Memo Overview

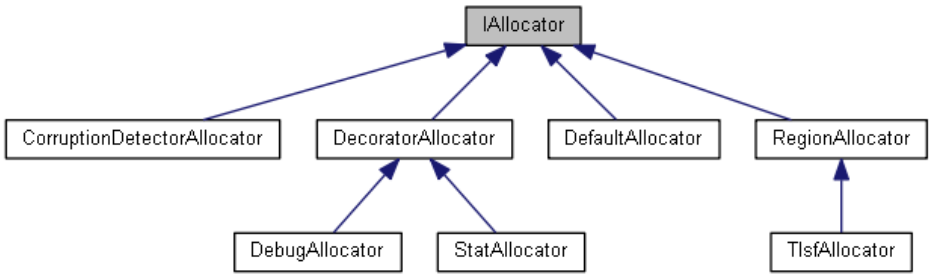
What's Memo

Memo is an open source C++ library that provides data-driven and object-oriented memory management.

The classic scenario of dynamic memory allocation consists of a program requesting randomly dynamic storage to a black-box allocator that implements a set of malloc\realloc\free functions, and that doesn't know and can't predict anything about the requests of the program.

Memo adds a layer between the allocator and the program, and allows the program to select the best memory allocation strategy with the best tuning for every part of the program. In Memo an allocation algorithm is wrapped by a class implementing the interface `IAllocator`.

- the default allocator, which wraps the system malloc\free
- the debug allocator, which decorates another allocator adding no man's land around memory blocks and initializing memory to help to catch uninitialized variables
- the statistics allocator, which decorates another allocator to keep tracks of: total memory allocated, total block count, and allocation peaks
- the tlsf allocator, which wraps the two level segregate allocator written by Matthew Conte (http://tlsf.baisoku.org)



Provided that the key functions in the source code are tagged with contexts, Memo allows to select and tune a different allocator for any context without altering the code, but just editing a memory *configuration file*.

Usage

Just as one can expect, Memo provides a set of global functions to allocate memory, similar to the ones of the standard C library, and a set of macros to allocate C++ objects:

```
void * buffer = memo::alloc( buffer_length, buffer_alignment, 0/*offset*/ );
memo::free( buffer );

Dog * bell = MEMO_NEW( Dog, "Bell" );
```

```
MEMO_DELETE( bell );
```

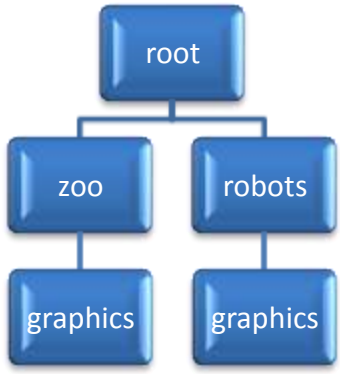
The allocation requests are redirected to an object implementing the interface `IAllocator`. But which one is used? Every thread has its own *current allocator*, that is used to allocate new memory blocks. Anyway, when a realloc or free is requested, the operation is performed by the allocator that allocated the block, regardless or the current allocator of the thread.
The memory configuration file can associate a startup allocator to every thread, otherwise the default allocator is assigned.

Memo allows to change the thread's current allocator, but it's not recommended. The best practice is opening contexts on the callstack:

```
const memo::StaticName g_graphics( "graphics" );

void load_archive( const char * i_file_name )
{
	memo::Context context( g_graphics );
	// ...
	void * buffer = memo::alloc( buffer_length, buffer_alignment, 0/*offset*/ );
	// ...
}
```

The context label is pushed on the calling thread when the object `Context` is constructed, and popped when it goes out of scope. Of course, contexts can be nested:

```
const memo::StaticName g_zoo( "zoo" );
const memo::StaticName g_robots( "robots" );

void load_zoo( const char * i_file_name )
{
	memo::Context context( g_zoo );
	load_archive( i_file_name );
}

void load_robots( const char * i_file_name )
{
	memo::Context context( g_robots );
	load_archive( i_file_name );
}
```



The `Context` objects pushed on the call stack form a context path. The function `load_archive`, called from `load_animals`, sets on the calling thread the context with the path "zoo/graphics". If the same function is called from `load_robots`, the path of the context is "robots/graphics".

The memory configuration file can assign and tune an allocator for:

- the context "robots", and all its child context

- the context “zoo/graphics”
- the context “robots/graphics”

Data Stack

Memo allows to use a thread specific data stack, to perform lifo allocations:

```
size_t required_size = get_required_size();

char * buffer = static_cast< char * >( memo::lifo_alloc( sizeof(char) * required_size,
MEMO_ALIGNMENT_OF(char), 0, nullptr ) );

strcpy( buffer, str1 );
strcat( buffer, str2 );

memo::lifo_free( buffer );
```

Lifo allocations are useful when the program needs a temporary storage with a size known only at runtime.

The lifo order must be respected, otherwise the memory gets corrupted. In a debug build, a mismatch is reported with an assert.

Space and execution overhead

If you allocate memory using directly an `IAllocator` object, you don’t have any space overhead. Anyway every allocation\deallocation has a time overhead due to the virtual call. If you use the global function `memo::alloc` or the macro `MEMO_NEW`, then memo will add, at the beginning of the memory block, a pointer to the current allocator of the thread. This is the space overhead.

Anyway, you may want to use memo just to analyze the memory usage of your program. In this case you can define, in the header memo_externals.h, the macro `MEMO_ONLY_DEFAULT_ALLOCATOR` as 1. In this way, the global function `memo::alloc` and the macro MEMO_NEW will resolve to a static call to the default allocator, to avoid both space and time overhead.

CorruptionDetectorAllocator

Memo includes a special allocator to help to find bugs in the code that causes wrong memory access and memory corrruption. CorruptionDetectorAllocator can detect:

- read accesses to memory that was allocated but never written (i.e. uninitialized memory usage)
- read or write access to memory outside allocated blocks

CorruptionDetectorAllocator runs an external debugger, "memo_debugger.exe", and communicates with it. memo_debugger tracks every read or write access inside the memory managed by CorruptionDetectorAllocator, so the access to this memory is extremely slow.

Currently this allocator is defined only for windows, as it uses windows specific APIs. Internally

CorruptionDetectorAllocator uses the tlsf allocator implemented by Matthew Conte (http://tlsf.baisoku.org).

When memo_debugger detects an invalid memory read or write operation, it breaks the execution of the program, writes about the error on its console window, and allows the user to choose one if these actions:

- detaching the target process, to allow another debugger to attach to it
- resuming the target process, ignoring the error
- quitting, after resuming the target process
- saving a dump that can be opened in a compatible debugger, such visual studio or windbg.
- dumping the memory content around
- saving or copying all the output of the debugger

Here is an example of the output of the debugger:

```
       *********** ERROR: attempt to write unwritable address 0x00790C9C ***********

*** STACK TRACE ***
->     0x0096047A memo::ContextTest::test_CorruptionDetectorAllocator (231)
       0x0095FA95 memo::ContextTest::test (247)
       0x0095F974 memo::test (262)
       0x0094B5A6 main (82)
       0x0098E7FF __tmainCRTStartup (555)
       0x0098E62F mainCRTStartup (371)
       0x75D5ED5C BaseThreadInitThunk (371)
       0x778B37EB RtlInitializeExceptionChain (371)
       0x778B37BE RtlInitializeExceptionChain (371)

*** SOURCE CODE AROUND 0x0096047A ***
module: D:\GitHub\mmemo\test\Debug\test.exe
source file: d:\github\mmemo\memo_test.cpp
       229:                int * array = MEMO_NEW_ARRAY( int, 5 );
       230:                for( int i = 1; i <= 5; i++ )
->     231:                       array[i] = i;
       232:                MEMO_DELETE_ARRAY( array );
       233:

*** MEMORY CONTENT AROUND 0x00790C9C grouped by 4 byte(s) ***
       0x00790C7C: 0x00790000 - not allocated
       0x00790C80: 0x0045842c - readable/writable
       0x00790C84: 0x00000005 - readable/writable
       0x00790C88: 0x40400000 - writable
       0x00790C8C: 0x00000001 - readable/writable
       0x00790C90: 0x00000002 - readable/writable
       0x00790C94: 0x00000003 - readable/writable
       0x00790C98: 0x00000004 - readable/writable
->     0x00790C9C: 0x00790c70 - not allocated
       0x00790CA0: 0x000ff359 - not allocated
       0x00790CA4: 0x00790000 - not allocated
       0x00790CA8: 0x00790000 - not allocated
       0x00790CAC: 0x00000000 - not allocated
       0x00790CB0: 0x00000000 - not allocated
       0x00790CB4: 0x00000000 - not allocated
       0x00790CB8: 0x00000000 - not allocated
       0x00790CBC: 0x00000000 - not allocated
```

```
Type a command:
       detach: detach the target process, to allow another debugger to attach to it
       ignore: resume the target process, ignoring the error
       quit: resume the target process, and quit
       minidump [file=test_memo_mini.dmp]: save a minidump
       dump [file=test_memo.dmp]: save a complete dump
       mem [1|2|4|8]: dump memory content around 0x00790C9C
       save [file=test_memo_output.txt]: save all the output of this program
       copy: copy all the output of this program into the clipboard
>>
```