# 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 (implementing a set of malloc\realloc\free functions) which doesn’t know and can’t predict anything about the requests of the program. Implementing a good black-box is a well known difficult problem.

The main point of memo is not providing memory allocation algorithms, but rather adding a layer between the allocator and the program, to allow to select the best memory allocation strategy with the best tuning for every part of the program. 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 (which uses the system malloc) 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”

# Allocators

Memo provides a toolkit of allocators

for example [region based](http://en.wikipedia.org/wiki/Region-based_memory_management) allocator

Memo includes among the built-in allocators the wrapper to some dynamic memory allocators written by third parties (currently the “[Two Level Segregated Fit memory allocator](http://tlsf.baisoku.org/)”). The code of these allocators is embedded in Memo under the directory external\_sources. Finally Memo provides some more specific features, such stack-based (lifo) allocations, and type-specific data-driven memory pools.

