CWMemory

Overview

The purpose of the CWMemory library (for C++) is to provide memory related management, functionality and tracking features, which go above and beyond those provided by the standard C++ runtime libraries. The library should define a platform independent interface for allocating, de-allocating, setting up heaps, reference counting pointers, and querying allocation information and statistics.

Terminology

Some of the features provided by the CWMemory library are intended for use during development only to aid debugging and tracking, whilst others are intended to aid the parent application at runtime for safe/fast allocations and garbage collection. From this stage forward, the *direct* user of the CWMemory (the software engineer who chooses to add CWMemory to their software application) will be referred to as the ***developer***, whilst the final consumer of the developer's software application will be referred to as the ***end user***. The developer's software application (which includes CWMemory) will be referred to as the ***parent application***.

Key Objectives

- To provide the developer with a platform independent interface (which can be extended to support additional platforms) allowing explicit control over memory allocation and usage.
- To provide basic functionality and tracking with minimal intrusion. Few (if any) modifications to an existing code base should be required to gain basic functionality.
- To provide runtime debugging aids such as memory quotas/usage, logs, debug output and logging.
- To provide additional functionality and tools such as reference counted pointers.
- To provide maximum functionality with minimum overhead. Debug functionality should be capable of being disabled for release/retail builds, eliminating this overhead.

Key Components

CWMemory Platform Agnostic Interface – This interface provides abstracted access to the entire CWMemory namespace and contained components. Memory allocations, heap creation, pointer tracking, and runtime statistics must be accessed via this interface.

C Runtime Overrides – This component is responsible for intercepting any standard allocations (via the global scope new and delete operators) and forwarding them through the CWMemory interface. This component ensures that code which is unaware of the CWMemory library, can still be configured to use a custom heap and track allocations (at a basic level)

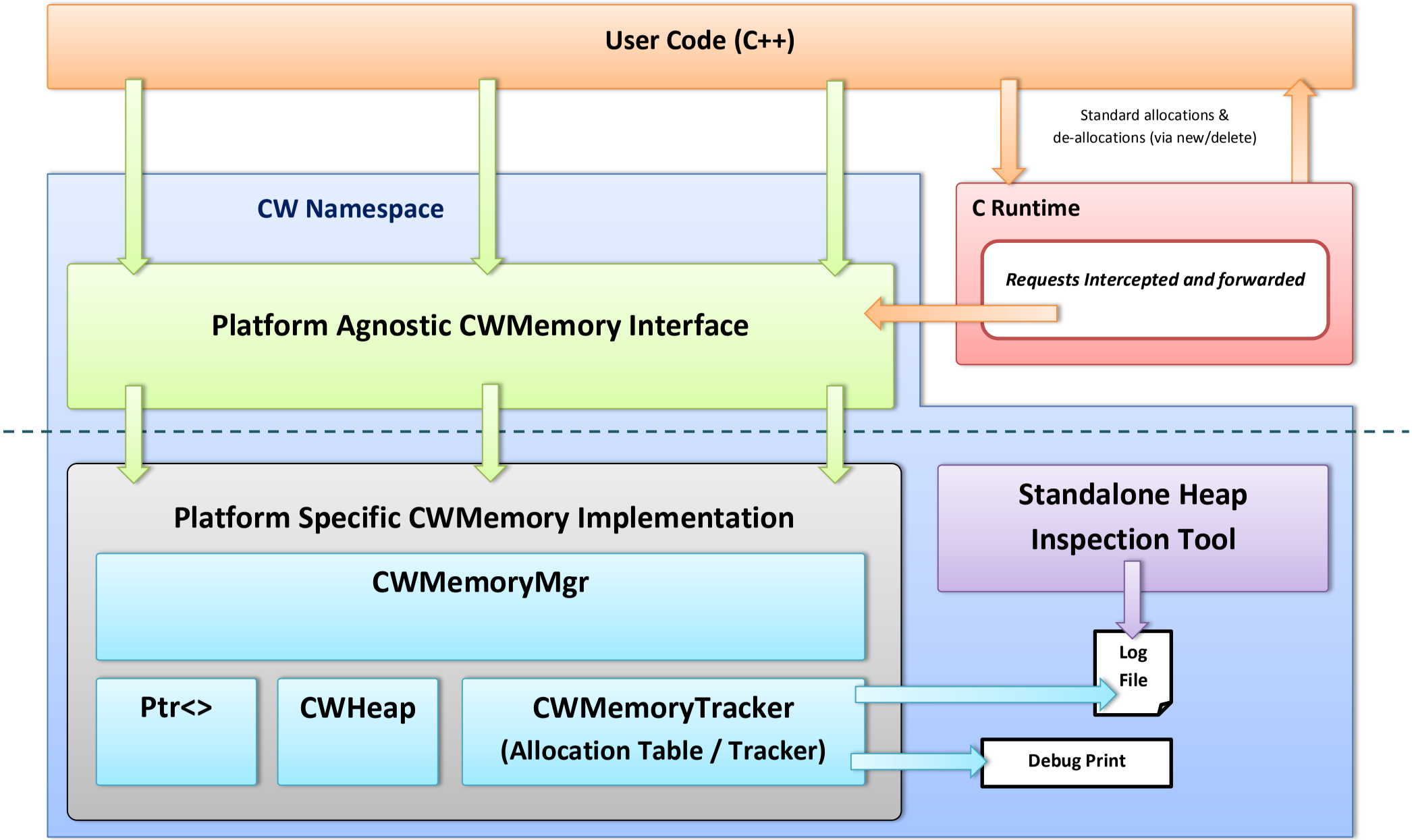
CWMemoryMgr – This component acts as a global store to register information about the platforms memory specification and quotas, and also provides access to the heap allocator and allocation table/tracker

Ptr<> - This is a smart/shared pointer provided by the CWMemory library. This should behave similarly to std::tr1::shared_ptr but additionally report reference counts and usage statistics (which can also be included in the debug print / log file output)

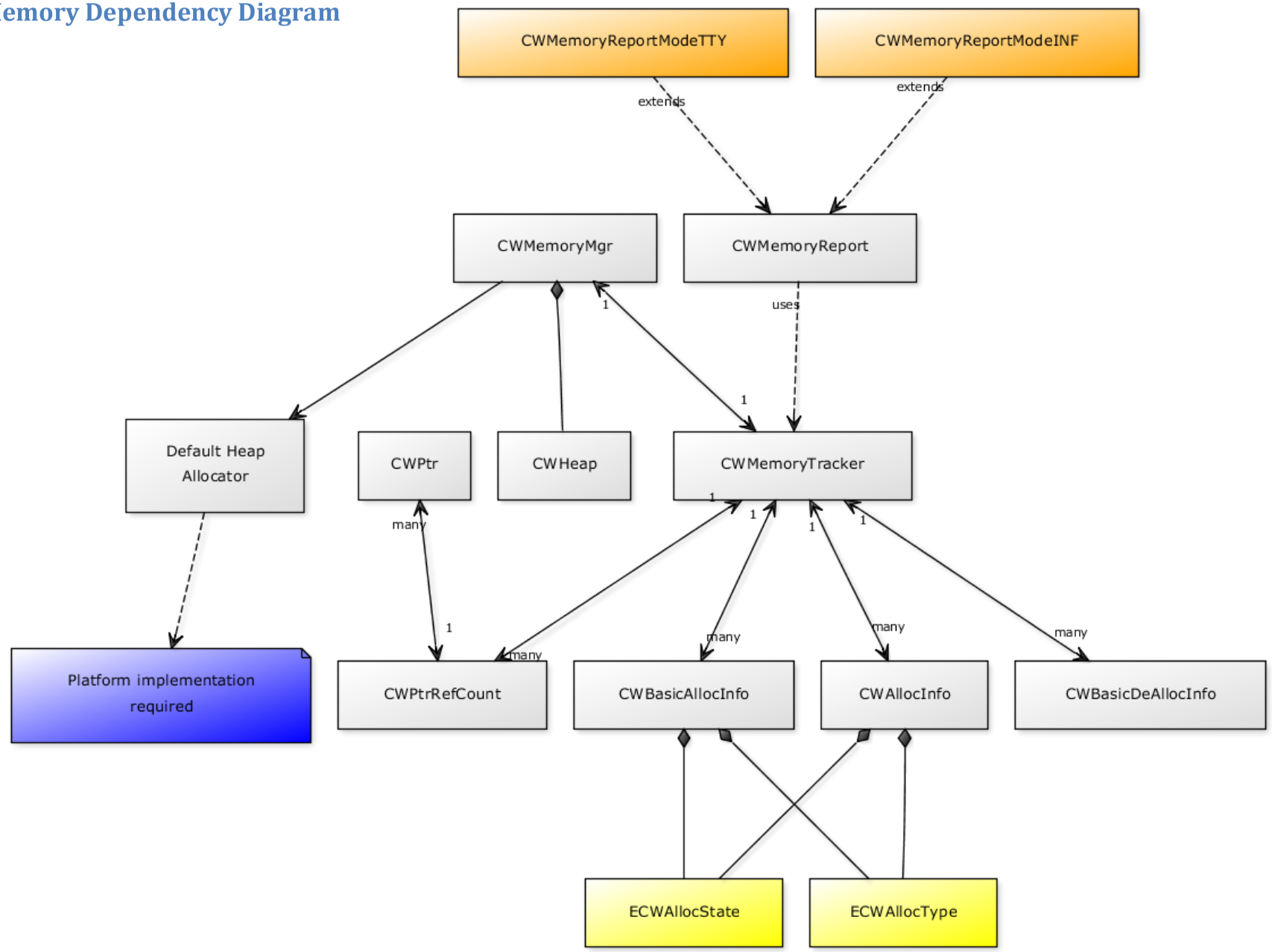
Allocation Table/Tracker – When used with the specific CWMemory allocation mechanism, this should keep track of all allocations with additional data to aid debugging (such as the file, line and call-stack where the allocation originated). When tracking standard allocations (via new/delete) only basic information will be stored.

Standalone Heap Inspection Tool – GUI tool capable of reading logs created by the CWMemory system, should be able to graphically represent memory usage using tables and graphs. Can also display the state of any active Ptr<> instances displaying ref counts and usage information.

CWMemory Components Diagram



CWMemory Dependency Diagram



C Runtime Allocation Interception

This is a feature of the library which is optional and can be enabled simply by defining the following pre-processor macro:

CW_ALLOC_INTERCEPT

When the above pre-processor flag is set, this causes the global ***new*** and ***delete*** operators to be overridden within the CWMemory library and forwarded to the CWMemoryMgr for tracking. Depending upon how CWMemoryMgr has been configured, these allocations / de-allocations may still occur on the standard heap, or may be carried out using a pre-defined CWHeap custom allocator.

Giving the developer control over this is crucial as they may only want to track a small subset of their allocations within the scope of their debugging exercise, and may wish for all regular allocations to be handled in the default manner.

Note: Standard allocations (originating from calls to global ***new*** and ***delete***) can only be tracked with basic debug data. However, allocations which are made explicitly through the CWMemory interface can be tracked much more closely, capturing additional information such as the file, line, function and call stack from which the call originated.

Cross Platform Approach

A cross platform approach will be taken towards this library, where the abstract interface to CWMemory is entirely separated from the platform specific implementation. This is particularly important as key components of the CWMemory system depend upon the underlying hardware, operating system, memory architecture (and to properly unwind the callstack, also the stack layout).

To achieve this there will be a set of **platform independent header files** which define the interface to the system. The data types used within these header files are also abstracted such that they can be redefined by each platform implementation. The implementation of this interface can be **implemented per platform** and built to create a platform specific library.

Taking this approach, the end user simply includes the platform agnostic header files, using the abstracted interface within their code. They would then **link against the platform specific library** to target the desired platform. For example, on windows, they would include the regular header files, and then link against *CWMemory_windows.lib* to include the Microsoft windows specific implementation.

Hooking into program shutdown

When enabled, the CWMemoryTracker is responsible for keeping track of any memory allocations throughout the duration of the application. These allocations may originate from three scenarios;

1. A standard allocation/de-allocation was made using ***new*** / ***delete***
2. An custom allocation/de-allocation made explicitly using ***cwnew*** / ***cwdelete***
3. A tracked memory allocation created when initialising a ***cw::Ptr*** object

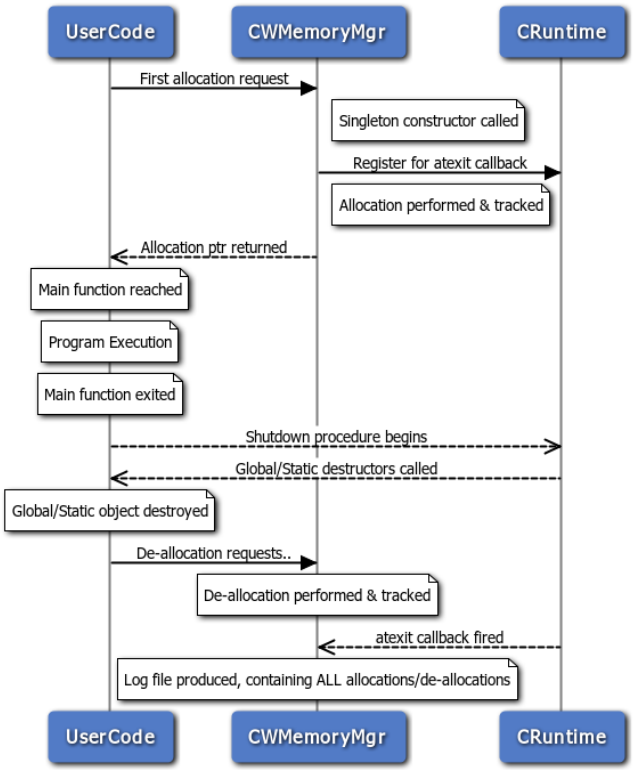
Allocations can take place at any point in the program's execution and are not limited to occurring only between the beginning and end of the programs main function. For example, allocations may occur as part of a global/static objects constructor (which executes before the main function) or as part of a global/static objects destructor (which executes after the end of the main function). Also, there is no way to determine the order of such allocations which occur outside of the main function.

This affects the CWMemory library as the internal allocation system must be ready to allocate as soon as possible (even before main) and may wish to produce logs, which include allocations which occurred *after* the main function. For this reason, the CWMemoryMgr system automatically hooks it's self into the C-runtime shutdown sequence, using the C-runtime ***atexit*** call-back**,** which provides the following guarantee;

> *"With the atexit function, you can specify an exit-processing function that executes prior to program termination. No global static objects initialized prior to the call to* ***atexit*** *are destroyed prior to execution of the exit-processing function."*

Although the wording of this statement is a little confusing, this means that as long as the atexit call-back is registered before the first allocation in the program, atexit is guaranteed to be called after all de-allocations. To achieve this, a special method *CWMemoryMgr::ScheduleForShutdown* is used to register the atexit call-back, and is called in the CWMemoryMgr singletons constructor. The CWMemoryMgr constructor is forced to be executed before the first allocation as the singleton object is must be constructed before it can perform any allocations. This process is demonstrated in ***Figure 1***.

Figure 1



Tracking Allocations (CWMemoryTracker)

Overview

CWMemoryTracker is responsible for maintaining logs and tables full of useful allocation information which can be either accessed explicitly by the developer at runtime, or exported to logs for offline inspection. All logging events and responsibilities are handled automatically by the CWMemoryMgr and CWMemoryTracker components, with such a level of abstraction that this is almost invisible to the developer. As different allocation types (basic and advanced) exist, there will be **two** allocation tables maintained by the CWMemoryTracker. The justification for this design choice is that a smaller table can be used to track basic allocations, and a larger table is required for advanced allocations. As minimising memory overhead is one of the key objectives of this library, decoupling these two tables makes sense. Storing basic allocation data amongst the advanced allocation data would be wasteful.

Basic Allocation Tracking

When *CW_ALLOC_INTERCEPT* is defined, all calls to global ***new*** and ***delete*** will be intercepted by the CWMemory library and forwarded through the CWMemoryMgr (and in turn, tracked through CWMemoryTracker). This functionality exists such that the end user can enable simple memory allocation tracking with no modification to existing code. However this does limit the amount of debug data which can be captured when tracking the allocation. As demonstrated in the next section, allocating memory explicitly via the CWMemory library is recommended to provide more in depth tracking data.

Alloc ID	Address	Size (bytes)	Call Stack	State	Ticks
0	0x0005EEDC	4	(ptr)	0	124558
1	x00062EDA	8	(ptr)	0	125889
2	x00055EAB	16	(ptr)	1	145689
3	x0005EEFF	32	(ptr)	1	188855

This is the amount of time elapsed since the initialisation of the CWMemory system. The relative time of the allocation is stored as it may be useful when debugging.

CallStack Data
Frame 0 (addr)
Frame 1 (addr)
Frame 2 (addr)
…….
Fame N (addr)

Memory Allocation Layout (Microsoft CRT Implementation)

Independent of the underlying platform, the memory layout for an allocation of ***n*** bytes will be as in the table below. Provided the allocation was performed my **malloc** on windows (which carries out some internal tracking), we can retrieve ***n*** during de-allocation using **_msize**. This allows us to validate the magic number and lookup the Allocation ID for tracking purposes. On other platforms this scheme may vary, and the size of the allocation may need to be stored at offset 0.

Offset	Description
0	User Data Size
0 + 4	User Data
	….
n + 4	Magic Number
n + 8	Allocation ID
n + 12	CRT header (~36 bytes)

Information taken from MSDN
http://msdn.microsoft.com/en-us/library/bebs9zyz.aspx

Advanced Allocation Tracking

When memory allocations in the user code are performed explicitly using the CWMemory library (via **cwnew** and **cwdelete**) more detailed tracking information can be collected, such as the file, function, and line where the allocation originated. As many allocations may originate from the same place in the user code (**one** origin **to many** allocations), a separate "Origin Table", will be used and referenced by the main allocation table.

CallStack Data

Frame 1 (addr)
Frame 2 (addr)
…….
Fame N (addr)

Alloc Table

Alloc ID	Address	Size (bytes)	Alignment	Origin	Call Stack	State	Ticks	User Flags
0	0x0005EEDC	4	4	**A** (ptr)	(ptr)	0	124558	0
1	x00062EDA	8	1	**A** (ptr)	(ptr)	0	125889	0
2	x00055EAB	16	8	**B** (ptr)	(ptr)	1	145689	0
3	x0005EEFF	32	1	**A** (ptr)	(ptr)	2	188855	0

Origin Table

Origin ID	File	Function	Line
A	Render.cpp	Render::Update	55
B	Log.cpp	LogMgr::Init	145

Singleton Pattern Access

Within the CWMemory library there are certain components (represented by C++ classes) which should have only one instance throughout the duration of the programs execution. These classes might represent components which are for internal use only, or components which the end user may interact with or even implement in order to extend the CWMemory library. An example of this is the main CWMemoryMgr class or the CWMemoryTracker, both of which only exist as only one instance. For this reason the Singleton design pattern was adopted and applied to such classes to enforce this limitation. There are different variations of the singleton design pattern, each with their own implementation details. The first singleton pattern investigated provides a class with singleton behaviour via a templatised base class, as shown in ***Figure 2***.

Figure 2

```
namespace CW
{
   template<typename T>
   class CWSingleton : public CWBase
   {
      public:
         // Access single instance of singleton object
         static T& Instance()
         {
            if(m_pInstance == 0)
               m_pInstance = new T();

            return *m_pInstance;
         }

      protected:
         // Force derived singleton object to specify a name
         inline explicit CWSingleton() { }

      private:
         static T* m_pInstance;

         // Prevent singeton object/reference from being copied/assigned
         inline explicit CWSingleton(const CWSingleton&) { }
         inline CWSingleton& operator=(const CWSingleton&)
         {
            return *this;
         }

         static void DetsroyInstance()
         {
               delete m_pInstance;
               m_pInstance = 0;
          }
   };
}

template<typename T>
typename T* CW::CWSingleton<T>::m_pInstance = 0;
```

Figure 3

```
private:
   ClassName()
   {
      //constructor code here
   }

public:
   static ClassName& Singleton()
   {
      static ClassName singleton;
      return singleton;
   }
```

The problem with using a base class to achieve singleton access has one main flaw. Most singleton patterns (even those which differ in approach) hide the constructor and copy constructors (making them ***private***) to prevent the end user creating their own instances. A base class has no way of enforcing that the constructors or the derived class are made private and forces the derived class to do this manually. For this reason alone, the base class singleton implementation was discarded as it can be considered incomplete and requires the user to make additional modifications to their class to achieve true singleton behaviour.

Instead I decided to adopt the traditional Meyer's Singleton (Conceived by Scott Meyers) in order to make classes behave as singletons. The implementation for this technique can be seen in ***Figure 3***.

The code shown in ***Figure 3*** can be added to any class to ensure that singleton behaviour is available and enforced. Manually forcing the end user (whether the end user is considered the internal maintainer of the library or the end user) to add this code is unrealistic and long winded. Instead I devised the following pre-processor macro (shown in ***Figure 4***) to add this code automatically at compile time, whilst still allowing the user to provide a custom constructor (as demonstrated in ***Figure 5***).

Figure 4

```
#define CW_OB (
#define CW_CB )

#define CW_DECLARE_SINGLETON(NAME) public: \
static EVALUATOR(NAME,&) Instance() \
{ \
      static NAME singleton; \
      return singleton; \
} \
private: \
      NAME CW_OB CW_CB
```

Figure 5

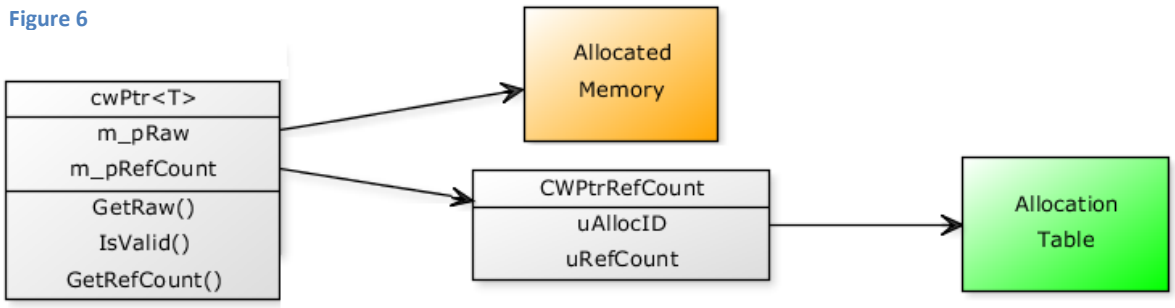
```
class CWMemoryMgr
{
   CW_DECLARE_SINGLETON(CWMemoryMgr)
   {
      m_stopWatch.Start();
      ScheduleForShutdown();
   }

   // .....

};
```

Resource Acquisition Is Instantiation (RAII)

RAII is a programming idiom vital to writing exception-safe C++ code. RAII is a technique used to "acquire" resources as they are created / instantiated by wrapping them inside a scoped object. This way, when the encapsulating object goes out of scope (naturally at the end local/block scope or when an exception is thrown) the underlying resource can be safely released to the system. The resource in question might be a window handle, database connection, or any number of system resources. Within the CWMemory library, the RAII technique will be present within the **CW::Ptr** class which will be used to wrap raw memory pointers.

When memory is allocated and assigned to a raw pointer, traditionally it is the developer's responsibility to track the usage of this pointer/memory, and determine when best to release the allocated memory back to the operating system. If several data structures within the C++ application hold reference to the allocated memory, knowing exactly when to release this memory is a difficult problem. Fortunately, when the raw pointer is wrapped inside the **CW::Ptr**, the copy constructor and destructor of the object can be used to track references by incrementing and decrementing a system-wide reference count. When the last instance of the **CW::Ptr** referencing the allocated memory goes out of scope, the destructor decrements the reference count to zero, and the memory can be safely released to the operating system. There are existing classes available to provide this functionality such as std::tr1::shared_ptr (or std::shared_ptr in C++11). The **CW::Ptr** class will be a core component of the CWMemory library, which aims to provide a similar interface to that provided by shared_ptr but with the additional benefit of the memory and reference count being tracked by the global CWMemoryTracker system. This way **CW::Ptr** instances will appear alongside regular allocations within any logs and unify the memory debugging / tracking process. **CW::Ptr** will be templated such that it can track memory allocated to any type. A class diagram for the **CW::Ptr** component is shown in ***Figure 6***.

Figure 6



Compile Time Pruning/Branching

The CWMemory library exists as a tool to aid developers when debugging and managing memory allocations and budgets, an area of key importance when developing large scale C++ applications. Most of the features and facilities provided by the CWMemory library are therefore aimed at the application developer, and not as the end user of the application. Memory management and tracking are essential during the debugging phase of an applications lifecycle, but would rarely be included in a release/retail final build. For this reason it is important to be able to minimise, and in some scenarios, strip the features and overhead of the CWMemory library when the parent application is being built for release. There are however some features the library provides (such as heap management and managed pointers) which would ideally remain within a release build of the parent application (as they are used at runtime to facilitate garbage collection and faster allocations).

Of course, to achieve this selective pruning of functionality, the application developer *could* spend time prior to the release of their application, sifting through code and removing all references and link dependencies to CWMemory. However this process is long and monotonous and likely to aggravate the developer. Instead, the aim is to provide pre-processor switches to selectively strip different features of the library at compile time, fully automating the pruning process.

Examples of this technique will be used throughout the libraries implementation; an example can be seen in ***Figure 6***, where such an approach is used to determine the construction of the CWMemoryTracker instance at compile time.

Figure 6

```
CWHeap(const CWstring& name) : CWBase("CWHeap"), m_bIsLogging(CWtrue)
{
    #ifdef CW_RELEASE_BUILD
        m_bIsLogging = CWfalse;
    #endif
}
```

User Flags

User flags are 1 byte flags which can be passed (optionally) by the user per allocation, used as a custom identifier distinguish it from any other allocations within the application. The reason for adding this mechanism is that due to the runtime performance penalty often imposed by most compilers inbuilt RTTI implementation, many game engines often disable RTTI (Run Time Type Information). Therefore it almost is impossible for the allocation tracker to determine the type of the object requesting an allocation. However, there will be few occasions when this data is useful, and when it is, the user can pass a type identifier (1 byte in size) to be associated with each allocation. These user flags are stored alongside the allocation within the allocation table maintained by CWMemoryTracker (and also added to any logs emitted by the CWMemory system).

Being able to associate meaningful identifiers with particular allocations empowers the developer when debugging. Allowing them to filter and easily spot any allocations issued with a particular user flag. Examples of user flag usage can be found in the user documentation.

User callback mechanism on bad alloc/dealloc

The CWMemory system should be capable of internally dealing with errors regarding memory allocation and de-allocations. These errors may stem from any of the following issues;

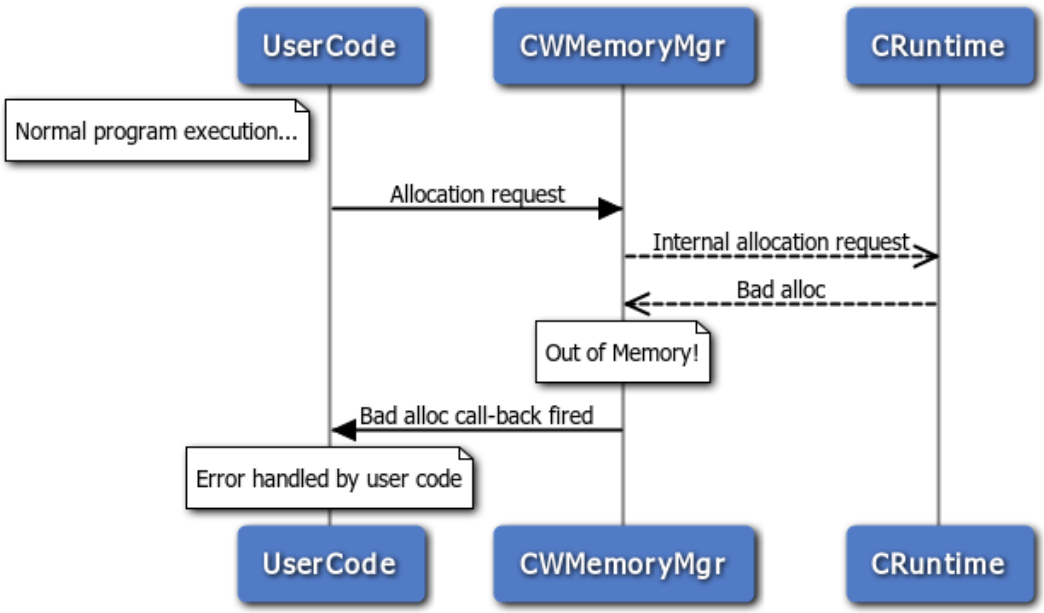
1. The operating system was unable to allocate the amount of memory requested
2. The operating system was unable to align the data as requested
3. A negative number of bytes was requested for allocation
4. An attempt was made to de-allocate memory which was never allocated or previously de-allocated
5. The memory being de-allocated is corrupted (due to overflow / bad writes)

When such critical errors do occur it is vital that they are handled immediately by the parent application. One option would be to use return values from the allocation/de-allocation functions to communicate the success of the operation. Unfortunately this is not feasible for a couple of reasons. Checking the return code of every memory allocation within an application can be tedious and is often forgotten / neglected by the developer. Secondly, methods such as new and delete do not typically return status/error codes so this might appear an un-natural approach.

Exceptions are another good method for handling errors and passing control back to the parent application. However, the overhead incurred by exceptions often makes them a bad choice, especially within games engines which typically disable SEE (structured exception handling).

As an alternative to throwing an exception, the CWMemory will allow the developer to register their own call-backs for important events and failures such as bad allocations and allocation exceptions. When such errors occur, if the parent application has registered a call-back for the relevant event, this call-back will be fired, passing with it a detailed description of the error. This allows the call-back function to safely deal with any critical errors and also allows the developer to add a breakpoint function to their call-back function in order to view the call stack as it was when the error originated. This process is demonstrated in ***Figure 7***.

Figure 7



Allocation Information

Once an allocation has been performed by the CWMemoryMgr (and a pointer has been returned), the developer will be able to use the memory pointer to request information about this allocation at any point. This functionality will be offered by the CWMemoryTracker, which when provided with a valid memory address (one which is being tracked within the system) it can look up the allocation data and return it to the developer for debugging purposes. For example, the developer may have a particular allocation for which they need to know the file, line and call stack where this occurred, or they may want to check the *state* of the allocation (whether or not the allocations is active or has been freed).

Reporting

The CWMemory library will have a built in reporting system, allowing all of the tracked allocation information and logs to be exported. This information will include the following;

- Allocation Table
  - ID
  - Size
  - timestamp,
  - state,
  - file,
  - line number
  - Call stack
- Bad allocation/de-allocation Event Data
- cw::Ptr Tracking Tables
  - ID
  - Ref count
- Statistics
  - Num allocations
  - Num deallocations
  - Num bad allocations/deallocations
  - Num cw::Ptr instances
  - Size allocated

The internal representation and organisation of this data within the system is well suited to its runtime usage/behaviour, however is not necessarily designed to be readable and comprehendible by the developer. For this reason the reporting system will communicate with a ***ReportComponent*** to delegate the task of re-interpreting/formatting the data for use by the developer, following the "Template Method Pattern" (see http://www.blackwasp.co.uk/TemplateMethod.aspx).

CWMemory will provide two ***ReportComponent*** objects by default, one which logs data to a file in a readable format, and another which writes the data in a similar format the Debug Output stream (for viewing within Visual Studio). The developer will be required to specify a ***ReportComponent*** (via template parameter) whenever a report is generated. The developer can also design their own ***ReportComponent*** for use with the report system. See the "Extending CWMemory" section of the user docs for more info.