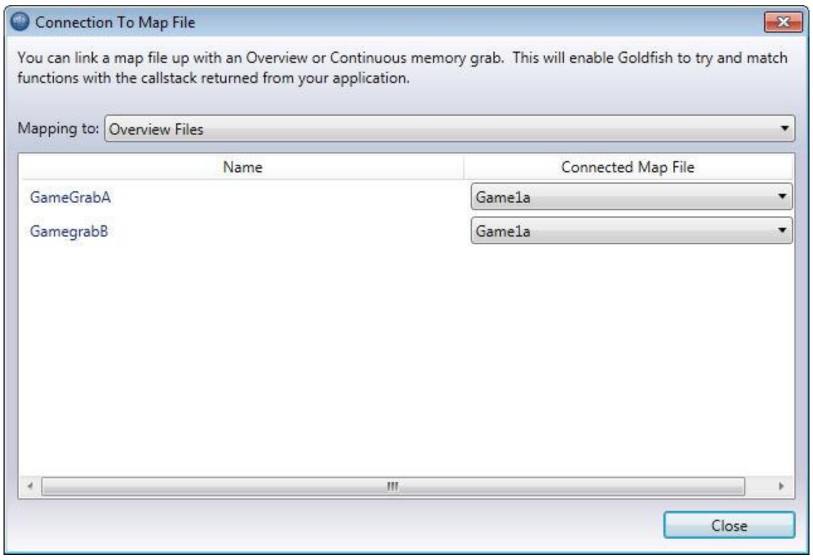
Type	Description
🟥🟩	Displays unique allocations in both Overviews
🟥	Displays the unique allocations in Overview A only.
🟩	Displays the unique allocations in Overview B only.

In the case of our menu/level example above you would only see the newer, unique allocations. These would be memory leaks in almost all situations and by using the Memory Display window and Detail panel you can track them down the exact point in your code they were allocated and then easily fixed.

Linking MAP Files

Linking MAP files to the other views is an important debugging aid. When you use the NAC or NACS libraries it will generate a callstack for the operation performed. Linking these files to a MAP file enables Goldfish to link them together and give you the function name rather than just callstack address.

Goldfish will display the linking dialog when it detects a new link is possible or you can display it at any time in the View menu by selecting ‘File Map’. The dialog will allow you to select from any available map file loaded and link it to the other types. Selecting the type by enabling the check box links it to that file.



You can link a map file up with an Overview or Continuous memory grab. This will enable Goldfish to try and match functions with the callstack returned from your application.

Mapping to: Overview Files

Name	Connected Map File
GameGrabA	Game1a
GamegrabB	Game1a

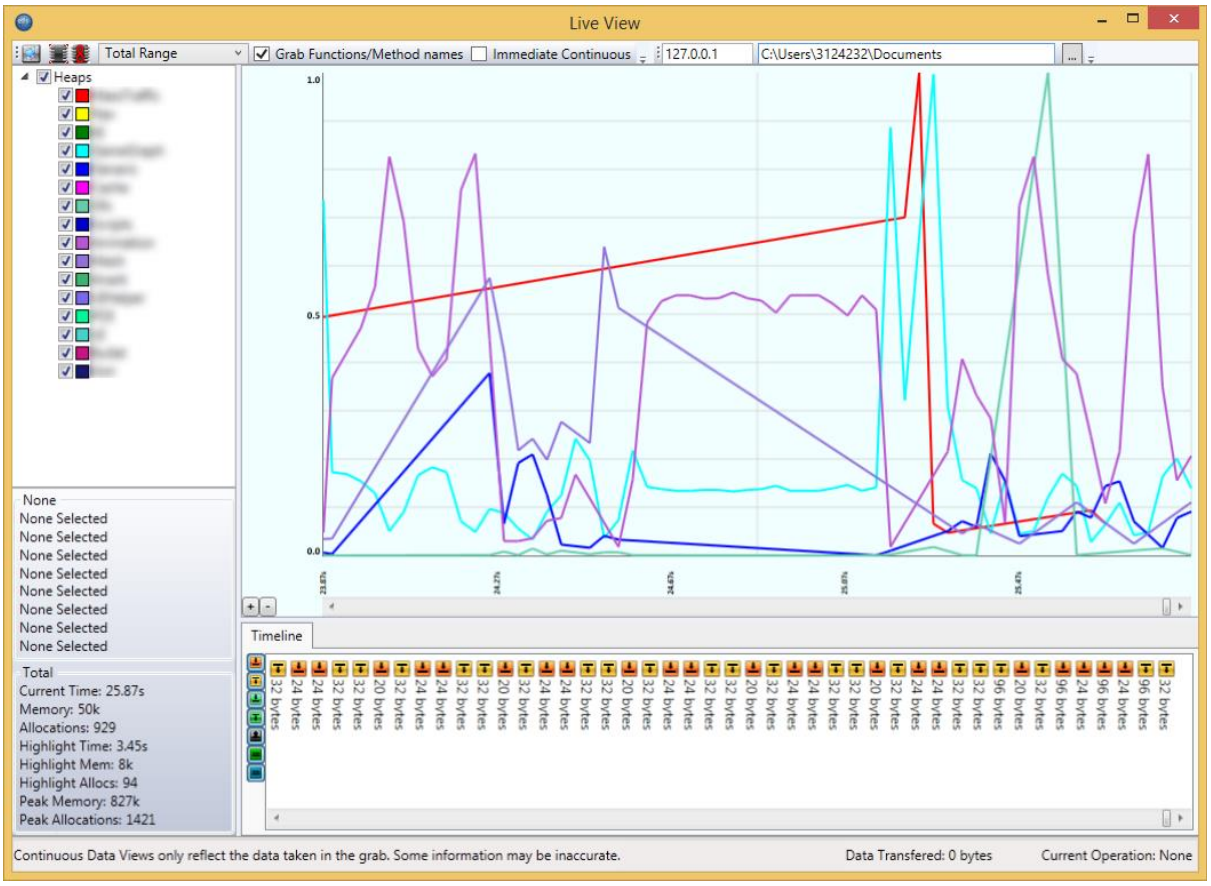
Live View

Live View is a real time view of the memory in your game or application. With Live View you can not only see what the application is doing on a frame by frame basis but also take Overview and Continuous grabs for viewing in Goldfish.

Your application should have been initialized with the Live View function, cMemoryManager::InitializeLiveView. Once initialized you can connect directly to Goldfish and Elephant does the rest.

By default the connection uses port 7133.

To open Live View use the ‘Live View’ option from the ‘View’ menu. You will be presented with the following window.



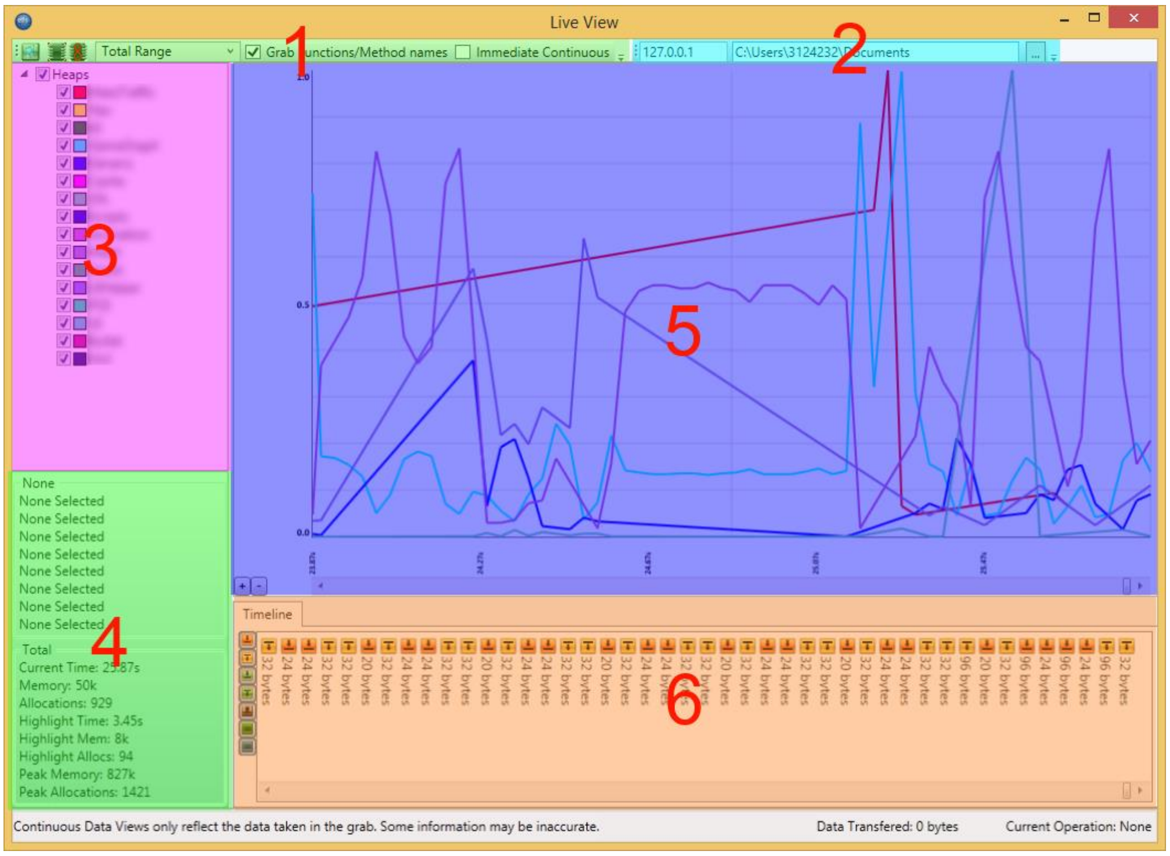
None
None Selected
None Selected
None Selected
None Selected
None Selected
None Selected
None Selected
None Selected

Total
Current Time: 25.87s
Memory: 50k
Allocations: 929
Highlight Time: 3.45s
Highlight Mem: 8k
Highlight Allocs: 94
Peak Memory: 827k
Peak Allocations: 1421

Continuous Data Views only reflect the data taken in the grab. Some information may be inaccurate. Data Transfered: 0 bytes Current Operation: None

Main Window

The main window can be broken into several areas.



None
None Selected
None Selected
None Selected
None Selected
None Selected
None Selected
None Selected
None Selected

Total
Current Time: 25.87s
Memory: 50k
Allocations: 929
Highlight Time: 3.45s
Highlight Mem: 8k
Highlight Allocs: 94
Peak Memory: 827k
Peak Allocations: 1421

Continuous Data Views only reflect the data taken in the grab. Some information may be inaccurate. Data Transfered: 0 bytes Current Operation: None

1. The tool bar is used to connect/disconnect, Overview snapshots, Continuous Timeline views and graph scaling options.
2. The current connection settings and directory to save the grabs in.
3. Heap View Panel.
4. Detail View Panel.
5. Live View Graph.
6. Allocation View Panel

Getting started

Click the button to connect to your game. Almost immediately, you should notice the Heap View Panel and Live View Graph start to fill with data.

Along the toolbar there are some other buttons which are only active once connected.

The button takes an Overview 'grab' of Elephant at that exact time. Depending on how many allocations you have in your game at the current time will depend on how long it will take. The bottom right of the Live View Window will tell you what its current activity is. The Overview will be saved to a file and stored in the directory specified in the Options menu. It will also be automatically loaded by Goldfish for instant viewing.

The will start taking a Continuous Timeline data grab. A second click will stop and save the file automatically. Unlike Overviews, the file will not automatically be loaded. These files can be large which is why we don't automatically load them.

The drop down combo box controls the Live View Graph scaling properties. This will be explained in the Live View Graph section of the documentation.

The 'Grab Function/Method names' box is only active on platforms that support it. When this is enabled Goldfish will try and capture the method names for each callstack as it is running. Upon saving it will create a MAP file. This eliminates the need to export a MAP file for some platforms.

The 'Immediate Continuous' will instantly start a collecting a Continuous Timeline data grab the moment it connects. This is primarily useful when you wish to continuously grab from the very start of your application loading to capture all operations.

Connection Settings

Panel 2 shows the connection settings.

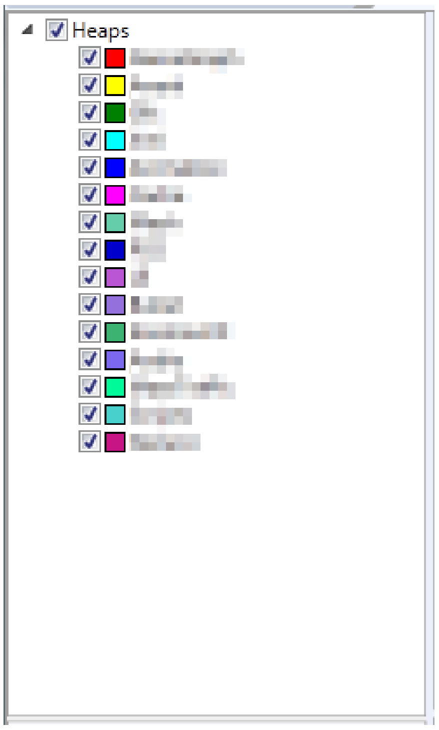
The first box allows you to input the name of the computer or an IP address that Goldfish will try to use to connect to your application with.

The second is the directory all the files will be saved too.

Heap View Panel

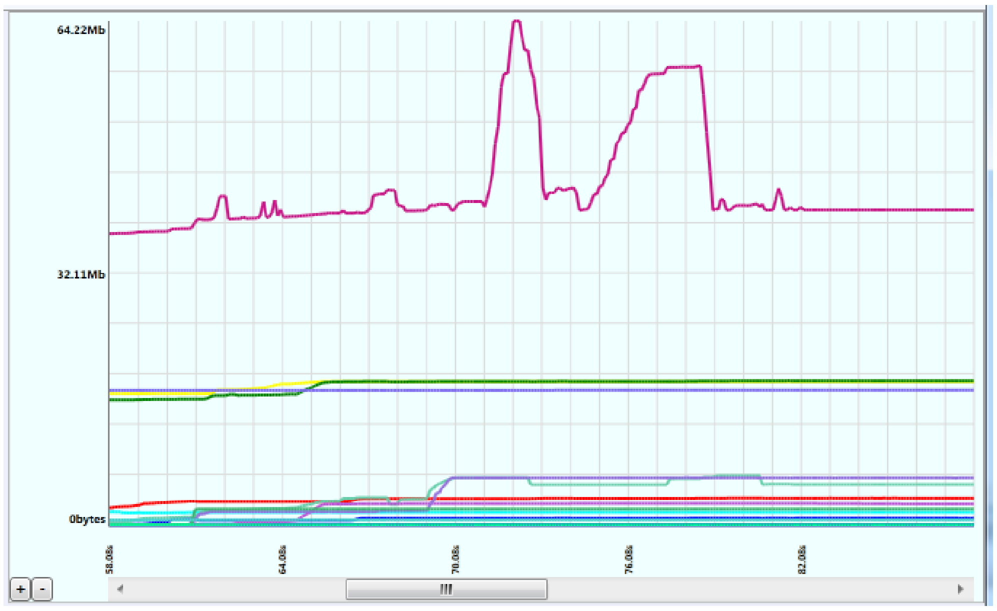
This panel gives you details on all the active Heaps inside Elephant at the current time. You can manually select a Heap to display more details about it in the Detail View Panel and display or hide it from the Live View Graph.

Each Heap is given a unique colour. When new Heaps are added to Elephant they will also appear here as well.



Live View Graph

This is the main window of Live View. It shows in real time what each Heap in Elephant is doing at a specific moment in time. Over time you will be able to see if your game has any memory trends such as logical leaks or spikes in allocations.



The combo box from tool bar has several options that will scale the data visible on the graph accordingly. These are:

- Scaled To Range – Scales the graphs to the smallest and largest amount of memory for the current zoom level. Ideal for zooming in on one specific Heap.
- Total Range – Starts at 0 and ranges to the largest Heap. Gives a good overview of memory at that time.

- Normalized Range – Scales all Heaps between 0 and 1. 0 being 0 bytes allocated, 1 being all the Heap memory is gone. This view will scale the Heaps which may be very small so you can track trends between them.

In the bottom left of the Live View Graph are two buttons. One will zoom in on the graph and one will zoom out.

Timeline View Panel

The timeline view panel shows all the operations being made within Elephant during realtime. You can add markers to help isolate what operations are happening and when.

The icons on the left hand side can be used to filter the view in real time.

Getting Started With LiveView

This guide will give you a quick tour of LiveView and show you how to grab every memory operation from the beginning of your application until you disconnect and take a full dump of every allocation at a set time for viewing later.

Setup

The application calls InitializeLiveView and sets the timeout value to -1. Starting the application will stall until a connection from LiveView is made.

1. Start Goldfish and then View->LiveView.
2. Start your application.

Performing captures

3. Because we want to start capturing all the memory operations, we are going to tick the 'Immediate Continuous' check in the menu bar along the top. This will enable us to capture every allocation from the very start.
4. Next hit the connect button. After a few seconds Goldfish will have made a connection and you will see the information come flooding in.
   a. Continuous Logging information can hit performance depending on how many memory operations occur. At any time, you can stop Continuous Logging by pressing (or if you are done). A file will automatically be create in the directory specified in the LiveView->Options menu.
   b. Pressing will resume the Continuous Logging at any time and write a new file when pressed again.
5. At some point, you may decide you wish to know how the memory at that moment in time is. To do this press and Goldfish will transfer it all over. This can take a bit of time but you will see the data transferred in the bottom right whiz up.
   a. Goldfish automatically loads these files into the main view so you can check them out later.
6. Sometimes you wish to visualize data the data a bit differently, especially when you have many Heaps.
   a. Along the top of the menu bar, you can see a drop down combo box that allows you to change the scaling options. This allows you to resize the graph to show values in proportion to one another. Just because one Heap is 4GB and the other 32MB does not mean that filling them up isn't critical. Some views do not allow you to clearly see this though.

Closing a connection

7. Once you are done you can terminate the connection by pressing .

Advanced

This section will cover some of the more advanced aspects of memory management and help you get the best out of Elephant. If you have questions email us at support@juryrigsoftware.com and we can help you out.

Replacing the Default Allocation Callbacks

Elephant comes with default system callbacks to reserve memory from the Operating System. Most of the time you will never need to replace this but occasionally it may be desirable.

Default setup includes calls to the system functions where possible. This is typically either VirtualAlloc/Free in Windows and mmap/munmap in Linux environments.

By calling InitializeAllocationCallbacks you can specify your own callbacks. See MemoryManagerDefaultAllocator, MemoryManagerDefaultFree and MemoryManagerDefaultSystemPageSize for more information on the callbacks.

Resizable or Fixed Mode

Elephant works in two modes. Fixed size of Resizable mode. Where memory needs to be strictly controlled Fixed size mode is best. On a console or embedded device. For applications where memory may spike or you do not know the maximum choose Resizable mode. The mode is determined when you initialize Elephant.

Fixed Size

In fixed size mode, Elephant will not ever consume more memory than you tell it. Because of this, it is ideal for consoles and embedded systems. When all the memory is consumed, Elephant will just stop allocating.

Resizable Mode

Resizable mode will force a heap to resize when it runs out of memory. This can incur a performance penalty as Elephant queries the OS. Heaps will then resize but the memory may not be contiguous. In this situation, the Heap will create a link to the next block.

For the user you will not notice much. The memory addresses may suddenly leap but the functions report the same information. These links make freeing memory more expensive in anything but the Master build as extra checks need to be performed to ensure it belongs to that Heap.

How the resizing occurs

When a heap runs out of memory it goes to the same allocation method used by Elephant to get its main pool of memory. First, it tries to allocate memory next to the end of the start of the current heap. This is the most efficient use internally as various elements can just be resized. If this cannot be achieved the heap creates a 'link' to the new block.

The heap never shrinks until it is destroyed at which point it releases all its memory back to the OS.

Limitations of Heap Resizing

Under some circumstances, resizing of the heap may fail. Elephant resizes using the OS system calls that returns low-level memory to Elephant. This is typically aligned to page boundaries guaranteeing a minimum gap between links is at least this size.

When the page size is very small or the Heap minimum allocation size happens to be larger than the gap in memory required to be linked, an error "`Link block to be created is too small.  Must be at least X bytes (currently X bytes)`" may be thrown.

To prevent this the minimum allocation size should be decreased to below a page size where possible. It should also be confirmed that where a custom system allocation has been defined it is correctly returning page aligned memory addresses.

Heaps

Elephant is a heap based memory allocation system to help improve the demands of many different systems and offer optimal performance options for all platforms. Elephant Heaps offer additional advantages in that they can be assigned to a particular area of memory used by the hardware such as VRAM which makes it as flexible as possible. This allows you to use any CPU accessible memory area with ease.

Non Intrusive heaps allow memory allocation to happen on areas of memory that are slow or inaccessible to the CPU. The creation and use is almost identical and designed to be easily swapped at a later date or on a per platform basis with no interruption.

Heaps with cHeap

Elephant has two types of heap, managed and self managed. Both are created with the cMemoryManager::CreateHeap functions and both get passed a cHeap::sHeapDetails structure. This is initialized with default values and is the main way of customizing how a heap functions. For example you may want just one heap to check for memory leaks or disable zero size memory allocations. You also may want to lock the heap at certain times to prevent unwanted allocations. See cMemoryManager::CreateHeap for details.

Elephant also allows for small allocations to be redirected to the small allocation heap. See Allocation Schemes for more information.

Customization

Each heap you create can be customised in a variety of ways for a variety of reasons. Even though you may have strict rules about how to manage your memory a third party library may not so you may need to relax them for this one area. Fortunately, you can do this easily when you create a heap with Elephant by passing in a cHeap::sHeapDetails structure to the function. This allows you to set various flags which the heap will then allow or disallow depending on what you may need. They are:

Flag	Default Value	What it does	Why might you need it?
uDefaultAlignment	16	Sets the minimum default alignment for the heap.	Certain types of memory or entire platforms need alignment to certain boundaries. VRAM is a common one

			which may require 128 bytes or even 4k. Setting this defaults any new allocation immediately to this value.
uMinAllocationSize	16	Sets the minimum default allocation size for the heap. Allocations smaller than this will be enlarged to this size.	Like default alignment certain systems require certain allocations of a minimum. This may be 4k for instance. Changing this value will default all new allocations to this size if they are smaller.
uMaxAllocationSize	0	Maximum allocation size allowed to be stored in the heap.	This limits the maximum allocation size a heap will allow. By default it is 0 which tells the heap to allow any size allocation.
bUseEndAllocationOnly	False	If the heap has any memory fragments this will prevent them being filled.	This flag offers the fastest possible memory allocation by only allocation at the end of the previous allocation. You might want to use this to find lingering pointers or if you are strict about your allocation procedures have total control over the potential fragmentation issues this may cause.
bReverseFreeOnly	false	The heap will only allow allocations to be freed in the reverse order they were created.	Allows allocations to only be freed in the reverse order they were created in. This is a strict method of controlling fragmentation and requires significant effort with allocating and de-allocation but can yield fragmentation free results.
bAllowNullFree	False	Allows a NULL pointer to be passed to free to pass without the heap throwing an exception.	You may want to be able to pass pointers to free even when they are null. By default Elephant doesn't allow this but it is easily enabled with this flag. Note: The C++ standard allows delete to be called with a null pointer.
bAllowZeroSizeAlllocation	False	Allows allocations of zero size to be allocated.	In other cases something may require an allocation with 0 bytes. Elephant disallows this by default. Note: The C malloc function allows this.
bAllowDestructionWithAllocations	False	Allows the heap to be destroyed with allocations still active.	In an attempt to reduce memory leaks Elephant disallows heaps to be destroyed with active allocations. This is to prevent errors further on if the memory gets overwritten or is freed later causing corruption. It does however limit the possibly of quickly destroying vast areas of memory which is something you may want.
bAllowNotEnoughSpaceReturn	False	When a heap runs out of memory and fails to allocate it will cause an exception.	Elephant will cause the heap to fire an error when it fails to allocate some memory. This has the advantage that you don't need to worry about an allocation ever failing and error checking in your code. However sometimes it is needed, for example you might need to stream data in and deal with allocation failures yourself.
bHeapIsSelfManaged	False	Tells the create function it is a self managed heap.	See the other parts of this heap section.
bEnableErrors	True	Disables any error handling.	Heaps by default warn about any errors and stop by default at the exact location. This flag disables this part of it and allows the code to continue. Error checking code will still be active so use the Master library for full speed.
bEnableErrorsAsWarnings	False	Allows all errors to continue but still log to the user TTY callback.	Turns all errors into warnings. We are not really sure why you might want to do this but you might so we give it to you as an option.

bEnableExhaustiveErrorChecking	False	Enables full error checking for the heap to ensure no buffer overruns have occurred.	Turns on full buffer overrun and error checking. By default it is disabled but if the standard sentinel checking doesn't catch any errors. This allows you to turn it on on a heap by heap basis so it doesn't cause you to much of a performance impact.
bEnableSentinelChecking	True	Performs basic sentinel checking on the currently working allocation/free operation.	Enabled by default this checks the memory around the operation you are performing. It's pretty quick but not free. For some heaps, especially those that may work out of memory where access is slow or you don't want the error checking you can turn it off.
bHeapClearing	False	Sets memory allocated or freed to a specific value.	You may want memory to be cleared to a specific value when allocated and freed. Enabling this flag will perform this. You can clear both the allocation and the free to a value of your choice by setting uHeapAllocClearValue and uHeapFreeClearValue respectively. We strongly recommend that these values are not 0. This will potentially help raise values not set correctly to be noticed earlier if you plan to disable this feature in your master build.
uResizableSize	32MB	Sets the minimum size the heap will resize to in Resizable Mode.	When a heap resizes in Resizable mode the minimum size the heap will resize is this size. This size should never be below 32MB and should increment in system page boundary sizes.
uReclaimSize	128MB	Sets the size of the minimum block that will try to be reclaimed.	RESERVED FOR FUTURE.
bAllowResizeReclaimation	False	When true the heap will automatically reclaim.	RESERVED FOR FUTURE.
systemAllocator	Default System Allocator	Called when the Heap is resized or created.	Allows the user to specialise the heap memory.
systemFree	Default System Free	Called when the Heap frees its base memory.	Allows the user to specialise the heap memory.
systemPageSize	Default System Page Size	Used throughout heap system for alignment and page sizes.	Allows the user to specialise the heap memory.
systemOpCallback	Null	Called when the systemAllocator or systemFree methods are called.	Allows the user to get feedback when one of the system allocators is called.

Managed Heaps

Managed heaps are standard heaps and fit well with stacked and multiple heap allocation strategies. They are easy to create and easy to maintain. cMemoryManager::Malloc will always allocate to the last created managed heap. Managed heaps always come out of the main pool of memory created with cMemoryManager::Initialize.

Self Managed Heaps

Elephant's greatest flexibility comes from self managed heaps. These are just like managed Heaps but you look after them and have to allocate to them directly. They do not come out of Elephants main pool of memory created with Initialize(). Creating them is almost the same as Managed Heaps except you must set the bHeapIsSelfManaged flag in sHeapDetails to true and can only use the cMemoryManager::CreateHeap(void *pMemoryAddress, jrs_u64 uHeapSize, const jrs_i8 *pHeapName, cHeap::sHeapDetails *pHeapDetails) function. This function however allows you to specify different, special areas of memory. The memory address passed in may be sound or VRAM, or slow ram for disk reading depending on the platform. It may also have been marked as non cachable or write enabled etc. It is a very powerful part of Elephant.

Non Intrusive Heaps with cHeapNonIntrusive

Non Intrusive heaps have an almost identical interface to standard cHeaps. The way they allocate memory however is different. Non Intrusive Heaps are for use when accessing memory via the CPU is slow or limited in some way.

Advantages

Non Intrusive Heaps are page based. This means that it is extremely fast to allocate memory, often faster than standard heaps. In addition, the overhead is less than that of standard Heaps. Instead of a 16 byte overhead per allocation, Non Intrusive heaps consume much less. In some situations just fractions of a byte.

Limitations

There obviously are some disadvantages. Mainly that the overhead needs to be stored somewhere. This requires a standard heap to be passed on creation of the Non Intrusive Heap. This overhead is fairly static but is about 8 megabytes per 1GB of memory available to allocate.

Waste is also much greater. The smallest size allocatable is 64bytes. Anything smaller is enlarged to 64bytes. This goes up in power of twos. Therefore, 65bytes consumes 128bytes of memory. This goes up to 8k. After that, everything becomes a multiple of 8k. For small allocations this is offset by the lower overhead.

Not all debugging options are possible. Enhanced debugging and sentinel checking is not possible due to limitations in accessing this memory on some machines. Name and callstack information also allocates directly out of the standard heap. These take space and may mean you need to increase the standard heap size.

Debugging

Non Intrusive heaps allow you to enable name and callstack information without switching library. This can consume a lot of extra memory but only in the attached standard heap. Use the cNonInstrusiveHeap::sDetails::bEnableMemoryTracking flag to enable this.

Customization

Flag	Default Value	What it does	Why might you need it?
uDefaultAlignment	16	Sets the minimum default alignment for the heap.	Certain types of memory or entire platforms need alignment to certain boundaries. VRAM is a common one which may require 128 bytes or even 4k. Setting this defaults any new allocation immediately to this value.
uMinAllocationSize	16	Sets the minimum default allocation size for the heap. Allocations smaller than this will be enlarged to this size.	Like default alignment certain systems require certain allocations of a minimum. This may be 4k for instance. Changing this value will default all new allocations to this size if they are smaller.

uMaxAllocationSize	0	Maximum allocation size allowed to be stored in the heap.	This limits the maximum allocation size a heap will allow. By default it is 0 which tells the heap to allow any size allocation.
bAllowNullFree	False	Allows a NULL pointer to be passed to free to pass without the heap throwing an exception.	You may want to be able to pass pointers to free even when they are null. By default Elephant doesn't allow this but it is easily enabled with this flag. Note: The C++ standard allows delete to be called with a null pointer.
bAllowZeroSizeAlllocation	False	Allows allocations of zero size to be allocated.	In other cases something may require an allocation with 0 bytes. Elephant disallows this by default. Note: The C malloc function allows this.
bAllowDestructionWithAllocations	False	Allows the heap to be destroyed with allocations still active.	In an attempt to reduce memory leaks Elephant disallows heaps to be destroyed with active allocations. This is to prevent errors further on if the memory gets overwritten or is freed later causing corruption. It does however limit the possibly of quickly destroying vast areas of memory which is something you may want.
bAllowNotEnoughSpaceReturn	False	When a heap runs out of memory and fails to allocate it will cause an exception.	Elephant will cause the heap to fire an error when it fails to allocate some memory. This has the advantage that you don't need to worry about an allocation ever failing and error checking in your code. However sometimes it is needed, for example you might need to stream data in and deal with allocation failures yourself.
bEnableErrors	True	Disables any error handling.	Heaps by default warn about any errors and stop by default at the exact location. This flag disables this part of it and allows the code to continue. Error checking code will still be active so use the Master library for full speed.
bEnableErrorsAsWarnings	False	Allows all errors to continue but still log to the user TTY callback.	Turns all errors into warnings. We are not really sure why you might want to do this but you might so we give it to you as an option.
`bResizable`	False	Set to true to enable resizing.	Non Intrusive heaps may be resized 16 times. Heaps will only resize with this flag set to true.
`uResizableSize`	32MB	Sets the minimum size the heap will resize to in Resizable Mode.	When a heap resizes in Resizable mode the minimum size the heap will resize is this size. This size should never be below 32MB and should increment in system page boundary sizes.
`bEnableMemoryTracking`	False	Set to True to enable name and callstacks.	Debug features on these heaps work on any library.
`uNumCallStacks`	8	Captures 8 callstack levels by default.	Allows you to customise the number of callstacks.
`systemAllocator`	Default System Allocator	Called when the Heap is resized or created.	Allows the user to specialise the heap memory.
`systemFree`	Default System Free	Called when the Heap frees its base memory.	Allows the user to specialise the heap memory.
`systemPageSize`	Default System Page Size	Used throughout heap system for alignment and page sizes.	Allows the user to specialise the heap memory.
`systemOpCallback`	Null	Called when the systemAllocator or systemFree methods are called.	Allows the user to get feedback when one of the system allocators is called.

Management
All cHeapNonIntrusive instances are self managed. The memory used by CreateHeap is never taken from Elephant Memory Managers main pool and must either be provided a pointer and a size or will use the system allocator.

Pools
Elephant also contains a concept called a Pool. Like a Heap, a Pool is used to allocate and free memory from a certain area of memory. Unlike a Heap, each allocation is of the same fixed size for entire Pool. This is one of the restrictions. However, every Heap allocation has an overhead (see Overhead for more details). All Pools in Elephant have a maximum overhead at least 4 times lower than that of a Heap. The Pool type used most often has zero overhead per allocation.

Pools therefore offer significant memory savings over Heaps for very small allocations. Pools are also very quick. Allocating and freeing is just a handful of processor instructions and the data is often local in cache.

Like a Heap, a Pool is also highly customizable.

If you use STL or similar libraries, you will often see a large improvement in both memory and perforce. Pools are ideal for code like this.

Elephant currently has two types of Pool, intrusive and non-intrusive.

Links with Heap
Each Elephant Pool is linked to a Heap when created. This is to ensure that the user is always performing the correct memory operation from the relevant area and eliminates potential corruption issues. The linked Heap is also used by Elephant to allocate any information the Pool may need allowing as many as memory can store unlike the lower fixed amount for a Heap.

Overrun Errors and Pools
Every Pool allows you to specify a Heap that the Pool will use when it runs out of space. This allows your game to continue running at the cost of potential fragmentation and performance cost. When this happens, the Pool will report a warning that it is out of space. It will only do this the once. This is to let you know that adjustments should be made for maximum performance.

Overrun buffers are not available for all types of Pool.

Sentinel, Name and Callstack
Pools, unlike Heaps do not require different libraries to enable these features. Only intrusive Pools support this and it can be enabled on a Pool by Pool basis. Enabling this will increase the size of the Pool if the minimum allocation size is to small.

Alignment
Like Heaps each Pool allocation will fit to the designated alignment. It should be noted that it may increase the size each alignment. For example aligning a Pool that contains only word size values to 128 bytes will increase the minimum size to 128 bytes.

Intrusive or Non Intrusive

An intrusive Pool is Elephants standard Pool. It takes the memory directly from the linked Heap. This memory is used to store all the data and has a zero overhead per allocation (the allocations are stored in the memory when free). It allows for sentinel, name and callstack tracing for each operation. In essence, they are a bit like a managed Heap. Each allocation can be as small as the system word size.

A Non Intrusive Pool is slightly different. The user may specify a pointer to some memory not directly accessible to the CPU. It similar to a self managed Heap only the Pool never tries to access the user specified memory. Instead, the linked Heap stores a small list of headers to map the user memory address. Typically, these types of Pool allow for use in areas like VRAM for something like small vertex arrays or to provide simple allocation for internal DMA controllers.

Thread safety

Pools are thread safe by default but this can be disabled unlike with a Heap. The reason is one of performance. Typically, thread syncing primitives are reasonably expensive. Even in the case of the quickest, these can still exceed the total time taken to allocate or free some memory from a Pool. Therefore we allow this to be disabled and potentially improve performance where it matters most.

Customisation

Pool customisation can be done in the same way as Heap customisation and with similar values. These are documented below.

Flag	**Default Value**	**What it does**	**Why might you need it?**
`uAlignment`	4 (8 in 64Bit)	Alignment default of each allocation.	Specify the default alignment for every memory allocation to come from the Pool.
`uBufferAlignment`	0	Internal memory buffer to store Pool contents will be aligned to this value.	Allows you to place the main Pool buffer on a specific Alignment value. By default it uses the Heap default value.
`pOverrunHeap`	NULL	A pointer to a valid Heap that allocations are redirected to if the Pool runs out of memory.	Sometimes during development or just to ensure your game is safe at all times you may wish to ensure any overruns will still be allocated. If it allocates memory from this Heap you can still free it with the Pool free.
`bEnableMemoryTracking`	FALSE	Enables Name and Callstack tracking for the Pool.	For the same reasons you may want it on a Heap.
`bEnableSentinel`	FALSE	Enables Sentinel tracking and checks for the Pool.	For the same reasons you may want it on a Heap.
`bThreadSafe`	TRUE	Enables or disables thread safety.	Ensuring Pools are safe can be expensive. If you are aware of no threading issues when using the Pool you can disable this.
`bAllowDestructionWithAllocations`	FALSE	Allows you to destroy a Pool with valid allocations.	In an attempt to reduce memory leaks Elephant disallows Pools to be destroyed with active allocations. This is to prevent errors further on if the memory gets overwritten or is freed later causing corruption. It does however limit the possibly of quickly destroying vast areas of

			memory which is something you may want.
`bAllowNotEnoughSpaceReturn`	FALSE	When the Pool runs out of memory and fails to allocate it will cause an exception.	Elephant will cause the Pool to fire an error when it fails to allocate some memory. This has the advantage that you don't need to worry about an allocation ever failing and error checking in your code. However sometimes it is needed, for example you might need to stream data in and deal with allocation failures yourself. If you are using an Overrun buffer the Pool wont report this.
`bEnableErrors`	TRUE	Disables any error handling.	Pools by default warn about any errors and stop by default at the exact location. This flag disables this part of it and allows the code to continue. Error checking code will still be active so use the Master library for full speed.
`bErrorsAsWarnings`	FALSE	Allows all errors to continue but still log to the user TTY callback.	Turns all errors into warnings. We are not really sure why you might want to do this but you might so we give it to you as an option.

Allocations

Overhead

Dynamic memory allocation has one small problem and that's overhead. This means every time you do an allocation it will consume a little bit of extra memory to keep track of it. There are schemes that can avoid this or at the least limit it but they come with restrictions. We have tried to make Elephant have as small an overhead as possible while keeping it fast and totally flexible but cannot eliminate it.

Each library type with Elephant can change the overhead that it uses. The standard library has a 16byte overhead per allocation. The 64bit library has to double this. If you use some of the debugging features like NAC, NACS or S libraries then it can increase this also to store some extra information needed. This means if you allocate one million 16byte allocations the over head will push that memory consumption to twice the size, 32MB's in this example but only half of that is actually in use by yourself.

Allocations smaller than 16bytes will always be rounded up to 16bytes minimum as well so smaller allocations will be padded. This again can waste memory but internally has to occur to maintain a 16byte alignment.

Be aware of this happening within your program. The reporting functions and Goldfish can help you track and keep this problem to a minimum and save you considerable memory in the process.

Alignment

Unlike some allocation systems Elephant allows for any power of two alignments you require. This is important for the wide variety of needs in a game. Some systems request minimum alignments and sizes for various systems. By default heaps align to 16 bytes but there is nothing stopping you use a wide variety of alignments in a heap.

You can change each heap to align to different values by default, leaving you free to type less or just cope with the platforms needs. For example some systems require a 4096 byte alignment for all textures.

Elephant will automatically create free blocks between allocations if the alignment has caused a large enough memory fragment. This means that on systems that require various items aligned to 1MB, any space between that and the previous allocation will have a free block created for use for something else that may fit later on.

When alignment becomes a problem

Be aware that if you require large alignments constantly you can create a lot of wasted memory. If you only allocate to 4k boundaries in 4k sizes it will consume 8k of memory in total. This is because Elephant still needs to place the memory header in memory immediately before the allocation.

It may happen that your application runs out of memory claiming it cannot allocate 4k but the memory report claims that the largest free space that can be allocated in 7k. This means that although there is not enough space available to fit an allocation on the alignment requested. In this situation we highly recommend using Goldfish to find out why you have so much wasted space and if needed employ a different scheme to manage those situations.

Types

When you allocate and free memory you can specify a type to associate with it. By default there are 15 different types you can set, Elephant uses six, JRSMEMORYFLAG_NONE, JRSMEMORYFLAG_NEW, JRSMEMORYFLAG_NEWARRAY and some reserved ones for future use and leaves the rest for the user. Types can be useful in situations where the memory needs to be marked as a certain type, say through allocations made with the C++ new[] operator and has be be removed with the delete[] operator. Elephant will ensure that the flags match when freeing occurs and warn if they dont.

New and Delete

Elephant doesn’t actually replace C++ new and delete operators for you, only what goes on behind the scenes. Of course overloading the standard global operators is easy, we will even give you the code for that (it’s in the SDK examples directory) but you are not really getting the most out of Elephant if you just do that. Sometimes you may want to overload them to pass in the function name or line number and that is trickier. Don’t forget with NAC or NACS libraries Elephant will try and find the callstack which can be traced back to the exact functions in Goldfish.

There is a reason we don’t replace them by default for you and this is because of the complexities in C++’s implementation.   There are some easy ways and some more complex ways and some other ways which can be deemed ugly. Our experience has shown us many different implementations but we have yet to find the perfect solution and one that fits perfectly for all users so we leave this for you to implement in a way suited for your project. We would love to know if there is a perfect solution but in the mean time we will give you a few of the better ones (in our opinion) that we know of with their pro’s and con’s.

Also be aware that not all systems allow for early initialization of Elephant before main() is called. Microsoft Visual C has this problem at the time this was written but other platforms allow for it. Why is it a problem? Well any global constructor's get called before main() which is where you may want to initialize Elephant cleanly. If the constructor try's to allocate memory and Elephant isn't initialized you need to either pass it through the standard allocation routines or initialize Elephant the first time an allocation is requested (or even just check if Elephant is initialized or not and pass it down a different route).

Standard Overload method

The easiest but least user friendly way is to add extra new and delete overloads. For example you could add another new that accepts line number and function. When programming each user will have to call the new overload a bit like in this example:

```
cFoo *pFoo = new(__LINE__, __FUNCTION__) cFoo;
```

Its not particuarly elegant seeing that everywhere in code but it does work.

The define method

This method was popularised by MFC that we know of and is pretty elegant. It's only a few lines and is simple to change the entire system. The one major downside is that it tends not to allow for in-place new. If you don't use it this may be a perfect solution.

Underneath all your new and delete overloads add (assuming you have overloaded new to accept a line and text string):

```
#define DEBUG_NEW new(__LINE__, __FILE__)
#define new DEBUG_NEW
```

Every new will then automatically call the overloaded new. It must be defined in a header included for all of your code that does allocations.

The replacing new with your own defines method

Another method is to simply add defines to do what you require and be more explicit in the code. For example made up company BigBadGames may have many different overloads to add alignment and allocation directly to heaps:

```
#define bbg_new new(__LINE__, __FUNCTION__)
#define bbg_newalign(alignment) new(alignment, __LINE__, __FUNCTION__)
#define bbg_newheap(elephantheap) new(elephantheap, __LINE__, __FUNCTION__)
#define bbg_delete delete
#define bbg_deletealign(alignment) delete(alignment)
#define bbg_deleteheap(elephantheap) delete(elephantheap)
...
```

BigBadGames may then explictly require programmers to use these defines over new as part of their coding standards.

This method of the three mentioned is probably the most flexible of them all at the cost of losing a clean new and delete in your code.

Performance

Elephant has been optimised for games but it will perform equally well in applications. There are various methods to get the most out of Elephant and help with general performance. There is the saying that the fastest polygons are the ones you don't draw and the same can be said for memory, the fastest allocation is the one you don't need. That said it's not possible to get away from that so we recommend that you try and limit allocations on frame by frame basis to as few as possible.

Allocating takes time, well actually in Elephants case the design normally means it's the freeing that takes time. We cannot stress enough the need to keep runtime allocations to a minimum. The PC is fairly forgiving but the same cannot be said of other platforms.

Memory Types

Some platforms allow you to change various areas of memory to other modes such as write combined or larger page sizes. These platforms can benefit from user defined heaps mapped to the different memory areas and we strongly encourage using that. For two lines of work to create these you can get 5% or more performance improvements so it is well worth it. Contact us if you want to know more.

Threading Issues

Threading can cause severe performance issues and most memory allocators will suffer if you allocate from multiple CPU's to the same heap at the same time. Elephant is no different but offers an easy and platform generic way around this. Should you find your application is performing lots of allocations from different CPU's consider giving those areas of code a separate heap to work in. Elephant can work on multiple heaps at the same time without a performance penalty (beyond that of the hardware accessing memory).

Debugging

Elephant comes with a whole lot of extras to help with debugging. Most of the time it's as simple as switching to the debugging libraries types (NAC, S, NACS) and other times you need a bit more. Goldfish will help you view the memory and track some leaks but other times it can be very tricky.

All but the Master library will warn on errors immediately to catch every foreseeable problem that can happen but is hard to track in the default system allocators. Things like double freeing of memory, simple out of memory conditions, alignment problems, null frees etc are caught and a clear warning is given.

Other system information is also desired and is easy to get. You might want to know how much memory is free in real time with practically zero overhead or how many allocations you have. You may even want to display the memory to screen showing you the potential fragmentation in the system. Things that are often difficult to get hold off are available to you.

Real time values

For onscreen debugging you may wish to know things like what heaps are active and how much memory is allocated in each at that exact time. Most of these are practically free (a few CPU cycles) to get hold of in all versions of the libraries. All our functions rough execution times are documented in the API part of this documentation so if you want to know if Elephant can give you some information instantly check there first.

Debugging individual allocations and frees

Every heap allows for precise tracking of each allocation and free. Every allocation and free has an associated address (obviously) and internally a unique ID. cHeap has numerous functions to enable tracking these to isolate where problems may be coming from. A memory overrun may have been detected, you know the allocations address but not where it is allocated from. Using the cHeap::DebugTrapOnAllocatedAddress function you can restart your game*, set this address when you have created the heap and wait for it to trap. Simply looking at the callstack will show you its exact location and to help determine what part of the code will be causing the overrun. Its as simple as that.

*Note: This only works where the memory allocation address, size and order is deterministic. Some games and platforms may not allow for this.

Memory Leaks

Detecting memory leaks is something Elephant tries to make as painless as possible. Each heap by default will not allow destruction if there are allocations still active. Depending on the allocation strategy used this can be a very quick way to pick up on leaks. For example your game may have a heap specifically for the main menu that will be destroyed when the user is about to go into a level. Elephant will instantly warn you if any allocations have been missed and report them. Of course this can be disabled if need be with the bAllowDestructionWithAllocations flag during heap creation on a heap by heap basis.

Other methods involve monitoring the number of allocations at certain points in your game and using the cHeap:: or cMemoryManager::ReportX functions. Goldfish also uses these and will help track the difficult to find leaks and leaks in heaps that have to remain between each play through.

Detecting Buffer Overruns

The best way to detect buffer overruns is in real time and using the S or NACS libraries. Elephant does this on a heap by heap basis decided by you. Link to one of the fore mentioned libraries and when you just need to enable it when you create the heaps.

You can check the memory at any time using the CheckForErrors() function which is part of cMemoryManager (checks all heaps) and cHeap (just checks this heap). That will check all allocations and even free blocks to ensure no overruns have occurred at that time. It can be time consuming so you don't want to be doing that every frame but will check everything to ensure its all stable. A buffer overrun can cause serious problems even for Elephant if some of his internals get corrupted.

Elephant can also help prevent buffer overruns and warn immediately without you needing to call anything. There are two different methods, one is pretty quick and only tracks the individual block of memory Elephant may work on and the other is the same as calling CheckForErrors.

The first method can be enabled when you create the heap by setting the sHeapDetails bEnableSentinalChecking flag or at any point when running with the cHeap::SetSentinalChecking function. This type of checking is quick and is enough for most real time checking and error detection.

The other method will check the entire heap when it performs a memory operation (it calls CheckForErrors internally). It will be very time consuming and will impact on performance but it is exhaustive. Set the bEnableExhaustiveErrorChecking flag of sHeapDetails during heap creation or use cHeap::SetExhaustiveSentinalChecking.

Error Flags

When a warning fires it will display a message saying what went wrong. For example a duplicate free of the same address. Elephant will do the best it can in trapping these errors and providing you a reason why. In most situations these are correct (for when they aren't see the next section Corruption to Elephant). By default Elephant will break on these errors (assuming you have set the error callback) to enable you to track this down quickly. More often than not just knowing where it happened is enough to allow you to quickly fix the problem.

Corruption to Elephant

Very occasionally, Elephant can be corrupted. It isn't possible to trap every error at every moment. Often this is due to sufficiently large memory overruns where internal headers get overrun. Other times it's because of dangling pointers. The Enhanced Debugging feature can track these a lot of time but sometimes it's just to severe to give you a correct warning or even prevent a serious crash.

Most of the time Elephant will report an invalid allocated or free block (or in master most likely crash in an internal binning function). We are working on ways of improving this, most likely with full dumps of your memory to Goldfish. What we are confident with is that it is very, very unlikely to be Elephant. If you suspect it is just contact support.

Enhanced Debugging

The previous section detailed the most common errors that can occur. Unfortunately, these are not the only memory bugs. Potentially serious bugs often arise and are the hardest to track down. With the advent of multi core CPU's these are even more common. Elephant has a system called Enhanced Debugging which can help track these errors.

Dangling Pointers

Elephant classes a dangling pointer like so:

A pointer to memory that has been freed and is still used by other parts of the program.

For example you have a pointer to the main player class in your game. When you exit game you free this memory. Another thread just happens to be running and updates some values in that player class, say his current health level.

In many situations, setting his health level will not have any adverse effects. The memory is overwritten and outside the range of sentinel checking Elephant provides. Therefore, this error goes unnoticed. However if some other allocations take place in between the freeing and the setting of a value serious errors can occur. Elephant will try and reuse the memory if it causes a fragment which often means it's reused fairly quickly if you allocate memory frequently. In most situations, this error may manifest itself as something quite trivial or even not at all. In other situations, it can destroy your data and potentially the integrity of Elephant as well, leading to serious errors and crashes.

Enhanced debugging aims to solve this by delaying your memory free by a small period of time. When it later comes to freeing the allocation it checks if it has been corrupted and warns you if anything has. This allows you to immediately find the cause by looking at what allocated the memory in the first place.

Potential Issues

Enhanced Debugging creates a separate thread to delay the memory freeing. This will consume some system resources.

All memory is cleared to a certain value and is checked at a later data that will affect performance. The delays will also affect the use of memory and will potentially consume more memory than not using it as the allocations will remain in memory.

When freeing large amount of allocations the Enhanced Debugging thread may cause significant stalls while they are cleared.

Memory corruption can still be missed if the corruption still takes places after it has been freed by the Enhanced Debugging thread. This time can be adjusted but it will affect the previously mentioned issues.

Pools are not tracked by the Enhanced Debugging system.

Initializing

Enhanced Debugging is initialized before the cMemoryManager:: Initialize() call. The following example demonstrates this.

```
cMemoryManager::InitializeEnhancedDebugging(true);
cMemoryManager::Get().Initialize(128 * 1024 * 1024, 0, false);
```

Live View

Live View is a feature that allows you to view memory consumption in real-time using Goldfish. It also allows you to take memory snapshots to track leaks and fragmentation from the click of a button much quicker than using the Report functions and output provided.

Live View runs a thread that transmits data to Goldfish when connected. It is very light weight and will have minimum impact on your game. Data will be transferred at a maximum rate of every 33ms.

See Goldfish Live View for more information on capturing this information.

Initialization

Live View is a one line initialize much like Enhanced Debugging and other functions which must be called before cMemoryManager:: Initialize(). This is demonstrated below.

```
cMemoryManager::InitializeLiveView();
cMemoryManager::Get().Initialize(384 * 1024 * 1024, 0, false);
```

Allocation Schemes

There are many schemes that are used in today's games to allow for management of various areas of the game. Each one is suited for different purposes and may not suit your needs exactly. Here we detail some of the most common, all of which Elephant can do with ease.

The Standard Method

If you only use new and delete or malloc and free this is really the 'standard' method. It's been around forever and works. Memory is allocated from a pool, in Elephants case one giant heap created with cMemoryManager::Initialize and you are left to do what you want with it. Nothing else needs to be done.

The upsides are that it is quick to set up and flexible since you have the entirety of system memory to play with and you will be using new and delete or malloc and free. If you haven't overloaded new and delete you will need to overload these to take advantage of Elephants features and any malloc or free calls need to be redirected to cMemoryManager::Malloc/Free.

You may have some one or two other heaps for something like VRAM but they are typically self-managed.

This method suffers from several downsides however and one most games and applications will tend to move away from. Fragmentation can be a huge issue with lots of little allocations mixed with the larger ones as well as performance in a multi-threaded environment. The size can be problematic for management of multi-platform products and needs to be monitored closely.

Small Allocation Heap

The small allocation heap is a simple extension of the standard method. When allocating all the allocations smaller than a certain size are redirected to another heap. This can improve performance for lots of very small allocations (actually this isn't really a problem for Elephant as it was designed to cope with it) and it can really improve fragmentation problems. For a very small change it may just save you if you suffer the occasional problem with these issues.

Elephant can enable the small heap with one line of code. Call cMemoryManager:: InitializeSmallHeap before Initilize and pass it the heap size and the maximum allocation size required. 256 bytes is normally a good value but you can give it any size you like. After that any calls to cMemoryManager::Malloc/Free that fit the size requirements get routed directly to the small heap.

Multiple Heaps

The multiple heap system has become common over the last 10 years as memory size has increased. It's simple in its idea that you have a heap for each system you have. For example you would have a heap each for the UI, Sound, Graphics, Controllers, Networking components of your game. Sometimes it's not many heaps but other times it's quite a few. It gives you a large amount of control over the exact size of each system and allows you to control memory for multiple platforms with ease.

Performance wise it's a good system for today's multi core CPU's and it allows for easy management of system sizes.

Downsides vary based on memory size and number of heaps. Lots of heaps can mean constant adjustments throughout development which can then adversely affect productivity for the team as one heap runs out of memory slowing up other team members until its adjusted.

Stacked

The final system is used occasionally but is more popular on consoles due to their static memory structure though falling out of favour recently. The premise is simple; you have one heap that everything at that time works in (excluding things like VRAM), when a new heap is created all operations work in that one. The previous heap at that time is resized to the last allocation and locked preventing changes.

For example the game starts up and creates its first heap for user data (configuration data), then a heap for loading screens, then a heap for the main menu. The user selects a level and it deletes the main menu heap and then creates a heap for the level. Every time the memory operations only work on the last heap created. The user data heap which may hold things like configuration data will remain there for the rest of the game.

The upside is that you don't really need to worry about memory leaks or correct destruction of data. You can just remove the heap and all the data goes with it. Then your game is in a state to immediately load and perform work on the next area of code. Another upside is the ability to dump these heaps directly to disk. These can simply be reloaded directly back to the location again, as the heap will be in exactly the same state with no fixing of up of data required.

The downsides and the reason it's falling further and further out of favour is that today's systems require memory to go to many different areas like VRAM and sound. These can complicate simple stack based methods.

Adding Additional Platforms

The base allocator in Elephant is platform agnostic. Only a few changes are required to ensure it works on another platform. More advanced features may require some additional stub code filled in. These cover thread creation, thread syncing primitives, a simple timer and if you wish to connect to Goldfish via the network, a few functions to communicate over the network.

Elephant already supports most compilers so many of the steps below can likely be skipped.

Getting Started

The first step in getting started is to create the stub functions. We have already done this for you. In the source directory, you will find a folder called 'Generic'. The files in this directory contain all the stub functions required for adding a new platform.

Copy this folder, rename it and the files inside to suit your new platform.

Adding the platform define and compiler

Adding the platform and additional compiler is performed via a series of defines in JRSCoreTypes.h. Looking in this file you will see where we define the currently supported platforms and compilers. Most platforms tend to support the GNU GCC types so you can save time here if you know it does.

Locate one of the supported compiler defines or add your own here towards the top of the file. For example if your compiler is GCC then goto around line 42 (locate `#elif defined(__GNUC__)`) and you will see platform definitions for a variety of platforms. You will need to know the specific compiler definition for your platform. Android for example is __ANDROID__. We recommend sticking with our naming conventions here. If we add support in the future for you platform you will just be able to take our changes. If you are unsure contact support for that platform.

You may also need to add some alignment defines. Microsoft Visual C is slightly different to most compilers. Use the previous values as a guide.

Finally you need to declare a few extra defines if required for your platform. If your platform does not support networking you can disable any potential compile errors by adding `#undef JRSMEMORY_HASSOCKETS` to your platform block.

If the platform you are working on is Big Endian (Elephant assumes Little Endian otherwise) you need to add `#define JRSMEMORYBIGENDIAN`. Around line 100 you will see how we define it for PPC platforms.

Around line 120 we set a define and declare sizet for 64 Bit platforms.

Finally yet importantly, you can save further time if you know your platform supports pthreads. These are the standard UNIX calls for thread creation and synchronisation. This is defined around line 105.

This is the hardest part when it comes to adding a new platform. As we have said, most compilers are already supported so it may be as simple as adding your one platform define.

Main Elephant Memory And Other Aids

Elephant calls two functions to gather the initial memory it needs in cMemoryManager::Intialize and to destroy this memory in cMemoryManager::Destroy.

These are found in JRSMemory_Generic.cpp (or the renamed file name) and called `MemoryManagerDefaultSystemAllocator` and `MemoryManagerDefaultSystemFree`.

`MemoryManagerDefaultSystemAllocator` returns a block of memory uSize (in bytes) or NULL if it fails. pExtMemoryPtr variable will be passed in to this function if the system can connect memory together.

`MemoryManagerDefaultSystemFree` destroys memory allocated by `MemoryManagerDefaultSystemAllocator`. The pointer passed in should be the value to remove. If you are unsure what to use the system malloc and free tend to work well. A size will also be passed in to indicate how much memory should be freed. This may be required for some OS's.

`MemoryManagerSystemPageSize` returns the page size used by the system. If you are unsure what this value should be 128k (1024 * 128) should be returned.

`MemoryManagerPlatformInit` is used by Elephant to initialize anything specific that the platform may need. Because Elephant is often one of the first things initialized in the system some modules may also need to be present. You can do that here.

`MemoryManagerPlatformDestroy` should clean up anything that was created in `MemoryManagerPlatformInit`.

Callstack Tracing

Should you require call stack tracing when debugging a couple of extra functions need to be filled in.

`MemoryManagerPlatformAddressToBaseAddress` should just return the uAddress passed in. Some systems relocated the code relative to the MAP file declaration. These addresses can be corrected here if need be but often that is not required.

`cHeap::StackTrace` and `cPoolBase::StackTrace` are used to gather the current call stack. pCallstack should be set to zero (0) for safety first. This can be achieved by this code `memset(pCallStack, 0, JRSMEMORY_CALLSTACKDEPTH * sizeof(jrs_sizet));`. After it should use your code to gather the addresses. You can use m_uCallstactDepth for the cHeap implementation to determine the starting depth that should be recorded.

Making Elephant Thread Safe

At this stage Elephant will not be thread safe. To make sure it is we need to add some code to JRSMemory_ThreadLock.h and to the generic file, JRSMemory_Threadlocks_*.cpp, you copied earlier.

First, we must add the new platform to the header file. For your platform you will need to include the relevant header file. You will see that all the platforms are declared individually. In the class `JRSMemory_ThreadLock` you then need to add the thread lock variable. If your platform supports pthreads then you will not have to do anything to this file.

Second, we must fill in the stub functions that exist in the cpp file. The generic file contains commented out implementations for pthreads while the windows version will contain the Microsoft implementations. Both are mutex implementations. Note that mutexes are not the fastest but are often one of the easiest to implement. If you have a lot of thread contention with Elephant this is one of the best ways to improve performance without having to change your code.

The constructor should do any creation needed for the locking primitive and the destructor any destruction. Following on from that the Lock and Unlock functions should do the same to the primitive.

Enabling Advanced Features

Elephant more advanced features requires it to be able to create and destroy some threads. Additionally it requires a timer implementation.

Threading

JRSMemory_Thread.h and the cpp file you copied earlier control threading. Like the locking primitives, you need to add your platform includes at the top. Then any thread variables to the class `cJRSThread`. Again, if you use Pthreads then you won't need to do anything in this file.

We do not fill in the constructor and destructor in any of our implementations but if you find you need to then they are the best place to do it. Next, we have the Create and Destroy functions. Create should pass in the relevant parameters and arguments as a must. Stack size is also defined and yours should not be smaller than the value we declared. It is safe for Create to start the thread immediately. Destroy should call WaitForFinish before destroying the thread.

Start is called by us but may be empty if Create starts the thread automatically. WaitForFinish is required by Destroy and should block until the thread is complete when called.

Some further functions need implementing. `SleepMilliSecond` should sleep the calling thread for the passed in MilliSeconds. `CurrentThreadId` will be required in future and should return the calling threads ID or an ID unique to that thread.

`YieldThread` and `SleepMicroSecond` and not used and will be removed in future.

Timing

Timing is required for enhanced debugging and live view features of Elephant. The timing header is JRSMemory_Timer.h and there is a relevant cpp file as well.

The timer must return the time in MilliSeconds.

The Constructor should perform any one time initialization and then call Start. The destructor should call Stop.

Start should call Reset and then set the m_TimerStarted variable to TRUE. Stop should set this variable to FALSE.

Reset should set m_ElapsedTimeMSSinceLastUpdate and m_ElapsedTimeMSTotal to zero (0).

Update should check if the m_TimerStarted variable is TRUE. If it is then the new time should be queried. The variable m_ElapsedTimeMSSinceLastUpdate should be set to the number of milliseconds since last called. Variable m_ElapsedTimeMSTotal should add m_ ElapsedTimeMSSinceLastUpdate to itself to give a total elapsed time.

GetElapsedTimeMillSec should return m_ElapsedTimeMSTotal. If the bUpdate argument is TRUE then it should call Update first.

Networking

For LiveView functionality to work and communication with Goldfish in real time some networking functions need to be implemented. Communications is via any BSD Socket compatible library.

We have created stub functions that mirror the socket library allowing you to call the platform implementations. These tend not to differ per platform other than by name. Some platforms require you to call 'closesocket' instead of 'socketclose' or even just 'close'. It also allows some of the more esoteric platforms to wrap things in a socket defined manor.

The functions that need to be filled in are `OpenSocket`, `AcceptSocket`, `CloseSocket`, `SelectSocket`, `SendSocket` and `RecvSocket`. The best way is to see the code that documents the arguments and return values in detail. Elephant deals with the endian swapping and relys on the endian defined in the very first section.

We define a `jrs_socket` type that is just a signed 64bit value. This maps to the standard socket libraries without an issue. LiveView then passes this around to the networking functions for use. For platforms that don't support sockets you can use this value as a handle to map the system specific data.

`OpenSocket` opens a server connection (Elephant, Goldfish is the client) and returns true if a connection could be made and returns the socket handle via the passed in socket argument. This function should bind and start listening to any incoming connections.

`AcceptSocket` returns the client socket id if an incoming connection is accepted. The function accepts the server socket as an argument and a value that specifies the buffersize. The function should set this buffer if possible.

`CloseSocket` closes the passed in socket. 0 will be returned if successful.

`SelectSocket` checks to see if the specified socket has any data to be read or sent. `pRreadfds` will be NOT NULL if the function is required to check for incoming data. `pWritefds` will be NOT NULL if data needs to be written. Either may be NULL or all may be NULL or NOT NULL. If the value is NOT NULL it should be set to TRUE if an action is required or FALSE otherwise. `timeout_usec` is the time out value in microseconds.

`SendSocket` sends data to the specified socket from buf of size len (bytes). Elephant tries to ensure data is at least 16 byte aligned but this cannot be guaranteed and should handle this. It should return the size of the data sent of -1 for an error.

`RecvSocket` receives data from the specified socket and copies it to the buffer argument. Len is the requested amount and it should never be more than this. Function should return the number of bytes received or -1 for an error.

Networking is the hardest thing to implement when adding an additional platform. Please contact support for additional help if you have a problem.

Hints and Tips

There are some hints and tips in this section. We can only recommend some techniques and although we recommend them they are taken from our experience over the years, therefore they may not be suitable for all applications.

How many allocations?

As few as possible! Elephant is good but like everything the lower the number of allocations the better. We have tried our best to make Elephant as quick and efficient as possible but it will not solve all your issues if you still allocate dynamically many, many times a frame for instance.

Remember that every allocation comes with an overhead. Lots of very small allocations can consume large amounts of memory in overhead, far more than a simple array would consume. This isn't just an issue with Elephant; it's an issue with every memory allocator we just want you to be aware of it.

So if you find yourself with hundreds of thousands of allocations the overhead may cause huge problems and you should look in to the benefits of pools or arrays. One million 16 byte allocations will consume an additional 16Mb of memory.

Consider limiting dynamic allocations where you know it could all be created at load time using the cHeap::Lock functions. The PC is very good with memory and you may not notice a performance problem allocating constantly but if you are porting to other platforms this could show up as a huge bottle neck. Preventing dynamic memory operations during run time can save a large amount of time later on.

Preventing CPU stalls

Multi core systems can be amazing when used well but memory operations can cause a significant problem. If you have two CPU's constantly performing memory operations to the same heap the system may perform worse than just using the one.

Like every standard system (that we know of anyway), Elephant will suffer from this if you use just one Heap. If you perform multiple allocations from different CPU's try and use multiple Heaps. Elephant can safely perform memory operations to multiple heaps at the same time without stalling other CPU's. Using heaps effectively can be the difference between a smooth 60fps game and one running at 15fps.

Understand other libraries memory use

Just because your company's code performs minimal memory allocations doesn't mean other libraries follow the same rules. Using a 'list' or 'binary tree' as an example, they may need to allocate nodes on the fly. Some Standard Template Library's will request memory that you may not know about for example. If you want to monitor their use give them their own heap.

Multi-platform development

Developing for multiple platforms can be difficult and memory is often one of the major factors. Different platforms have different amounts allocated to different areas. Even compiled code can be vastly larger depending on the processor.

Our experience has shown that one method works particularly well and that is to allocate a set amount for the game that all platforms must use (i.e the generic data). After that the remaining memory can be given to graphics and sound which can be tailored for the platform in question. i.e extra texture memory. Use heaps to split this if those platforms use one giant area of memory for everything. It will help keep the budgets in check.

The important thing is to stick to these budgets. It's much easier to increase the sizes later than it is to find memory to cut a day before you master.

Frequently Asked Questions

Why operations such as creating a Heap are not thread safe?

For some operations, we found it unlikely that you would be creating heaps at the same time from multiple threads. They are not that fast so not really designed for constant changing around. They are also often only created and destroyed in known places that are single threaded anyway so we decided to keep the system simple by not making them thread safe.

Why do I keep running out of memory?

For applications or tools have you initialized Elephant with a fixed memory size?

Can I allocate using 'High' memory from within a Heap?

No, there is no such thing as 'High' memory in Elephant. If we get enough requests we shall add it. The reason we didn't add it was simply because our test showed a very small difference in actual fragmentation saving on systems with reasonable amounts of memory, if any. Elephant is good at filling any fragments efficiently and simplifies the code without needing to know which allocation has to temporally into high memory. The other downside is that it would fix you to fixed heap sizes so we knew where to take high memory from. This can cause problems for systems which use lots of managed heaps.

Elephant says I have enough memory but fails to allocate, why?

It's likely down to alignment. See Allocations, Alignment.

I cannot initialize Elephant with very large values, why?

Some Operating Systems have a problem allocating one large, contiguous block of memory. Even though the OS reports several GB free it may not be able to issue this in one go. This is typically an issue with Win32 applications requiring more than 1.5GB of memory. Depending on the OS or installed applications you may be able to allocate said space on one computer but not on another.

There are a couple of ways of dealing with this. The simplest is to use the Resizable Memory Mode. The other is to use self-managed heaps and pass memory created directly from the OS. With this method, you will need to be careful of the OS failing to allocate the memory.

I want to use Elephant with tools and do not know my memory requirements?

The best solution is to use the Resizable Memory feature.

SDK

In this section

This section contains reference information for Memory Manager's API. Elephant Memory Manager is written in C++ using nothing more advanced than classes and operator overloading enabling it to be compatible with almost every C++ compiler in use today. The API exists in the Elephant namespace.

We additionally give you a rough estimate of the performance cost and if the function is thread safe.

Callbacks

Several callbacks can be registered by the user to enable warnings, errors and information to enable features in a way suitable for the user.

Callbacks

MemoryManagerTTYOutputCB
MemoryManagerErrorCB
MemoryManagerOutputToFile
MemoryManagerDefaultAllocator
MemoryManagerDefaultFree

Constants And Defines

Various constants and defines are implemented that can be used by the user.

Defines:

The largest available memory allocation amount to Elephant can initialize too. See cMemoryManager::Initialize.

JRSMEMORYINITFLAG_LARGEST

Flags passed to the allocation and free functions that are available for use.
JRSMEMORYFLAG_NONE
JRSMEMORYFLAG_NEW
JRSMEMORYFLAG_NEWARRAY

Flags reserved by Elephant.
JRSMEMORYFLAG_EDEBUG
JRSMEMORYFLAG_POOL
JRSMEMORYFLAG_POOLMEM
JRSMEMORYFLAG_RESERVED0
JRSMEMORYFLAG_RESERVED1
JRSMEMORYFLAG_RESERVED2
JRSMEMORYFLAG_RESERVED3

MemoryManagerTTYOutputCB Callback

This is the user callback to generate output to the TTY window. In other words this is a custom callback passed any logging errors that may be generated by Elephant.

Pass this to the cMemoryManager::InitializeCallbacks function.

Syntax

```
void (*MemoryManagerTTYOutputCB)(const jrs_i8 *pString);
```

Parameters

pString – A null terminated string containing the warning generated by Elephant.

Return Value

Remarks

The calling heap will be called from a thread safe environment however several threads may potentially call this function at the same time.

MemoryManagerErrorCB Callback

This is a user called callback called by Elephant when an error has occurred. This is an ideal place to put custom code to trigger in the event of a serious error. For example in MSVC __debugbreak() will cause the program to halt. Checking the TTY is recommended to see what error occurred.

Pass this to the cMemoryManager::InitializeCallbacks function.

Syntax

```
void (*MemoryManagerErrorCB)(const jrs_i8 *pError, jrs_u32 uErrorID);
```

Parameters

pError – NULL terminated string of the error if you require it.
uErrorID – Unique Error identifyer.

Return Value

Remarks

The calling heap will be called from a thread safe environment however several threads may potentially call this function at the same time.

MemoryManagerOutputToFile Callback

This user callback is used to write data to a file to generate views for use in Goldfish. This should be a text file.

Pass this to the cMemoryManager::InitializeCallbacks function.

Syntax

```
void (*MemoryManagerOutputToFile)(void *pData, int size, const jrs_i8
*pFilePathAndName, jrs_bool bAppend);
```

Parameters

pData – Data to be written to a file.
size – Size of the data in bytes to be written.
pFilePathAndName – NULL terminated string containing the full path and file name of the file to have data witten too. Recommended .txt extension.
bAppend – TRUE when the file should be appended too. FALSE if the file should be recreated.

Return Value

Remarks

Thread safety is not required by the user. Elephant will ensure this function cannot be called from multiple threads.

MemoryManagerDefaultAllocator Callback

This custom callback is what Elephant will call during Initialize to allocate the main pool for managed heaps. Internally this function may just call malloc to allocate a large block of memory for internal Heap pools. Elephant provides a default function so this is not needed unless you wish to override how Elephant is provided with this memory.

Set with cMemoryManager::InitializeAllocationCallbacks.

Syntax

```
void *(*MemoryManagerDefaultAllocator)(jrs_u64 uSize, void
pExtPointer);
```

Parameters

uSize – Size in bytes of the total amount of memory.
pExtPointer – Pointer that the system allocator may use to try and find continuous memory.

Return Value

A valid memory pointer if the memory of size uSize was successfully allocated. NULL otherwise.

Remarks

Only called from one place within Elephant so will inherently be thread safe.

MemoryManagerDefaultFree Callback

This custom callback is what Elephant will call during Destroy to free the main pool for managed heaps. Internally this function may just call free. Elephant provides a default function so this is not needed unless you wish to override how Elephant destroys this memory.

Set with cMemoryManager::InitializeAllocationCallbacks.

Syntax

```
void (*MemoryManagerDefaultFree)(void *pFree, jrs_u64 uSize);
```

Parameters

pFree – Memory address to be freed.
uSize – Amount of memory to be freed for pFree address.

Return Value

Remarks

Only called from one place within Elephant so will inherently be thread safe.

MemoryManagerDefaultSystemPageSize Callback

This custom callback is what Elephant used to return the system page size and alignment. It should map to the system calls used by the other callbacks.

Set with cMemoryManager::InitializeAllocationCallbacks.

Syntax

```
jrs_sizt (*MemoryManagerDefaultSystemPageSize)(void);
```

Parameters

Return Value

Page size in bytes.

Remarks

Only called from one place within Elephant so will inherently be thread safe.

cMemoryManager::Get Function

This function is the global singleton accessor for all cMemoryManager functions except those marked in the remarks as static.

Syntax

```
cMemoryManager &cMemoryManager::Get(void)
```

Parameters

Return Value

A reference to the memory manager singleton.

Remarks

Non-locking but thread safe. It only returns a reference to the global cMemoryManager.

cMemoryManager::InitializeCallbacks Function

This function accepts C style function pointers to pass back errors from the memory manager. Must be called before Initialize.

Syntax

```
void cMemoryManager::InitializeCallbacks(MemoryManagerTTYOutputCB
TTYOutput, MemoryManagerErrorCB ErrorHandle, MemoryManagerOutputToFile
FileOutput)
```

Parameters

TTYOutput - This callback outputs text to the log TTY of your choice.
ErrorHandle - This callback executes any error handling.
FileOutput - This callback handles file output.

Return Value

Remarks

Static function of cMemoryManager. Do not access with cMemoryManager::Get.

cMemoryManager::InitializeAllocationCallbacks Function

This function accepts C style function pointers to change the default system allocator and free for Initialize. Must be called before Initialize.

By default this does not need to be called as Elephant has defaults which can be used. However if you wish to override the default pool allocation for some reason use this function.

Syntax

```
void
cMemoryManager::InitializeAllocationCallbacks(MemoryManagerDefaultAlloc
ator DefaultAllocator, MemoryManagerDefaultFree DefaultFree,
MemoryManagerDefaultSystemPageSize DefaultPageSize)
```

Parameters

DefaultAllocator - Default allocation call.
DefaultFree - Default free call.
DefaultPageSize – Default callback for returning the size of the system page.

Return Value

Remarks

Static function of cMemoryManager. Do not access with cMemoryManager::Get.

cMemoryManager::InitializeSmallHeap Function

This function initializes the small heap which will then automatically be used by Malloc. Recommended size is atleast 256k. Must be called before Initialize.

Syntax

```
void cMemoryManager::InitializeSmallHeap(jrs_sizet uSmallHeapSize,
jrs_u32 uMaxAllocSize, cHeap::sHeapDetails *pDetails)
```

Parameters

uSmallHeapSize - The size of the heap.

uMaxAllocSize - The maximum allocation size to divert to the small heap.

pDetails – Allows customisation parameters of the small heap. Default NULL.

Return Value

Remarks

Static function of cMemoryManager. Do not access with cMemoryManager::Get.

cMemoryManager:: InitializeContinuousDump Function

Initializes the filename of the continuous dump. Must be called before Initialize. The string will be passed to your file callback whenever logging is enabled. You do not need to call this unless you want to enable continuous logging.

Syntax

```
void cMemoryManager::InitializeContinuousDump(const jrs_i8
*pFileNameAndPath)
```

Parameters

pFileNameAndPath - The full name and path of the file you want as a continuous log file.

Return Value

Remarks

Static function of cMemoryManager. Do not access with cMemoryManager::Get.

cMemoryManager::InitializeLiveView Function

Initializes the live view network thread. The thread is of low priority and is polled roughly every uMilliSeconds. The thread will always be active but do very little if Goldfish isnt running.

Syntax

```
void cMemoryManager::InitializeLiveView(jrs_u32 uMilliSeconds, jrs_u32
uPendingContinuousOperations, jrs_bool bAllowUserPostInit, jrs_i32
iExternalConnectionTimeOutMS = 0, jrs_u16 uPort = 7133)
```

Parameters

uMilliSeconds - Time to poll in MilliSeconds. Minimum time is 16. Any lower time will be capped to this.

uPendingContinuousOperations – Default 16384. The maximum number of operations that can be queued before stalling Elephant (and subsequently the application). More memory is consumed when this is larger on initialization.

bAllowUserPostInit – Default false. Allows this feature to be initialized after cMemoryManager::Initialize in the situations where early initialization isnt possible.

iExternalConnectionTimeOutMS – Default 0. During Initialize Elephant will stall for this amount of time until a connection from Goldfish has been made. Continues after time out. -1 will wait indefinitly for a connection.

uPort – Default 7133. Default port number to run Live View connection on.

Return Value

Remarks

Static function of cMemoryManager. Do not access with cMemoryManager::Get.

cMemoryManager::InitializeEnhancedDebugging Function

Initializes the enhanced debugging thread. The thread is of low priority and is polled roughly every 16ms. This thread checks the memory has not been used after it was freed.

It does however mean that some memory will remain 'valid' until it has been checked and confirmed as unused else where. This means that memory consumption will be slightly higher than normal but is the ultimate debugging aid when tracking down memory corruption, especially over multiple threads.

Must be called before Initialize.

Syntax

```
void cMemoryManager::InitializeEnhancedDebugging(jrs_bool
bEnhancedDebugging, jrs_u32 uDeferredTimeMS, jrs_u32 uMaxAllocation ,
jrs_bool bAllowUserPostInit)
```

Parameters

bEnhancedDebugging - true to enable enhanced debugging features of Elephant.

uDeferredTimeMS - The amount of time in MilliSeconds before Elephant decides to remove the memory (approximate - it may be longer but never shorter). Default 66.

uMaxAllocation - Maximum number of allocations to cache at any one time. When this limit is reached stalls may occur. Default is 32k.

bAllowUserPostInit – Default false. Allows this feature to be initialized after cMemoryManager::Initialize in the situations where early initialization isnt possible.

Return Value

Remarks

Static function of cMemoryManager. Do not access with cMemoryManager::Get.

cMemoryManager::Initialize Function

Initializes Elephant to a size that you request which all managed heaps will take their memory from.

This function will automatically create the heap if needed and try automatically get as much memory as requested if the maximum size isn't available by reducing the size 64k until it can be achieved. By default Initialize will try and use all the memory allocated for one heap. The default heap, if created will be called "DefaultHeap".

bFindMaxClosestToSize defaults to true.
pMemory defaults to NULL.

The parameter uMemorySize determines if Elephant should use a fixed size of work in resizable mode. In fixed size mode, Elephant will never allocate more memory than uMemorySize. In resizable mode, Elephant will dynamically resize Heaps when it runs out of memory. Note: bFindMaxClosestToSize must be FALSE to enter resizable mode.

Syntax

```
jrs_bool cMemoryManager::Initialize(jrs_u64 uMemorySize, jrs_u64
uDefaultHeapSize, jrs_bool bFindMaxClosestToSize, void *pMemory)
```

Parameters

uMemorySize - The total size in bytes. 16byte multiple. Set to JRSMEMORYINITFLAG_LARGEST to place Elephant in resizable mode.

uDefaultHeapSize - The size of the default heap. Set to JRSMEMORYINITFLAG_LARGEST to use all of the available memory. Set to 0 if you want no default heap created.

bFindMaxClosestToSize - TRUE to find the largest amount of memory available if it it cannot allocate uMemorySize. FALSE to fail if uMemorySize cannot be allocated. This must be FALSE in resizable mode.

pMemory – Instead of calling internal allocation routines Elephant will use this address instead.

Return Value

TRUE if successfully created.
FALSE for failure.

Remarks

Not thread safe.

cMemoryManager::Destroy Function

Destroys Elephant. This will close all heaps which may or may not check for memory leaks depending on the creation options of that heap.

Syntax

```
jrs_bool cMemoryManager::Destroy(void)
```

Parameters

Return Value

TRUE if successfully destroyed.
FALSE for failure.

Remarks

Not thread safe.

cMemoryManager::GetMaxNumHeaps Function

Retrieves the maximum number of managed heaps Elephant can handle.

Syntax

```
jrs_u32 cMemoryManager::GetMaxNumHeaps(void) const
```

Parameters

Return Value

The maximum number of heaps.

Remarks

Safe to call from multiple threads only if CreateHeap and DestroyHeap are not being called at the time otherwise this value may potentially be out by one.

cMemoryManager::GetMaxNumUserHeaps Function

Retrieves the maximum number of self managed heaps Elephant can handle. Default inbuilt maximum is 32.

Syntax

```
jrs_u32 cMemoryManager::GetMaxNumUserHeaps(void) const
```

Parameters

Return Value

The maximum number of heaps.

Remarks

Thread safe.

cMemoryManager::CreateHeap Function

Creates a managed heap only, self managed heaps will fail creation. A unique name may be specified and it will automatically come out of Elephants memory created in Initialize. Use the pHeapDetails to customize the operation of the heap for example to allow zero size allocations.

cHeap::sHeapDetails. This structure is used to customize the heap with various values. See the advanced section for details on its initial defaults and operation of each.

Syntax

```
cHeap *cMemoryManager::CreateHeap(jrs_u64 uHeapSize, const jrs_i8
*pHeapName, cHeap::sHeapDetails *pHeapDetails)
```

Parameters

uHeapSize - The total size in bytes of the heap. 16byte multiple.
pHeapName - Null terminated string for the heap name. Smaller than 32bytes.
pHeapDetails - A valid pointer to heap details which can be created locally on the stack. Its lifetime is only the length of CreateHeap function. NULL will use the sHeapDetails values.

Return Value

Valid cHeap if successful.
NULL for failure.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::CreateHeap Function

Creates a managed or self managed heap. A unique name may be specified and it will automatically come out of Elephants memory created in Initialize if and only pHeapDetails has the self managed flag set to FALSE. Otherwise the heap will be marked as self managed. Use the details to customize the operation of the heap for example to allow zero size allocations. Use this type of create heap to direct to other predefined areas of the system for types such as VRAM .

cHeap::sHeapDetails. This structure is used to customize the heap with various values. See the advanced section for details on its initial defaults and operation of each.

Syntax

```
cHeap *cMemoryManager::CreateHeap(void *pMemoryAddress, jrs_u64
uHeapSize, const jrs_i8 *pHeapName, cHeap::sHeapDetails *pHeapDetails)
```

Parameters

pMemoryAddress - The memory address to create the heap on. May be other CPU accessible areas like VRAM.
uHeapSize - The total size in bytes of the heap. 16byte multiple.
pHeapName - Null terminated string for the heap name. Smaller than 32bytes.
pHeapDetails - A valid pointer to heap details which can be created locally on the stack. Its lifetime is only the length of CreateHeap function.

Return Value

Valid cHeap if successful.
NULL for failure.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::CreateHeapNonIntrusive Function

Creates a Non Intrusive Heap. A unique name must be specified. Use the details to customize the operation of the heap for example to allow zero size allocations. Use this type of create heap to direct to other predefined areas of the system for types such as VRAM .

cHeapNonIntrusive::sHeapDetails. This structure is used to customize the heap with various values. See the advanced section for details on its initial defaults and operation of each.

Syntax

```
cHeapNonInstrusive *cMemoryManager::CreateHeapNonIntrusive(void
*pMemoryAddress, jrs_u64 uHeapSize, cHeap *pHeap, const jrs_i8
*pHeapName, cHeapNonInstrusive::sHeapDetails *pHeapDetails)
```

Parameters

pMemoryAddress - The memory address to create the heap on. May be other CPU accessible areas like VRAM.
uHeapSize - The total size in bytes of the heap. 8k multiple multiple.
pHeap – The default heap to manage the Non Intrusive Heaps overheads.
pHeapName - Null terminated string for the heap name. Smaller than 32bytes.
pHeapDetails - A valid pointer to heap details which can be created locally on the stack. Its lifetime is only the length of CreateHeap function.

Return Value

Valid cHeapNonInstrusive if successful.
NULL for failure.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::CreateHeapNonIntrusive Function

Creates a Non Intrusive Heap. A unique name must be specified. Use the details to customize the operation of the heap for example to allow zero size allocations. Use this type of create heap to direct to other predefined areas of the system for types such as VRAM .

cHeapNonIntrusive::sHeapDetails. This structure is used to customize the heap with various values. See the advanced section for details on its initial defaults and operation of each.

This Non Intrusive heap uses the standard system allocators or a custom allocator specified in cHeapNonIntrusive::sHeapDetails to obtain the memory.

Syntax

```
cHeapNonInstrusive *cMemoryManager::CreateHeapNonIntrusive(jrs_u64
uHeapSize, cHeap *pHeap, const jrs_i8 *pHeapName,
cHeapNonInstrusive::sHeapDetails *pHeapDetails)
```

Parameters

uHeapSize - The total size in bytes of the heap. 8k multiple multiple.
pHeap – The default heap to manage the Non Intrusive Heaps overheads.
pHeapName - Null terminated string for the heap name. Smaller than 32bytes.
pHeapDetails - A valid pointer to heap details which can be created locally on the stack. Its lifetime is only the length of CreateHeap function.

Return Value

Valid cHeapNonInstrusive if successful.
NULL for failure.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::Reclaim Function

Experimental – please report any issues

Reclaims as much memory as possible and returns it to the OS. Elephant must be running in resizable mode.

This method runs through all heaps and tries to resize.

It can only reclaim the blocks that were first created when the heap was resized to make it bigger.

Syntax

```
void cMemoryManager::Resize(void)
```

Parameters

None

Return Value

None

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::ResizeHeap Function

Resizes a heap to the size you require. For self managed heaps you must be very careful when resizing that the expansion does not trample other memory. Elephant has no way of detecting this. Elephant will disallow resizing of managed heaps if it is not the last on the stack.

Syntax

```
jrs_bool cMemoryManager::ResizeHeap(cHeap *pHeap, jrs_u64 uSize)
```

Parameters

pHeap - Valid pointer to a cHeap.
uSize - The total size in bytes to resize the heap. 16byte multiple.

Return Value

TRUE if successful.
FALSE for failure.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::ResizeHeapToLastAllocation Function

Resizes a heap to the end of the greatest allocated block. Elephant will disallow resizing of self managed heaps. Use cHeap::Resize instead.

Syntax

```
jrs_bool cMemoryManager::ResizeHeapToLastAllocation(cHeap *pHeap)
```

Parameters

pHeap - Valid pointer to a cHeap.

Return Value

TRUE if successful.
FALSE for failure.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::FindHeap Function

Finds a heap based on its name.

Syntax

```
cHeap *cMemoryManager::FindHeap(const jrs_i8 *pHeapName)
```

Parameters

pHeapName - Null terminated string with the heap's name.

Return Value

Valid cHeap if successful.
NULL if heap couldn't be found.

Remarks

Not thread safe with any other cMemoryManager functions.

cMemoryManager::FindNonIntrusiveHeap Function

Finds a Non Intrusive heap based on its name.

Syntax

```
cHeapNonInstrusive *cMemoryManager::FindNonInstrusiveHeap(const jrs_i8
*pHeapName)
```

Parameters

pHeapName - Null terminated string with the heap's name.

Return Value

Valid cHeapNonInstrusive if successful.
NULL if heap couldn't be found.

Remarks

Not thread safe with any other cMemoryManager functions.

cMemoryManager::GetHeap Function

Gets a managed heap using an index.

Syntax

```
cHeap *cMemoryManager::GetHeap(jrs_u32 iIndex)
```

Parameters

iIndex - Index of heap, starting at 0.

Return Value

Valid cHeap if successful.
NULL if heap couldn't be found.

Remarks

Not thread safe with any cMemoryManager functions.

cMemoryManager::GetUserHeap Function

Gets a self managed heap from an index.

Syntax

```
cHeap *cMemoryManager::GetUserHeap(jrs_u32 iIndex)
```

Parameters

iIndex - Index of heap, starting at 0.

Return Value

Valid cHeap if successful.
NULL if heap couldn't be found.

Remarks

Not thread safe with any cMemoryManager functions.

cMemoryManager::GetNIHeap Function

Gets a non instrusive heap from an index.

Syntax

```
cHeapNonInstrusive *cMemoryManager::GetNIHeap(jrs_u32 iIndex)
```

Parameters

iIndex - Index of heap, starting at 0.

Return Value

Valid cHeapNonInstrusive if successful.
NULL if heap couldn't be found.

Remarks

Not thread safe with any cMemoryManager functions.

cMemoryManager::DestroyHeap Function

Destroys the specified heap. The heap can be managed or self managed. On destruction the heap may warn you if there are any allocations remaining depending on the settings specified at creation time.

Syntax

```
jrs_bool cMemoryManager::DestroyHeap(cHeap *pHeap)
```

Parameters

pHeap - Valid cHeap.

Return Value

TRUE if successfully destroyed.
FALSE otherwise.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::DestroyNonIntrusiveHeap Function

Destroys the specified heap. The heap can be managed or self managed. On destruction the heap may warn you if there are any allocations remaining depending on the settings specified at creation time.

Syntax

```
jrs_bool cMemoryManager::DestroyNonIntrusiveHeap(cHeapNonInstrusive
*pHeap)
```

Parameters

pHeap - Valid cHeapNonIntrusive.

Return Value

TRUE if successfully destroyed.
FALSE otherwise.

Remarks

Not thread safe with any other cMemoryManager functions except heap creation, resize and destruction functions.

cMemoryManager::ResetHeapStatistics Function

Resets any heap statistics that have accrued since either Initialization or the last time this function was called.

Syntax

```
void cMemoryManager::ResetHeapStatistics(void)
```

Parameters

Return Value

Remarks

Not thread safe with any cMemoryManager functions.

cMemoryManager::CreatePool Function

Creates a memory pool. It will be intrusive and so could negatively affect performance depending on type of memory and the system it is used on.

Syntax

```
cPool *cMemoryManager::CreatePool(jrs_u32 uElementSize, jrs_u32
uMaxElements, const jrs_i8 *pPoolName, sPoolDetails *pDetails, cHeap
*pHeap)
```

Parameters

uElementSize - The size of the Element to be stored. Minimum size is sizeof(jrs_sizet) and the pool may expand. See cPool documentation.

uMaxElements - Maximum number of elements to be stored.

pDetails - Pointer to heap details. NULL will just use the defaults.

pHeap - Pointer to a heap with which to take the Pool memory from. NULL will call cMemoryManager::Malloc and obey allocation there.

Return Value

Valid cPool. NULL if there is a failure.

Remarks

cMemoryManager::CreatePoolNonIntrusive Function

Creates a non-intrusive memory pool. With a non-intrusive pool the main memory comes from one area and the main headers come from another. This is useful if the memory you want to store data in is particularly slow or not directly accessible.

The cPoolNonIntrusive will never access pDataPointer and only return you valid pointers to data elements within side this.

Syntax

```
cPoolNonIntrusive *cMemoryManager::CreatePoolNonIntrusive(jrs_u32
uElementSize, jrs_u32 uMaxElements, const jrs_i8 *pPoolName, void
*pDataPointer, jrs_u64 uDataPointerSize, sPoolDetails *pDetails, cHeap
*pHeap)
```

Parameters

uElementSize - The size of the Element to be stored. Minimum size is sizeof(jrs_sizet) and the pool may expand. See cPool documentation.
uMaxElements - Maximum number of elements to be stored.
pDetails - Pointer to heap details. NULL will just use the defaults.
pHeap - Pointer to a heap with which to take the Pool memory from. NULL will call cMemoryManager::Malloc and obey allocation there.
pHeaderHeap - Pointer to a heap with which the headers come from.

Return Value

Valid cPoolNonIntrusive. NULL if there is a failure.

Remarks

cMemoryManager::DestroyPool Function

Destroys any type of pool created by CreatePool.

Syntax

```
void cMemoryManager::DestroyPool(cPoolBase *pPool)
```

Parameters

uElementSize - The size of the Element to be stored. Minimum size is sizeof(jrs_sizet) and the pool may expand. See cPool documentation.
uMaxElements - Maximum number of elements to be stored.
pDetails - Pointer to heap details. NULL will just use the defaults.
pHeap - Pointer to a heap with which to take the Pool memory from. NULL will call cMemoryManager::Malloc and obey allocation there.

Return Value

Remarks

cMemoryManager::Malloc Function

Replaces standard system malloc but allows for more advanced allocation parameters such as alignment. Malloc will automatically send the allocation to the last created heap, unless a small heap is specified. If a small heap is specified and the allocation size is <= the maximum allocation size of the small heap the small heap will try to allocate. Flags are set by the user. It can be one of JRSMEMORYFLAG_xxx or any user specified flags > JRSMEMORYFLAG_RESERVED3 but smaller than or equal to 15, values greater than 15 will be lost and operation of Malloc is undefined. Input text is limited to 32 chars including terminator. Strings longer than this will only store the last 31 chars.

Syntax

```
void *cMemoryManager::Malloc(jrs_sizet uSizeInBytes, jrs_u32
uAlignment, jrs_u32 uFlag, jrs_i8 *pText, const jrs_u32 uExternalId)
```

Parameters

uSizeInBytes - Size in bytes. Minimum size will be 16bytes unless the heap settings have set a larger minimum size.

uAlignment - Default alignment is 16bytes unless heap settings have set a larger alignment. Any specified alignments must be a power of 2. Passing 0 will set it to the correct alignment for the heap.

uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. See description for more details.

pText - NULL terminating text string to associate with the allocation.

uExternalId – Application defined value to associate with the allocation.

Return Value

Valid pointer to allocated memory.
NULL otherwise.

Remarks

This function is part thread safe. No CreateHeap, DestroyHeap or Resize heap functions should not be called while this is operating or errors may occur. Safe to call all cHeap and other cMemoryManager functions.

cMemoryManager::Malloc Function

Replaces standard system malloc but allows for more advanced allocation parameters such as alignment. Malloc will automatically send the allocation to the last created heap, unless a small heap is specified. If a small heap is specified and the allocation size is <= the maximum allocation size of the small heap the small heap will try to allocate. Allocation flag defaults to JRSMEMORYFLAG_NONE and passed in string is NULL. In NAC or NACS libraries the string will be given a default value of 'Unknown'.

Syntax

```
void *cMemoryManager::Malloc(jrs_sizet uSizeInBytes, jrs_u32
uAlignment)
```

Parameters

uSizeInBytes - Size in bytes. Minimum size will be 16bytes unless the heap settings have set a larger minimum size.
uAlignment - Default alignment is 16bytes unless heap settings have set a larger alignment. Any specified alignments must be a power of 2. Passing 0 will set it to the correct alignment for the heap.

Return Value

Valid pointer to allocated memory.
NULL otherwise.

Remarks

This function is part thread safe. No CreateHeap, DestroyHeap or Resize heap functions should not be called while this is operating or errors may occur. Safe to call all cHeap and other cMemoryManager functions.

cMemoryManager::Realloc Function

Reallocs a block of memory. Interally it is often a free and a malloc that is performed. It will perform this within the same heap that the original allocation came from.

Syntax

```
void *cMemoryManager::Realloc(void *pMemory, jrs_sizet uSizeInBytes,
jrs_u32 uAlignment, jrs_u32 uFlag, jrs_i8 *pText, const jrs_u32
uExternalId)
```

Parameters

pMemory – Pointer to the previous block of allocated memory. If NULL is passed in Malloc is called instead.

uSizeInBytes - Size in bytes. Minimum size will be 16bytes unless the heap settings have set a larger minimum size.

uAlignment - Default alignment is 16bytes unless heap settings have set a larger alignment. Any specified alignments must be a power of 2. Passing 0 will set it to the correct alignment for the heap.

uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. See description for more details.

pText - NULL terminating text string to associate with the allocation.

uExternalId – Application defined value to associate with the allocation.

Return Value

Valid pointer to allocated memory.
NULL otherwise.

Remarks

This function is part thread safe. No CreateHeap, DestroyHeap or Resize heap functions should not be called while this is operating or errors may occur. Safe to call all cHeap and other cMemoryManager functions.

cMemoryManager::Free Function

Frees allocated memory. Elephant will automatically search for the heap the allocation was allocated from (See note). DeAllocation flag defaults to JRSMEMORYFLAG_NONE. In NAC or NACS libraries the string associated with the free will be given a default value.

Note: If heaps are created from memory allocated within other heaps this function may corrupt memory. If you are allocating self managed heaps or heaps with addresses from memory allocated by Elephant in Initialize it is recommended that you do not use this function to avoid potential errors.

Syntax

```
void cMemoryManager::Free(void *pMemory, jrs_u32 uFlag)
```

Parameters

pMemory - Valid memory address. A NULL value may cause a warning depending on the settings of the heap.
uFlag - Defaults to JRSMEMORYFLAG_NONE but can be other user specified values.

Return Value

Remarks

This function is part thread safe. No CreateHeap, DestroyHeap or Resize heap functions should be called while this is operating or errors may occur. Safe to call all cHeap and other cMemoryManager functions.

cMemoryManager::Free Function

Frees allocated memory. Elephant will automatically search for the heap the allocation was allocated from (See note). DeAllocation flag defaults to JRSMEMORYFLAG_NONE. In NAC or NACS libraries the string associated with the free will be given a default value otherwise a custom value may be assigned. See Malloc for futher flag values. This function is identical to the other cMemoryManager::Free except that it allows a string to be associated with the free.

Note: If heaps are created from memory allocated within other heaps this function may corrupt memory. If you are allocating self managed heaps or heaps with addresses from memory allocated by Elephant in Initialize it is recommended that you do not use this function to avoid potential errors.

Syntax

```
void cMemoryManager::Free(void *pMemory, jrs_u32 uFlag, jrs_i8 *pText,
const jrs_u32 uExternalId)
```

Parameters

pMemory - Valid memory address. A NULL value may cause a warning depending on the settings of the heap.
uFlag - Defaults to JRSMEMORYFLAG_NONE but can be other user specified values.
pText - 32 byte including terminator value string to be associated with the free.
uExternalId – Application defined value to associate with the allocation.

Return Value

Remarks

This function is part thread safe. No CreateHeap, DestroyHeap or Resize heap functions should be called while this is operating or errors may occur. Safe to call all cHeap and other cMemoryManager functions.

cMemoryManager::LogMemoryMarker Function

Logs a memory marker to the continuous logging information. Goldfish will use this value as a marker also for its continuous views.

Syntax

```
void cMemoryManager::LogMemoryMarker(const jrs_i8 *pMarkerName)
```

Parameters

pMarkerName - NULL terminated string of the name of the marker to log.

Return Value

Remarks

Thread safe.

cMemoryManager::SizeofAllocation Function

Returns the size of the allocated memory allocated with Malloc or one of the heap allocation functions.

Syntax

```
jrs_sizet cMemoryManager::SizeofAllocation(void *pMemory) const
```

Parameters

pMemory - Valid memory address. A NULL value may cause a warning depending on the settings of the heap.

Return Value

Size in bytes of the allocation.

Remarks

Thread safe as long as the memory pointer will not be modified during function use. This function is very fast and can be used in real time.

cMemoryManager::SizeofAllocationAligned Function

Returns the size of the allocated memory allocated with Malloc or one of the heap allocation functions. This function returns the actual aligned size to the next header in memory. This means that it may be considerably larger than the size returned by cMemoryManager::SizeofAllocation

Syntax

```
jrs_sizet cMemoryManager::SizeofAllocation(void *pMemory) const
```

Parameters

pMemory - Valid memory address. A NULL value may cause a warning depending on the settings of the heap.

Return Value

Size in bytes of the allocation.

Remarks

Thread safe as long as the memory pointer will not be modified during function use. This function is very fast and can be used in real time.

cMemoryManager::SizeofAllocatedBlock Function

Returns the size of the allocation header. This will be 16 bytes (32 for 64bit) normally but different library configurations change this.

Syntax

```
jrs_sizet cMemoryManager::SizeofAllocatedBlock(void) const
```

Parameters

Return Value

Size in bytes of the allocation header.

Remarks

Thread safe. This function is very fast and can be used in real time.

cMemoryManager::SizeofFreeBlock Function

Returns the size of the free header. This is typically always 16bytes larger than the allocation header.

Syntax

```
jrs_sizet cMemoryManager::SizeofFreeBlock(void) const
```

Parameters

Return Value

Size in bytes of the free header.

Remarks

Thread safe. This function is very fast and can be used in real time.

cMemoryManager:: FindHeapFromMemoryAddress Function

Returns the heap where the memory address lies. Can be useful for finding out general information of where memory came from. Does not need to be an allocated address, any will do.

Syntax

```
cHeap *cMemoryManager::FindHeapFromMemoryAddress(void *pMemory)
```

Parameters

pMemory - Memory address to search heaps against.

Return Value

A valid cHeap pointer if a heap contains the memory location. NULL otherwise.

Remarks

Not thread safe if Create/Destroy Heap functions can take place.

cMemoryManager:: FindNIHeapFromMemoryAddress Function

Returns the non intrusive heap where the memory address lies. Can be useful for finding out general information of where memory came from. Does not need to be an allocated address, any will do.

Syntax

```
cHeapNonIntrusive *cMemoryManager::FindNIHeapFromMemoryAddress(void
*pMemory)
```

Parameters

pMemory - Memory address to search heaps against.

Return Value

A valid cHeapNonIntrusive pointer if a heap contains the memory location. NULL otherwise.

Remarks

Not thread safe if Create/Destroy Heap functions can take place.

cMemoryManager::GetDefaultHeap Function

Gets the default heap allocations will go into when calling cMemoryManager::Malloc.

Syntax

```
cHeap *cMemoryManager::GetDefaultHeap(void)
```

Parameters

Return Value

Valid cHeap pointer or NULL.

Remarks

Thread safe.

cMemoryManager::GetSystemPageSize Function

Gets the default system page size.

Syntax

```
jrs_sizet cMemoryManager::GetSystemPageSize(void) const
```

Parameters

Return Value

System page size in bytes

Remarks

Thread safe.

cMemoryManager::GetLVPortNumber Function

Gets the default system page size.

Syntax

```
jrs_sizet cMemoryManager::GetLVPortNumber(void) const
```

Parameters

Return Value

The current set port number

Remarks

Thread safe.

cMemoryManager:: SetMallocDefaultHeap Function

Sets the default heap allocations will go into when calling cMemoryManager::Malloc.

Syntax

```
Void cMemoryManager::GetDefaultHeap(cHeap *pHeap)
```

Parameters

pHeap – Set the default heap.

Return Value

None

Remarks

cMemoryManager::GetFreeUsableMemory Function

Returns the amount of free usable memory that Elephant has to allocate managed heaps from. This does not include memory taken with self managed heaps.

Syntax

```
jrs_u64 cMemoryManager::GetFreeUsableMemory(void) const
```

Parameters

Return Value

Remarks

Not thread safe during any cMemoryManager operations.

cMemoryManager::IsInitialized Function

Used to determine if the memory manager is initialized or not. No other operations may occur if it is not initlaized.

Syntax

```
jrs_bool cMemoryManager::IsInitialized(void) const
```

Parameters

TRUE if initialized.
FALSE otherwise.

Return Value

Remarks

Thread safe. This function is very fast and can be used in real time.

cMemoryManager::IsLoggingEnabled Function

Determines if logging is currently enabled for Elephant as a whole. If this is disabled no heap logging will occur either.

Syntax

```
jrs_bool cMemoryManager::IsLoggingEnabled(void) const
```

Parameters

Return Value

TRUE if logging is currently enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used in real time.

cMemoryManager::CheckForErrors Function

Runs a full check on all heaps to see if their are any errors like overruns. Must be using NACS or S libraries for this to be performed. It is thorough but not a quick operation.

Syntax

```
void cMemoryManager::CheckForErrors(void)
```

Parameters

Return Value

Remarks

Thread safe. This function is very can be very slow.

cMemoryManager::ReportAll Function

Does a full memory report (statistics and allocations in memory order) to the user TTY callback. If you want to report these to a file full path and file name to the function. The generated file is an Overview file for use in Goldfish. The file is in a compatible CSV format for loading into a spread sheet for self analysis. When connected to a debugger TTY output may cause this function to take a lot longer than just writing to a log file can take. Time is proportionate to the number of allocations.

Syntax

```
void cMemoryManager::ReportAll(const jrs_i8 *pLogToFile, jrs_bool
includeFreeBlocks, jrs_bool displayCallStack)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

includeFreeBlocks – Default FALSE. When logging to the debugger output window include/exclude free blocks.

displayCallStack – Default FALSE. Output will display the callstack. On supported platforms it will try and resolve the function names to the actual value rather than display memory addresses.

Return Value

Remarks

Thread safe. This function is can be very slow.

cMemoryManager::ReportStatistics Function

Reports basic statistics about all heaps to the user TTY callback. Use this to get quick information on when need that is more detailed that simple heap memory realtime calls.

Syntax

```
void cMemoryManager::ReportStatistics(void)
```

Parameters

Return Value

Remarks

Thread safe. This function is very fast and can very slow.

cMemoryManager::ReportAllocationsMemoryOrder Function

Reports on all the allocations in every heap. This function is like ReportAll but doesnt output the statistics. Calling this function with a valid file name will create an Overview file for use in Goldfish.

Syntax

```
void cMemoryManager::ReportAllocationsMemoryOrder(const jrs_i8
*pLogToFile, jrs_bool includeFreeBlocks, jrs_bool displayCallStack)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

includeFreeBlocks – Default FALSE. When logging to the debugger output window include/exclude free blocks.

displayCallStack – Default FALSE. Output will display the callstack. On supported platforms it will try and resolve the function names to the actual value rather than display memory addresses.

Return Value

Remarks

Thread safe. This function is very fast and can be very slow.

cMemoryManager::ReportAllToGoldfish Function

When an active LiveView connetion is made to Goldfish this performs a full Overview Grab.

Syntax

```
void cMemoryManager::ReportAllToGoldfish(void)
```

Parameters

Return Value

Remarks

Thread safe. This function will call ReportAll.

cMemoryManager::ReportContinuousStartToGoldfish Function

When an active LiveView connetion is made to Goldfish this starts a Continuous Grab.

Syntax

```
void cMemoryManager::ReportContinuousStartToGoldfish(void)
```

Parameters

Return Value

Remarks

Thread safe.

cMemoryManager::ReportContinuousStopToGoldfish Function

When an active LiveView connetion is made to Goldfish this stops a Continuous Grab previous started with ReportContinuousStartToGoldfish.

Syntax

```
void cMemoryManager::ReportContinuousStopToGoldfish(void)
```

Parameters

Return Value

Remarks

Thread safe.

cHeap::AllocateMemory Function

Main call to allocate memory directly to the heap. This function is called by cMemoryManager::Malloc. Call this to avoid the extra overhead of malloc or to redirect to a specific heap. This function is the main allocation functionto use. Flags are set by the user. It can be one of JRSMEMORYFLAG_xxx or any user specified flags > JRSMEMORYFLAG_RESERVED3 but smaller than or equal to 15, values greater than 15 will be lost and operation of AllocateMemory is undefined. Input text is limited to 32 chars including terminator. Strings longer than this will only store the last 31 chars. Alignment must be a power of two however a 0 will default to the default allocation size set when the heap was created.

Syntax

```
void *cHeap::AllocateMemory(jrs_sizet uSize, jrs_u32 uAlignment,
jrs_u32 uFlag, const jrs_i8 *pName, const jrs_u32 uExternalId)
```

Parameters

uSize - Size in bytes. Minimum size will be 16bytes unless the heap settings have set a larger minimum size.
uAlignment - Default alignment is 16bytes unless heap settings have set a larger alignment. Any specified alignments must be a power of 2. Setting 0 will default to the minimum requested alignment of the heap.
uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. See description for more details.
pName - NULL terminating text string to associate with the allocation. May be NULL.
uExternalId – An application specific number. Default 0.

Return Value

Valid pointer to allocated memory.
NULL otherwise.

Remarks

Thread safe. This function is moderate in operation speed.

cHeap::ReAllocateMemory Function

Reallocates memory. Generally should be avoided if at all possible as most of the time it will just Free and Allocate except for some circumstances. If you are constantly reallocating it is recommended to see if there is a better alternative. Flags are set by the user. It can be one of JRSMEMORYFLAG_xxx or any user specified flags > JRSMEMORYFLAG_RESERVED3 but smaller than or equal to 15, values greater than 15 will be lost and operation of AllocateMemory is undefined. Input text is limited to 32 chars including terminator. Strings longer than this will only store the last 31 chars.

Syntax

```
void *cHeap::ReAllocateMemory(void *pMemory, jrs_sizet uSize, jrs_u32
uAlignment, jrs_u32 uFlag, const jrs_i8 *pName, const jrs_u32
uExternalId)
```

Parameters

uSize - Size in bytes. Minimum size will be 16bytes unless the heap settings have set a larger minimum size.
uAlignment - Default alignment is 16bytes unless heap settings have set a larger alignment. Any specified alignments must be a power of 2.
uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. See description for more details.
pName - NULL terminating text string to associate with the allocation. May be NULL.
uExternalId – An application specific number. Default 0.

Return Value

Valid pointer to allocated memory.
NULL otherwise.

Remarks

Thread safe. This function is has a moderate execution time.

cHeap::FreeMemory Function

Frees memory allocated from AllocateMemory or Malloc. If the allocation does not come from this heap FreeMemory will fail and/or warn first depending on heap settings. FreeMemory will free the memory that matches the flag of the allocation. Text is limited to 32 chars including terminator. Strings longer than this will only store the last 31 chars. In certain situations FreeMemory may free memory from a heap which has been allocated from with in other heaps. In this rare situation memory corruption may occur.

Syntax

```
void cHeap::FreeMemory(void *pMemory, jrs_u32 uFlag, const jrs_i8
*pName, const jrs_u32 uExternalId)
```

Parameters

pMemory - Valid memory pointer. A null pointer may only be passed in if bAllowNullFree is set in the heap CreateHeap details.
uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. Must match the flag set at allocation time.
pName - NULL terminating text string to associate with the allocation. May be NULL.
uExternalId – An application specific number. Default 0.

Return Value

Remarks

Thread safe. This function has a moderate execution time.

cHeap::Resize Function

Resizes the heap. It is up to the user to ensure the size being enlarged is valid other wise memory overruns could occur. It is recommended to avoid problems to use cMemoryManager::ResizeHeap instead as this performs more safety checks.

Syntax

```
jrs_bool cHeap::Resize(jrs_sizet uSize)
```

Parameters

uSize - Size to resize the heap too. 16byte multiples.

Return Value

TRUE if successful.
FALSE for failure.

Remarks

Thread safe.

cHeap::Reclaim Function

Experimental – please report any issues

Reclaims as much memory as possible and returns it to the OS. Elephant must be running in resizable mode.

It can only reclaim the blocks that were first created when the heap was resized to make it bigger.

Syntax

```
void cHeap::Resize(void)
```

Parameters

None

Return Value

None

Remarks

Thread safe unless heap is removed during operation.

cHeap::IsAllocatedFromThisHeap Function

Checks if the memory pointer was allocated from this heap by checking the heaps memory range. May get confused if memory is located within other heaps.

Syntax

```
jrs_bool cHeap::IsAllocatedFromThisHeap(void *pMemory) const
```

Parameters

pMemory - Valid memory pointer to a block of memory.

Return Value

TRUE if memory belongs to heap.
FALSE otherwise.

Remarks

Thread safe as long as heap is not removed during operation. This function is fast and can be used real time.

cHeap:: IsAllocatedFromAttachedPoolFunction

Checks if the memory pointer was allocated from this heap by checking the heaps memory range. May get confused if memory is located within other heaps.

Syntax

```
jrs_bool cHeap:: IsAllocatedFromAttachedPool(void *pMemory) const
```

Parameters

pMemory - Valid memory pointer to a block of memory.

Return Value

TRUE if memory belongs to an attached pool.
FALSE otherwise.

Remarks

Thread safe as long as heap is not removed during operation.

cHeap::GetPoolFromAllocatedMemory Function

Returns the pool that the memory was allocated from. This will recusively search for the pool list on the heap so will get slower as more heaps are added.

Syntax

```
cPool cMemoryManager:: GetPoolFromAllocatedMemory (void *pMemory)
```

Parameters

pMemory - Valid memory address.

Return Value

Valid cPool pointer if successful.
NULL otherwise.

Remarks

Partially thread safe.

cHeap::GetSizeOfLargestFragment Function

Returns the size largest free fragment of memory in the cheap

Syntax

```
jrs_sizet cMemoryManager:: GetSizeOfLargestFragment(void)
```

Parameters

None

Return Value

Size of the largest largest

Remarks

Thread safe while heap is valid.

cHeap:: IsOutOfMemoryReturnEnabled Function

Returns the size largest free fragment of memory in the cheap

Syntax

```
jrs_bool cMemoryManager:: IsOutOfMemoryReturnEnabled (void)
```

Parameters

None

Return Value

True if Out of Memory return flag is enabled. False otherwise.

Remarks

Thread safe while heap is valid.

cHeap:: GetTotalAllocations Function

Returns the total number of active allocations in the heap.

Syntax

```
jrs_u32 cMemoryManager::GetTotalAllocations(void)
```

Parameters

None

Return Value

The total number of active allocations in the heap.

Remarks

Thread safe while heap is valid.

cHeap:: GetMaxAllocations Function

Returns the size largest free fragment of memory in the cheap

Syntax

```
jrs_ u32 cMemoryManager:: GetMaxAllocations (void)
```

Parameters

None

Return Value

The total number of active allocations in the heap

Remarks

Thread safe while heap is valid.

cHeap:: GetTotalFreeMemoryFunction

Returns the total free memory in the heap.

Syntax

```
jrs_sizet cMemoryManager::GetTotalFreeMemory(void)
```

Parameters

None

Return Value

Total amount of free memory.

Remarks

Thread safe while heap is valid.

cHeap:: GetResizableSize Function

Returns the minimum size the heap will resize in resizable mode. This property is set in sHeapDetail structure on Heap creation.

Syntax

```
jrs_sizet cMemoryManager:: GetResizableSize(void)
```

Parameters

None

Return Value

The size in bytes of the minimum resize size.

Remarks

Thread safe while heap is valid.

cHeap:: GetNumberOfLinks Function

Returns the total number of active links with in a heap. During Resizable mode hard links may be inserted into the heaps to join memory locations up. This returns how many fragments the heap has been brocken into internally.

Syntax

```
jrs_u32 cMemoryManager::GetNumberOfLinks(void)
```

Parameters

None

Return Value

The total number of active links in a heap.

Remarks

Not thread safe.

cHeap:: GetPool

Returns the total free memory in the heap.

Syntax

```
cPoolBase *GetPool(cPoolBase *pPool)
```

Parameters

pPool – Valid pool pointer.

Return Value

The next linked pool in the heap. NULL if it was the end.

Remarks

Thread safe while heap is valid.

cHeap::AreErrorsEnabled Function

Returns if error checking is enabled or not for the heap. This is determined by the bEnableErrors flag of sHeapDetails when creating the heap.

Syntax

```
jrs_bool cHeap::AreErrorsEnabled(void) const
```

Parameters

Return Value

TRUE if errors are enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::AreErrorsWarningsOnly Function

Returns if all errors have been demoted to just warnings for the heap. This is determined by the bErrorsAsWarnings flag of sHeapDetails when creating the heap. If error checking enabled this will be ignored internally.

Syntax

```
jrs_bool cHeap::AreErrorsWarningsOnly(void) const
```

Parameters

Return Value

TRUE if errors are demoted to warnings.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::IsMemoryManagerManaged Function

Returns if this heap is managed by Elephant or if it is self managed.

Syntax

```
jrs_bool cHeap::IsMemoryManagerManaged(void) const
```

Parameters

Return Value

TRUE if managed by Elephant.
FALSE if managed by the user.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetMaxAllocationSize Function

Return the maximum allocation size allowed by the heap. This is set by the uMaxAllocationSize flag of sHeapDetails when creating the heap.

Syntax

```
jrs_sizet cHeap::GetMaxAllocationSize(void) const
```

Parameters

Return Value

Maximum allowed allocation size.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetMinAllocationSize Function

Return the minimum allocation size allowed by the heap. This is set by the uMinAllocationSize flag of sHeapDetails when creating the heap. The heap will automatically resize the minimum size to match this which is different functionality to the maximum size which will fail. This is because some systems may require a minimum size internally for some memory items due to page or boundary sizes.

Syntax

```
jrs_sizet cHeap::GetMinAllocationSize(void) const
```

Parameters

Return Value

Minimum allocation size.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetDefaultAlignment Function

Return the default alignment allowed by the heap. This is set by the uDefaultAlignment flag of sHeapDetails when creating the heap .

Syntax

```
jrs_u32 cHeap::GetDefaultAlignment(void) const
```

Parameters

Return Value

Size of the default alignment.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetName Function

Returns the name of the heap created when calling CreateHeap.

Syntax

```
const jrs_i8 *cHeap::GetName(void) const
```

Parameters

Return Value

NULL terminated string of the heap name.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetSize Function

Returns the name of the heap created when calling CreateHeap.

Syntax

```
jrs_sizet cHeap::GetSize(jrs_bool bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE GetSize returns the total size of all linked Heaps. FALSE to return size of this Heap. Default TRUE.

Return Value

Size in bytes of the heap or linked heaps.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetAddress Function

Returns the start address of the heap. For managed heaps this will be one created by Elephant or it will be the user requested one.

Syntax

```
void *cHeap::GetAddress(void) const
```

Parameters

Return Value

Memory address the heap starts at.

Remarks

This function is very fast and can be used real time.

cHeap::GetAddressEnd Function

Returns the end address of the heap. The end address minus the start address may not add up to the Heap size, especially in resizable mode.

Syntax

```
void *cHeap::GetAddressEnd(void) const
```

Parameters

Return Value

Memory address the heap ends at.

Remarks

This function is very fast and can be used real time.

cHeap::GetMemoryUsed Function

Returns the amount of memory used in allocations of the heap. This does not include allocation overhead but does include free block fragmentation. Just because a heap may have the memory free does not mean it is one contiguous block of memory so allocating this amount may fail.

Syntax

```
jrs_sizet cHeap::GetMemoryUsed(jrs_bool bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE returns the memory used of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Size of memory allocated in bytes.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetNumberOfAllocations Function

Returns the total number of active allocations in the heap. Multiply this value with cMemoryManager::SizeofAllocatedBlock to get the total overhead.

Syntax

```
jrs_u32 cHeap::GetNumberOfAllocations(jrs_bool bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE returns the number of allocations of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Total number of allocations in the heap.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetMemoryUsedMaximum Function

Returns the maximum amount of allocated memory the heap has had to manage. The actual value may be the same or lower but never higher than the current. Use ResetHeapStatistics set this to the current.

Syntax

```
jrs_sizet cHeap::GetMemoryUsedMaximum(jrs_bool bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE returns the maximum high use memory of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Total memory allocated in bytes at its peak.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::GetNumberOfAllocationsMaximum Function

Returns the maximum number of allocations the heap has dealt with. The actual value may be the same or lower but never higher than the current. Use ResetHeapStatistics to set this to the current.

Syntax

```
jrs_u32 cHeap::GetNumberOfAllocationsMaximum(jrs_bool bIncludeLinked)
const
```

Parameters

bIncludeLinked – If TRUE returns the maximum high number of allocations of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Number of allocations managed by this heap at its peak.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::IsLocked Function

Returns if the heap is locked and thus prevented from allocating any more memory.

Syntax

```
jrs_bool cHeap::IsLocked(void) const
```

Parameters

Return Value

TRUE if locked.
FALSE otherwise.

Remarks

cHeap::IsNullFreeEnabled Function

Returns if the heap is allowed to free NULL values passed to it. Set bAllowNullFree of sHeapDetails when creating or use SetNullFreeEnable.

Syntax

```
jrs_bool cHeap::IsNullFreeEnabled(void) const
```

Parameters

Return Value

TRUE if NULL free memory is enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::IsZeroAllocationEnabled Function

Returns if the heap is allowed to allocate memory of 0 size. Set bAllowNullFree of sHeapDetails when creating or use SetZeroAllocationEnable. Standard system malloc allows for 0 size memory and sometimes their may be a requirement for this. Note that Elephant will allocate the minimum sized memory and it must still be freed.

Syntax

```
jrs_bool cHeap::IsZeroAllocationEnabled(void) const
```

Parameters

Return Value

TRUE if 0 size memory allocation is allowed.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::EnableLock Function

Enables locking and unlocking of the heap to prevent or allow dynamic allocation.

Syntax

```
void cHeap::EnableLock(jrs_bool bEnableLock)
```

Parameters

bEnableLock - TRUE to lock the heap. FALSE otherwise.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::EnableNullFree Function

Enables freeing of NULL pointers. Set bAllowNullFree of sHeapDetails when creating the heap.

Syntax

```
void cHeap::EnableNullFree(jrs_bool bEnableNullFree)
```

Parameters

bEnableNullFree - TRUE to enable freeing of NULL memory pointers. FALSE otherwise.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::EnableZeroAllocation Function

Enables allocation of 0 size memory allocations. Sometimes it may be needed to allocate 0 byte allocations and emulate malloc. Set bAllowNullFree of sHeapDetails when creating the heap.

Syntax

```
void cHeap::EnableZeroAllocation(jrs_bool bEnableZeroAllocation)
```

Parameters

bEnableZeroAllocation - TRUE to enable zero size allocations. FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::EnableSentinelChecking Function

Enables sentinel checking of the heap. This is the quick, basic form of checking and will enable the current block of memory checked when an allocation or free is performed. Must be using the S or NACS library.

Syntax

```
void cHeap::EnableSentinelChecking(jrs_bool bEnable)
```

Parameters

bEnable - TRUE to enable quick sentinel checking. FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap:: SetCallstackDepth Function

Sets the callstack depth to start storing its addresses from N from the actual internal Elephant allocation.

By default Elephant reports a callstack from its main allocation function. However sometimes the top level callstack isn't high enough to capture exactly where the call came from. This is typical in middleware solutions where most of the callstacks will be identical internally and you cannot see where in your code the allocation originated.

The default value is 2.

Syntax

```
void cHeap::SetCallstackDepth(jrs_u32 uDepth)
```

Parameters

uDepth – The starting depth of the callstack.

Return Value

Remarks

Should be called only when no operations to this heap are being made.

cHeap::IsSentinelCheckingEnabled Function

Checks if the heap has sentinel checking enabled or not.

Syntax

```
jrs_bool cHeap::IsSentinelCheckingEnabled(void) const
```

Parameters

Return Value

TRUE if sentinel checking is enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::EnableExhaustiveSentinelChecking Function

Enables full error checking of the heap every allocation or free. This is time consuming and you must be using the S or NACS lib.

Syntax

```
void cHeap::EnableExhaustiveSentinelChecking(jrs_bool bEnable)
```

Parameters

bEnable - TRUE to enable. FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::IsExhaustiveSentinelCheckingEnabled Function

Checks if exhaustive sentinel checking is enabled or not.

Syntax

```
jrs_bool cHeap::IsExhaustiveSentinelCheckingEnabled(void) const
```

Parameters

TRUE if enabled.
FALSE otherwise.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::IsLoggingEnabled Function

Returns if continuous logging for this heap is enabled or not. By default it is enabled. Using this can remove results from heaps you do not and improve performance of coninuous logging.

Syntax

```
jrs_bool cHeap::IsLoggingEnabled(void) const
```

Parameters

Return Value

TRUE if enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::EnableLogging Function

Enables or disables continuous logging for the heap.

Syntax

```
void cHeap::EnableLogging(jrs_bool bEnable)
```

Parameters

bEnable - TRUE to enable logging or FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeap::CheckForErrors Function

Checks all the allocations in the heap to see if there are any errors and buffer overruns. This function can be time consuming but will do a thorough check and warn on the first error. When an error is found there is a large chance that a buffer overrun may cause problems with the consistency of the memory further on so it is recommended you check the first error thoroughly.

Syntax

```
void cHeap::CheckForErrors(void)
```

Parameters

Return Value

Remarks

Thread safe during cHeap operations but not from cMemoryManager functions. This function can be time consuming.

cHeap::DebugTrapOnFreeNumber Function

Sets a flag to check on the heap if the unique free number matches the one passed to this function. This functionality can be used to track when certain allocations are about to be freed in order to help debug. Will trap in the user error callback.

Syntax

```
void cHeap::DebugTrapOnFreeNumber(jrs_u32 uFreeNumber)
```

Parameters

uFreeNumber - Unique free number of the allocation to catch.

Return Value

Remarks

Partially thread safe. Operation is safe as long as no Heap creation or destruction is occurring. Very fast and suitable for runtime updating.

.

cHeap::DebugTrapOnFreeAddress Function

Sets a flag to check on the heap if the address matches the allocation being freed. This functionality can be used to track when certain allocations are about to be freed in order to help debug. Will trap in the user error callback.

Syntax

```
void cHeap::DebugTrapOnFreeAddress(void *pAddress)
```

Parameters

pAddress - Address memory allocation to catch.

Return Value

Remarks

Partially thread safe. Operation is safe as long as no Heap creation or destruction is occurring. Very fast and suitable for runtime updating.

cHeap::DebugTrapOnAllocatedUniqueNumber Function

Sets a flag to check on the heap if the unique allocation number matches the one passed to this function. This functionality can be used to track when certain allocations are being allocated in order to help debug. Will trap in the user error callback.

Syntax

```
void cHeap::DebugTrapOnAllocatedUniqueNumber(jrs_u32 uUniqueNumber)
```

Parameters

Return Value

uUniqueNumber - Unique number to test allocations against.

Remarks

Partially thread safe. Operation is safe as long as no Heap creation or destruction is occurring. Very fast and suitable for runtime updating.

cHeap::DebugTrapOnAllocatedAddress Function

Sets a flag to check on the heap if the allocation address matches the one passed to this function. This functionality can be used to track when certain allocations are about to be allocated in order to help debug. Will trap in the user error callback.

Syntax

```
void cHeap::DebugTrapOnAllocatedAddress(void *pAddress)
```

Parameters

pAddress - Memory address to compare new allocations against.

Return Value

Remarks

Partially thread safe. Operation is safe as long as no Heap creation or destruction is occurring. Very fast and suitable for runtime updating.

cHeap::ReportAll Function

Does a report on the heap (statistics and allocations in memory order) to the user TTY callback. If you want to report these to a file full path and file name to the function. The generated file is an Overview file for use in Goldfish. The file is in a compatible CSV format for loading into a spread sheet for self analysis. When connected to a debugger TTY output may cause this function to take a lot longer than just writing to a log file can take. Time is proportionate to the number of allocations.

Syntax

```
void cHeap::ReportAll(const jrs_i8 *pLogToFile, jrs_bool
includeFreeBlocks, jrs_bool displayCallStack)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

includeFreeBlocks – Default FALSE. When logging to the debugger output window include/exclude free blocks.

displayCallStack – Default FALSE. Output will display the callstack. On supported platforms it will try and resolve the function names to the actual value rather than display memory addresses.

Return Value

Remarks

Thread safe. Can be time consuming.

cHeap::ReportStatistics Function

Reports basic statistics about the heap to the user TTY callback. Use this to get quick information on when need that is more detailed that simple heap memory real time calls.

Syntax

```
void cHeap::ReportStatistics(void)
```

Parameters

Return Value

Remarks

Thread safe. Can be time consuming.

cHeap::ReportAllocationsMemoryOrder Function

Reports on all the allocations in the heap. This function is like ReportAll but doesn't output the statistics. Calling this function with a valid file name will create an Overview file for use in Goldfish.

Syntax

```
void cHeap::ReportAllocationsMemoryOrder(const jrs_i8 *pLogToFile,
jrs_bool includeFreeBlocks, jrs_bool displayCallStack)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

includeFreeBlocks – Default FALSE. When logging to the debugger output window include/exclude free blocks.

displayCallStack – Default FALSE. Output will display the callstack. On supported platforms it will try and resolve the function names to the actual value rather than display memory addresses.

Return Value

Remarks

Thread safe, can be time consuming.

cHeap::ResetStatistics Function

Resets the heap statistics to the current values.

Syntax

```
void cHeap::ResetStatistics(void)
```

Parameters

Return Value

TRUE if logging is currently enabled.
FALSE otherwise.

Remarks

Not thread safe but actually safe to call as long as no heap creation or destruction functions are called. At most, results may be out by one allocation/free if memory operations to this heap are called on other threads at the same time.

cHeapNonIntrusive::AllocateMemory Function

Main call to allocate memory directly to the heap. This function is called by cMemoryManager::Malloc. Call this to avoid the extra overhead of malloc or to redirect to a specific heap. This function is the main allocation functionto use. Flags are set by the user. It can be one of JRSMEMORYFLAG_xxx or any user specified flags > JRSMEMORYFLAG_RESERVED3 but smaller than or equal to 15, values greater than 15 will be lost and operation of AllocateMemory is undefined. Input text is limited to 32 chars including terminator. Strings longer than this will only store the last 31 chars. Alignment must be a power of two however a 0 will default to the default allocation size set when the heap was created.

Syntax

```
void *cHeapNonIntrusive::AllocateMemory(jrs_sizet uSize, jrs_u32
uAlignment, jrs_u32 uFlag, const jrs_i8 *pName, const jrs_u32
uExternalId)
```

Parameters

uSize - Size in bytes. Minimum size will be 16bytes unless the heap settings have set a larger minimum size.
uAlignment - Default alignment is 16bytes unless heap settings have set a larger alignment. Any specified alignments must be a power of 2. Setting 0 will default to the minimum requested alignment of the heap.
uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. See description for more details.
pName - NULL terminating text string to associate with the allocation. May be NULL.
uExternalId – Here for compatibility with cHeap. Not used internally.

Return Value

Valid pointer to allocated memory.
NULL otherwise.

Remarks

Thread safe. This function is moderate in operation speed.

cHeapNonIntrusive::FreeMemory Function

Frees memory allocated from AllocateMemory or Malloc. If the allocation does not come from this heap FreeMemory will fail and/or warn first depending on heap settings. FreeMemory will free the memory that matches the flag of the allocation. Text is limited to 32 chars including terminator. Strings longer than this will only store the last 31 chars. In certain situations FreeMemory may free memory from a heap which has been allocated from with in other heaps. In this rare situation memory corruption may occur.

Syntax

```
void cHeapNonIntrusive::FreeMemory(void *pMemory, jrs_u32 uFlag, const
jrs_i8 *pName, const jrs_u32 uExternalId)
```

Parameters

pMemory - Valid memory pointer. A null pointer may only be passed in if bAllowNullFree is set in the heap CreateHeap details.
uFlag - One of JRSMEMORYFLAG_xxx or user defined value. Default JRSMEMORYFLAG_NONE. Must match the flag set at allocation time.
pName - NULL terminating text string to associate with the allocation. May be NULL.
uExternalId – Here for compatibility with cHeap. Not used internally.

Return Value

Remarks

Thread safe. This function has a moderate execution time.

cHeapNonIntrusive::IsAllocatedFromThisHeap Function

Checks if the memory pointer was allocated from this heap by checking the heaps memory range. May get confused if memory is located within other heaps.

Syntax

```
jrs_bool cHeap::IsAllocatedFromThisHeap(void *pMemory) const
```

Parameters

pMemory - Valid memory pointer to a block of memory.

Return Value

TRUE if memory belongs to heap.
FALSE otherwise.

Remarks

Thread safe as long as heap is not removed during operation. This function is fast and can be used real time.

cHeapNonIntrusive::GetSizeOfLargestFragment Function

Returns the size largest free fragment of memory in the cheap

Syntax

```
jrs_sizet cHeapNonIntrusive:: GetSizeOfLargestFragment(void)
```

Parameters

None

Return Value

Size of the largest largest

Remarks

Thread safe while heap is valid.

cHeapNonIntrusive:: IsOutOfMemoryReturnEnabled Function

Returns the size largest free fragment of memory in the cheap

Syntax

```
jrs_bool cHeapNonIntrusive:: IsOutOfMemoryReturnEnabled (void)
```

Parameters

None

Return Value

True if Out of Memory return flag is enabled. False otherwise.

Remarks

Thread safe while heap is valid.

cHeapNonIntrusive:: GetTotalAllocations Function

Returns the total number of active allocations in the heap.

Syntax

```
jrs_u32 cHeapNonIntrusive:: GetTotalAllocations(void)
```

Parameters

None

Return Value

The total number of active allocations in the heap.

Remarks

Thread safe while heap is valid.

cHeapNonIntrusive:: GetMaxAllocations Function

Returns the size largest free fragment of memory in the cheap

Syntax

```
jrs_ u32 cHeapNonIntrusive:: GetMaxAllocations (void)
```

Parameters

None

Return Value

The total number of active allocations in the heap

Remarks

Thread safe while heap is valid.

cHeapNonIntrusive:: GetTotalFreeMemoryFunction

Returns the total free memory in the heap.

Syntax

```
jrs_sizet cHeapNonIntrusive:: GetTotalFreeMemory(void)
```

Parameters

None

Return Value

Total amount of free memory.

Remarks

Thread safe while heap is valid.

cHeapNonIntrusive:: GetResizableSize Function

Returns the minimum size the heap will resize in resizable mode. This property is set in sHeapDetail structure on Heap creation.

Syntax

```
jrs_sizet cHeapNonIntrusive::GetResizableSize(void)
```

Parameters

None

Return Value

The size in bytes of the minimum resize size.

Remarks

Thread safe while heap is valid.

cHeapNonIntrusive:: GetNumberOfLinks Function

Returns the total number of active links with in a heap. During Resizable mode hard links may be inserted into the heaps to join memory locations up. This returns how many fragments the heap has been brocken into internally.

Syntax

```
jrs_u32 cHeapNonIntrusive:: GetNumberOfLinks(void)
```

Parameters

None

Return Value

The total number of active links in a heap.

Remarks

Not thread safe.

cHeapNonIntrusive::AreErrorsEnabled Function

Returns if error checking is enabled or not for the heap. This is determined by the bEnableErrors flag of sHeapDetails when creating the heap.

Syntax

```
jrs_bool cHeapNonIntrusive::AreErrorsEnabled(void) const
```

Parameters

Return Value

TRUE if errors are enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::AreErrorsWarningsOnly Function

Returns if all errors have been demoted to just warnings for the heap. This is determined by the bErrorsAsWarnings flag of sHeapDetails when creating the heap. If error checking enabled this will be ignored internally.

Syntax

```
jrs_bool cHeapNonIntrusive::AreErrorsWarningsOnly(void) const
```

Parameters

Return Value

TRUE if errors are demoted to warnings.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetMaxAllocationSize Function

Return the maximum allocation size allowed by the heap. This is set by the uMaxAllocationSize flag of sHeapDetails when creating the heap.

Syntax

```
jrs_sizet cHeapNonIntrusive::GetMaxAllocationSize(void) const
```

Parameters

Return Value

Maximum allowed allocation size.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetMinAllocationSize Function

Return the minimum allocation size allowed by the heap. This is set by the uMinAllocationSize flag of sHeapDetails when creating the heap. The heap will automatically resize the minimum size to match this which is different functionality to the maximum size which will fail. This is because some systems may require a minimum size internally for some memory items due to page or boundary sizes.

Syntax

```
jrs_sizet cHeapNonIntrusive::GetMinAllocationSize(void) const
```

Parameters

Return Value

Minimum allocation size.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetDefaultAlignment Function

Return the default alignment allowed by the heap. This is set by the uDefaultAlignment flag of sHeapDetails when creating the heap .

Syntax

```
jrs_u32 cHeapNonIntrusive::GetDefaultAlignment(void) const
```

Parameters

Return Value

Size of the default alignment.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetName Function

Returns the name of the heap created when calling CreateHeap.

Syntax

```
const jrs_i8 *cHeapNonIntrusive::GetName(void) const
```

Parameters

Return Value

NULL terminated string of the heap name.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetSize Function

Returns the name of the heap created when calling CreateHeap.

Syntax

```
jrs_sizet cHeapNonIntrusive::GetSize(jrs_bool bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE GetSize returns the total size of all linked Heaps. FALSE to return size of this Heap. Default TRUE.

Return Value

Size in bytes of the heap or linked heaps.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetAddress Function

Returns the start address of the heap. For managed heaps this will be one created by Elephant or it will be the user requested one.

Syntax

```
void *cHeapNonIntrusive::GetAddress(void) const
```

Parameters

Return Value

Memory address the heap starts at.

Remarks

This function is very fast and can be used real time.

cHeapNonIntrusive::GetAddressEnd Function

Returns the end address of the heap. The end address minus the start address may not add up to the Heap size, especially in resizable mode.

Syntax

```
void *cHeapNonIntrusive::GetAddressEnd(void) const
```

Parameters

Return Value

Memory address the heap ends at.

Remarks

This function is very fast and can be used real time.

cHeapNonIntrusive::GetMemoryUsed Function

Returns the amount of memory used in allocations of the heap. This does not include allocation overhead but does include free block fragmentation. Just because a heap may have the memory free does not mean it is one contiguous block of memory so allocating this amount may fail.

Syntax

```
jrs_sizet cHeapNonIntrusive::GetMemoryUsed(jrs_bool bIncludeLinked)
const
```

Parameters

bIncludeLinked – If TRUE returns the memory used of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Size of memory allocated in bytes.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetNumberOfAllocations Function

Returns the total number of active allocations in the heap. Multiply this value with cMemoryManager::SizeofAllocatedBlock to get the total overhead.

Syntax

```
jrs_u32 cHeapNonIntrusive::GetNumberOfAllocations(jrs_bool
bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE returns the number of allocations of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Total number of allocations in the heap.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetMemoryUsedMaximum Function

Returns the maximum amount of allocated memory the heap has had to manage. The actual value may be the same or lower but never higher than the current. Use ResetHeapStatistics set this to the current.

Syntax

```
jrs_sizet cHeapNonIntrusive::GetMemoryUsedMaximum(jrs_bool
bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE returns the maximum high use memory of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Total memory allocated in bytes at its peak.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::GetNumberOfAllocationsMaximum Function

Returns the maximum number of allocations the heap has dealt with. The actual value may be the same or lower but never higher than the current. Use ResetHeapStatistics to set this to the current.

Syntax

```
jrs_u32 cHeapNonIntrusive::GetNumberOfAllocationsMaximum(jrs_bool
bIncludeLinked) const
```

Parameters

bIncludeLinked – If TRUE returns the maximum high number of allocations of all linked Heaps. FALSE to return just this Heap. Default TRUE.

Return Value

Number of allocations managed by this heap at its peak.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::IsLocked Function

Returns if the heap is locked and thus prevented from allocating any more memory.

Syntax

```
jrs_bool cHeapNonIntrusive::IsLocked(void) const
```

Parameters

Return Value

TRUE if locked.
FALSE otherwise.

Remarks

cHeapNonIntrusive::IsNullFreeEnabled Function

Returns if the heap is allowed to free NULL values passed to it. Set bAllowNullFree of sHeapDetails when creating or use SetNullFreeEnable.

Syntax

```
jrs_bool cHeapNonIntrusive::IsNullFreeEnabled(void) const
```

Parameters

Return Value

TRUE if NULL free memory is enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::IsZeroAllocationEnabled Function

Returns if the heap is allowed to allocate memory of 0 size. Set bAllowNullFree of sHeapDetails when creating or use SetZeroAllocationEnable. Standard system malloc allows for 0 size memory and sometimes their may be a requirement for this. Note that Elephant will allocate the minimum sized memory and it must still be freed.

Syntax

```
jrs_bool cHeapNonIntrusive::IsZeroAllocationEnabled(void) const
```

Parameters

Return Value

TRUE if 0 size memory allocation is allowed.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::EnableLock Function

Enables locking and unlocking of the heap to prevent or allow dynamic allocation.

Syntax

```
void cHeapNonIntrusive::EnableLock(jrs_bool bEnableLock)
```

Parameters

bEnableLock - TRUE to lock the heap. FALSE otherwise.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::EnableNullFree Function

Enables freeing of NULL pointers. Set bAllowNullFree of sHeapDetails when creating the heap.

Syntax

```
void cHeapNonIntrusive::EnableNullFree(jrs_bool bEnableNullFree)
```

Parameters

bEnableNullFree - TRUE to enable freeing of NULL memory pointers. FALSE otherwise.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::EnableZeroAllocation Function

Enables allocation of 0 size memory allocations. Sometimes it may be needed to allocate 0 byte allocations and emulate malloc. Set bAllowNullFree of sHeapDetails when creating the heap.

Syntax

```
void cHeapNonIntrusive::EnableZeroAllocation(jrs_bool
bEnableZeroAllocation)
```

Parameters

bEnableZeroAllocation - TRUE to enable zero size allocations. FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::EnableSentinelChecking Function

Enables sentinel checking of the heap. This is the quick, basic form of checking and will enable the current block of memory checked when an allocation or free is performed. Must be using the S or NACS library.

Syntax

```
void cHeapNonIntrusive::EnableSentinelChecking(jrs_bool bEnable)
```

Parameters

bEnable - TRUE to enable quick sentinel checking. FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::SetCallstackDepth Function

Sets the callstack depth to start storing its addresses from N from the actual internal Elephant allocation.

By default Elephant reports a callstack from its main allocation function. However sometimes the top level callstack isn't high enough to capture exactly where the call came from. This is typical in middleware solutions where most of the callstacks will be identical internally and you cannot see where in your code the allocation originated.

The default value is 2.

Syntax

```
void cHeapNonIntrusive::SetCallstackDepth(jrs_u32 uDepth)
```

Parameters

uDepth – The starting depth of the callstack.

Return Value

Remarks

Should be called only when no operations to this heap are being made.

cHeapNonIntrusive::IsLoggingEnabled Function

Returns if continuous logging for this heap is enabled or not. By default it is enabled. Using this can remove results from heaps you do not and improve performance of coninuous logging.

Syntax

```
jrs_bool cHeapNonIntrusive::IsLoggingEnabled(void) const
```

Parameters

Return Value

TRUE if enabled.
FALSE otherwise.

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::EnableLogging Function

Enables or disables continuous logging for the heap.

Syntax

```
void cHeapNonIntrusive::EnableLogging(jrs_bool bEnable)
```

Parameters

bEnable - TRUE to enable logging or FALSE to disable.

Return Value

Remarks

Thread safe. This function is very fast and can be used real time.

cHeapNonIntrusive::CheckForErrors Function

Checks all the allocations in the heap to see if there are any errors and buffer overruns. This function can be time consuming but will do a thorough check and warn on the first error. When an error is found there is a large chance that a buffer overrun may cause problems with the consistency of the memory further on so it is recommended you check the first error thoroughly.

Syntax

```
void cHeapNonIntrusive::CheckForErrors(void)
```

Parameters

Return Value

Remarks

Thread safe during cHeap operations but not from cMemoryManager functions. This function can be time consuming.

cHeapNonIntrusive::ReportAll Function

Does a report on the heap (statistics and allocations in memory order) to the user TTY callback. If you want to report these to a file full path and file name to the function. The generated file is an Overview file for use in Goldfish. The file is in a compatible CSV format for loading into a spread sheet for self analysis. When connected to a debugger TTY output may cause this function to take a lot longer than just writing to a log file can take. Time is proportionate to the number of allocations.

Syntax

```
void cHeapNonIntrusive::ReportAll(const jrs_i8 *pLogToFile, jrs_bool
includeFreeBlocks, jrs_bool displayCallStack)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

includeFreeBlocks – Default FALSE. When logging to the debugger output window include/exclude free blocks.

displayCallStack – Default FALSE. Output will display the callstack. On supported platforms it will try and resolve the function names to the actual value rather than display memory addresses.

Return Value

Remarks

Thread safe. Can be time consuming.

cHeapNonIntrusive::ReportStatistics Function

Reports basic statistics about the heap to the user TTY callback. Use this to get quick information on when need that is more detailed that simple heap memory real time calls.

Syntax

```
void cHeapNonIntrusive::ReportStatistics(void)
```

Parameters

Return Value

Remarks

Thread safe. Can be time consuming.

cHeapNonIntrusive::ReportAllocationsMemoryOrder Function

Reports on all the allocations in the heap. This function is like ReportAll but doesn't output the statistics. Calling this function with a valid file name will create an Overview file for use in Goldfish.

Syntax

```
void cHeapNonIntrusive::ReportAllocationsMemoryOrder(const jrs_i8
*pLogToFile, jrs_bool includeFreeBlocks, jrs_bool displayCallStack)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

includeFreeBlocks – Default FALSE. When logging to the debugger output window include/exclude free blocks.

displayCallStack – Default FALSE. Output will display the callstack. On supported platforms it will try and resolve the function names to the actual value rather than display memory addresses.

Return Value

Remarks

Thread safe, can be time consuming.

cHeapNonIntrusive::ResetStatistics Function

Resets the heap statistics to the current values.

Syntax

```
void cHeapNonIntrusive::ResetStatistics(void)
```

Parameters

Return Value

TRUE if logging is currently enabled.
FALSE otherwise.

Remarks

Not thread safe but actually safe to call as long as no heap creation or destruction functions are called. At most, results may be out by one allocation/free if memory operations to this heap are called on other threads at the same time.

cPoolBase::GetHeap Function

Returns the cHeap that the Pool is attached too. This will always be a valid pointer.

Syntax

```
cHeap *cPoolBase::GetHeap(void) const
```

Parameters

None

Return Value

Heap the pool is attached too.

Remarks

Not thread safe regardless of flag.

cPoolBase::GetName Function

Returns the name of the Pool.

Syntax

```
const jrs_i8 *cPoolBase::GetName(void) const
```

Parameters

None

Return Value

Returns the Pool name.

Remarks

Not thread safe regardless of flag.

cPoolBase::IsLocked Function

Returns if the pool will allow allocations or not.

Syntax

Parameters

None

Return Value

TRUE if the pool is locked. FALSE if allocations can occur.

Remarks

Not thread safe regardless of flag.

cPoolBase::EnableLock Function

Enables locking and unlocking of the pool to prevent or allow dynamic allocation.

Syntax

```
void cPoolBase::EnableLock(jrs_bool bEnableLock)
```

Parameters

bEnableLock - TRUE to lock the heap. FALSE otherwise.

Return Value

Nothing

Remarks

Not thread safe regardless of flag.

cPoolBase::AreErrorsEnabled Function

Returns if error checking is enabled or not for the pool. This is determined by the bEnableErrors flag of sPoolDetails when creating the heap.

Syntax

```
jrs_bool cPoolBase::AreErrorsEnabled(void) const
```

Parameters

None

Return Value

TRUE if errors are enabled. FALSE otherwise.

Remarks

Not thread safe regardless of flag.

cPoolBase::AreErrorsWarningsOnly Function

Returns if all errors have been demoted to just warnings for the pool. This is determined by the bErrorsAsWarnings flag of sPoolDetails when creating the heap. If error checking is enabled this will be ignored internally.

Syntax

```
jrs_bool cPoolBase::AreErrorsWarningsOnly(void) const
```

Parameters

None

Return Value

TRUE if errors are demoted to warnings. FALSE otherwise.

Remarks

Not thread safe regardless of flag.

cPoolBase::IsAllocatedFromThisPool Function

Checks if the memory allocated was allocated from within this pool.

Syntax

```
jrs_bool cPoolBase::IsAllocatedFromThisPool(void *pMemory) const
```

Parameters

None

Return Value

TRUE if allocated by the pool. FALSE otherwise.

Remarks

Not thread safe regardless of flag.

cPoolBase::ReportAll Function

Does a report on the pool (statistics and allocations in memory order) to the user TTY callback. If you want to report these to a file include full path and file name to the function.

Syntax

```
void cPoolBase::ReportAll(const jrs_i8 *pLogToFile)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.

Return Value

Nothing

Remarks

Thread safe if the Pool if flag is set at creation time.

cPoolBase::ReportStatistics Function

Reports basic statistics about the pool to the user TTY callback. Use this to get quick information on when need that is more detailed that simple heap memory real time calls.

Syntax

```
void cPoolBase::ReportStatistics(void)
```

Parameters

None

Return Value

Nothing

Remarks

Thread safe if the Pool if flag is set at creation time.

cPoolBase::ReportAllocationsMemoryOrder Function

Reports on all the allocations in the pool. This function is like ReportAll but doesn't output the statistics. Calling this function with a valid file name will create an Overview file for use in Goldfish.

Logging each allocation to file can be extremely slow. This is because it is not possible to tell if that allocation is free or not without traversing the whole list. Goldfish doesn't need to know about this to function correctly but for debugging it may be needed.

Syntax

```
void cPoolBase::ReportAllocationsMemoryOrder(const jrs_i8 *pLogToFile,
jrs_bool bLogEachAllocation)
```

Parameters

pLogToFile - Full path and file name null terminated string. NULL if you do not wish to generate a file.
bLogEachAllocation - TRUE to log each allocation. FALSE by default.

Return Value

Nothing

Remarks

Thread safe if the Pool if flag is set at creation time.

cPool::AllocateMemory Function (and cPoolNonIntrusive)

Allocates a block of memory from the pool. This will be the size specified at creation time unless alignment or various debugging aids boost this up. Minimum size is the size of the jrs_sizet which changes depending on 32 or 64 bit modes of Elephant.

If an overrun Heap is specified at creation time and the Pool is full, allocation will be redirected to the Heap. It is recommended you still use cPool::FreeMemory to free any overrun allocations.

Syntax

```
void *cPool::AllocateMemory(void)
```

Parameters

None

Return Value

Valid memory address. NULL otherwise

Remarks

Thread safe if the Pool if flag is set at creation time.

cPool::AllocateMemory Function (and cPoolNonIntrusive)

Allocates a block of memory from the pool. This will be the size specified at creation time unless alignment or various debugging aids boost this up. Minimum size is the size of the jrs_sizet which changes depending on 32 or 64 bit modes of Elephant.

If an overrun Heap is specified at creation time and the Pool is full, allocation will be redirected to the Heap. It is recommended you still use cPool::FreeMemory to free any overrun allocations.

Syntax

```
void *cPool::AllocateMemory(const jrs_i8 *pName)
```

Parameters

pName - Name of the allocation. 31 chars not including null terminator.

Return Value

Valid memory address. NULL otherwise.

Remarks

Thread safe if the Pool if flag is set at creation time.

cPool::FreeMemory Function (and cPoolNonIntrusive)

Frees a block of memory from the pool that was allocated with AllocateMemory. Allows a name to specified if memory tracking for the pool is enabled.

Syntax

```
void cPool::FreeMemory(void *pMemory, const jrs_i8 *pName)
```

Parameters

pMemory - Memory address previously allocated with AllocateMemory.
pName - Name of the allocation. 31 chars not including null terminator.

Return Value

Remarks

Thread safe if the Pool if flag is set at creation time.

cPoolBase:: GetNumberOfAllocations Function (and cPoolNonIntrusive)

Gets the number of allocations currently allocated.

Syntax

```
jrs_u32 cPool::GetNumberOfAllocations(void) const
```

Parameters

None

Return Value

Number of currently allocated elements in the pool.

Remarks

Not thread safe regardless of the thread flag on the Pool.

cPool::GetSize Function

Returns the size of the Pool in bytes.

Syntax

```
jrs_sizet cPool::GetSize(void) const
```

Parameters

None

Return Value

Size of the Pool in bytes.

Remarks

Not thread safe regardless of the thread flag on the Pool.