C-Memo: A Generic Function Memoization Library for C (Manual)
By: Benjamin D. Nye
Date: September, 2011

1 Overview

The C-Memo library is intended to be used for rapid deployment of memoization to C functions. Function memoization is a type of optimization that enables a cache that maps between set of function argument values to a function output (Michie, 1968). The first time a function is called with a given set of argument values, it will calculate the value and cache this value. After this calculation, the return value is cached in a map where a set of function values maps to an output value. High-level such as Python and LISP are able to implement generic function memoization relatively elegantly, since arguments are much more abtract than those in C. Shockingly, after a significant search, no memoization libraries appeared to exist for the C language. To address this shortcoming and to approach an optimization problem that was currenly in progress, the C-Memo memoization library was created. The primary use-case for creating this library was to have a drop-in memoization capability that required minimal changes to the original function definitions. This was accomplished by creating a simple macro that can be used to replace the function header of a function to implement memoization.

A secondary use-case for this library was to allow the management of different memoization contexts, to allow invalidating different subsets of cached values while retaining other values. This increases the flexibility of the library when dealing with state-based systems. For example, functions querying a database may need to have their cached values cleared when the DB changes, while simple functions such as factorials will have their cached values remain valid. Memoization contexts also allow a code developer to turn caching on and off for different sets of functions, to prevent storing certain results or to force a recalculation of a value. In theory, memoization contexts can also be used to manage the cache of each individual memoized function as well (by using each context with only one memoized function). This fine-grained control may be useful for some systems.

2 Technical Details

The primary hurdle for a generic memoization library is that C and the C-Preprocessor (CPP) have extremely limited capabilities when dealing with variadic functions. C is a machine-level language, in which a variadic function represents a stack or memory block that is iterated over. The C-Preprocessor is an extremely basic text manipulation language, lacking even the ability to split strings. Neither language provides significant capabilities to transmit the types of arguments at runtime. This is a major hurdle for generic function memoization: how can the library store the arguments to a function without knowing how many exist or what type they are?

This challenge was solved by creating a macro that would replace the function header definition and create two functions. One function would implement the original function code as a private function. The other function would be a function-wrapper for the original function, providing cache set and fetch functionality. This approach is portable and relatively straightforward. Its only significant drawback is a slight pollution of the c module namespace. This approach also makes debugging significantly easier- the macro never has to deal with the function body of the original calculation function. Instead,

the macro only needs to generate a function wrapper and a new function header for the original function. So then, the function memoization macro has the primary purpose of generating a function wrapper for implementing memoization. This is similar to how memoization is often implemented in Python under the hood.

The function wrapper is generated using a CPP macro called MEMOIZE_FUNCT. This macro receives the following information: the memoization context ID (where to store the data), the function return type, the cached function name, and the type and name of each argument. Unfortunately due to the limitations of the preprocessor, this macro requires that each argument type and argument name be separated by a comma (e.g. 'int, arg1, long, arg2'). From the macro's point of view, these must be separate pieces of information since there is no way to separate them if they are passed in as an individual argument. While this prevents the macro from being a complete drop-in piece of code, there is really no way to work around this restriction- the standard C preprocessor contains no function to split strings. With this information in hand, it is straightforward to count the number of arguments and the total size of arguments by using variants of the PP_NARG macro for counting the number of arguments passed to a macro.

The PP_NARG macro is an approach for counting the number of arguments to a C macro (Deniau, 2010). It does so by appending a numeric count to the arguments and using a second variadic macro to return the value at a fixed point of the cummulative argument list. Since the value returned is be a member of the numeric count and the position in that count is shifted by the number of original arguments, PP_NARG can return the number of arguments passed to a variadic macro. Using this value and string concatenation, it is possible to implement FOR_EACH macros that can run various macros internally on the arguments. The FOR_EACH macros rely on a dispatch design- each number of arguments is used to generate the name of a particular macro that will iterate over that number of arguments. While this is clunky, the C preprocessor does not give any real alternatives. Using this approach, macros were created for the following tasks: calculating the sizeOf all arguments (SIZE_OF_ARGS), storing arguments (STORE_ARGUMENTS), merging argument pairs (MERGE_ARGUMENTS _PAIRWISE), filtering even arguments (FILTER_EVEN_ARGUMENTS), and filtering odd arguments (FILTER_ODD_ARGUMENTS). These functions are sufficient to process the information present in the MEMOIZED_FUNCT macro and generate a unique signature for a given function call (FUNCTION_MEMOIZATION_SIGNATURE).

The function wrapper calculates the number of arguments for the function, the sizeOf the arguments, and stores all argument values into a continguous block of allocated memory. Combined with the function pointer, this is sufficient information to capture the complete information of about a call to a given function. This information is stored in a function signature (FUNCTION_MEMOIZATION_SIGNATURE) which is used as a key in a hash map linking function signatures to their output values. Each memoization context contains its own hash map, which is managed separately from all other contexts. The khash library was used for hash map access, because it was lightweight and relatively efficient (Chaos, 2008). Since khash does not contain an efficient hashing algorithm for calculating hash keys from data structures, the murmur_hash3 library was used to generate the hashing key for function signatures (Appleby, 2011).

The memoization storage itself is relatively straightforward, operating as a standard cache in which cache misses are added to the cache after calculation. In terms of management, the only non-standard functionality made available is the ability to enable or disable a cache context. This lets the program prevent caching and cache retrieval for a memoization context, to prevent caching when global variables or data used by those functions might be in flux. Otherwise, all management functions are the standard cache-management capabilities: initialization, clearing, and destroying.

As a final word, this library has some significant limitations. Some of these limitations are easily adjusted, while some are inflexible. The first major limitation is that the library is only intended to memoize functions with 0 to 64 arguments. This boundary is easily extensible by extending PP_NARG to a larger counting set and by extending all the enumerated macro functions to have appropriate dispatch macro names. With that said, if you are using more than 64 arguments in your functions it may be time to re-evaluate your coding style instead. Secondly, it does not allow memoization of variadic functions. While the storage format allows this, the MEMOIZED_FUNCT macro does not. It is probably possible to implement a second macro of that nature to handle memoizing variadic functions. This might be a direction for future development of this library for a secondary author if they are so interested. Finally, as with most memoization libraries, pointers are taken at face value. This means that the meaning of a pointer must be fully captured by its address for the purposes of using the function. This also means that if the function returns a block of allocated memory, this memory allocation will only occur when the function calculates a value (and never when it reads from the cache).

3 Usage

Using the library consists of two components: Cache management and Memoizing Functions. Each will be described briefly here.

3.1 Memoization Cache Management

Memoization cache management is the process of managing the storage container for memoized functions. A cache has a few basic functions:

- newMemoizationContainer - Create a memoization container
- freeMemoizationContainer - Free the container
- isMemoizationContextEnabled - Check if a context is enabled
- isMemoizationContextDirty - Check if a context might have been modified
- enableMemoizationContext - Enable the context
- disableMemoizationContext - Disable the context (no caching used)
- clearMemoizationContext - Clear the cache for the context
- clearMemoizationContainer - Clear all contexts in the container
- setMemoizationCacheValue - Set a value in the memoization context cache
- fetchMemoizationCacheValue - Get a value from the memoization context cache

For the current implementation, creating a new global memoization container is mandatory. The memoization get and fetch functions assume such a container exists when they make their calls. This would be relatively simple to relax, however, by adjusting the memoization to take a memoization container name as one of the arguments to MEMOIZED_FUNCT. Additionally, all memoization contexts start in the disabled state. They must be enabled for caching to occur. Finally, to prevent memory leaks, the memoization container should be freed after it is no longer in use. The example file memoizationTests.c gives an example of proper usage.

3.2 Memoizing Functions

To memoize a function is relatively straightforward, but requires careful attention to detail. This is particularly important because preprocessor functions are inherently fragile and ugly to debug. If you run into issues, it is recommended to generate the intermediate file by directly calling the c preprocessor (cpp) and dumping the output to a file. To add memoization to a function, the macro MEMOIZED_FUNCT found in memoization.h should be used. This macro takes the form:

```
1 MEMOIZED\_FUNCT(CONTEXT\_ID, RETURN\_TYPE, FUNCT\_NAME, ...)
```

The context ID is the context where the function should cache to, the return type is the function return type, and the function name is the name for the memoized function. The intended use of this macro is to allow a nearly drop-in replacement of the original function.

The macro for memoizing a function only needs to touch the function header and can leave the function body entirely untouched. For example, if one had a function in the form:

```
1 long myFunct(long val, long x){
2         return val+x;
3 }
```

This could be turned into a memoized function by changing this to:

```
1 MEMOIZED_FUNCT(0, long, myFunct, long, val, long, x){
2         return val+x;
3 }
```

In practice, this will resolve to two functions. One of those functions is the function wrapper that implements function caching. The other function is a function that is internal to the C file. The above example resolves to the following code after the preprocessor has executed:

4 Troubleshooting

The most effective tool for troubleshooting issues with the memoization macro is to use the C Preprocessor. Unfortunately due to how the preprocessor is implemented, all the whitespace will be stripped out (including line breaks). This means that the actual intermediate output will be a bit less readable. However, it is the only view for the generated C code. Usage errors and mistakes can also occur when managing the memoization cache. A few typical errors are likely and will be enumerated here.

4.1 Compilation Errors

Two types of compile errors are possible: preprocessor errors and compiler errors. Preprocessor errors will typically occur when the number of arguments is even (the number of arguments must always be odd). The number of arguments must be even because all arguments are (type, name) pairs except for the contextID. Compile errors

```
long myFunct(long val, long x){
        <cached function that calls myFunct_MEMOIZED_CONTEXT_0 when it needs a value>
}

long myFunct_MEMOIZED_CONTEXT_0(long val, long x){
        return val+x;
}
```

will typically occur when arguments are incorrectly specified, such as 'long val, long x' instead of 'long, val, long, x'. These errors are likely to occur for a few basic reasons.

4.1.1 No Context Specified

Note carefully that the first value must be the number of the context to cache this function in. If this number is omitted, the macro will use the first argument given in place of the context. If everything else is correct, this will encounter a preprocessor error (which implies an incorrect number of arguments).

4.1.2 Missing Comma Between Type and Name of Argument

Also note that the type and name for each argument are separated by a comma. When changing a function heading to this format, those commas are easy to overlook. Forgetting those commas will lead to either preprocessor or compilation errors. If an odd number are forgotten, the most likely outcome is a preprocessor error. Otherwise, a compiler error will occur due to the code attempting to pass something like 'long x' as an argument to sizeOf.

4.2 Runtime Errors and Problems

The most common runtime errors will be due to failure to initialize the global memoization container before using it.

4.2.1 Failure to Initialize the Memoization Container (Segfault)

On startup of the program, it is necessary to run 'initGlobalMemoizationContexts' before using any memoized functions. Failure to do so will result in a crash due to attempting to access a null pointer.

4.2.2 Disabled Memoization Context

The second possible runtime issue is that a memoization context is not enabled before attempting to use it. This will not cause any failures, but no caching will be applied to any functions stored in that context. Make sure to enable each context using 'enableGlobalMemoizationCo when this context is applicable for caching.

4.2.3 Outdated Cache Values

Some systems reach conditions where cached values for a context are no longer applicable and the cache should be repopulated. In that case, please run 'clearGlobalMemoizationContext'

on the given context to clear out the old cache data. If functions are not giving the expected results, this is a possible cause.

4.2.4 Pointers as Arguments

Pointers are used at face value. A cached function call stores the actual memory address for the pointer, not the data at that address (which it would have no way to store because it has no way to know the size of said data). So then, be very careful using pointers as arguments to a memoized function. It may not be accomplishing the intended purpose.

4.2.5 Function Runs Slower or Same Speed

To note, this is a generic library- not an optimized approach to speeding up any given function. Many functions, especially small functions, will run faster without caching. This is because the caching process is relatively expensive. First, a function signature and hash key must be generated (which requires memory allocation). Second, the hash map must be checked for this signature. Finally, a copy of the value must be returned. If the original function works faster than this process, caching will certainly slow it down. Of these, generating the function signature is the most costly due to its use of memory allocation and copying. Alternatives to this exist (e.g. creating a special struct for each function to store the arguments), but have their own drawbacks. This is an area which could be optimized more for speed if needed.

On the other hand, for costly and time consuming functions this approach can lead to substantial speedups. As always, profiling is an excellent approach to seeing the benefits of employing this memoization library to a particular function or functions. The bright side of this library is that it makes it easy to apply caching to functions. Moreover, by simply commenting out the old function header rather than replacing it, it is straightforward to compile a function with or without memoization.

5 Acknowledgments

Thank you to Lustick Consulting for providing the funding and the necessity to pursue this work as part of the PS-I agent-based simulation project (Lustick, 2002). This code is used for optimizing some of the costlier functions in the PS-I project and was originally developed as an optimization tool for that context.

References

Appleby, A. (2011). *Murmurhash3 library.* `http://code.google.com/p/smhasher/`(Retrieved July 2011).

Chaos, A. (2008). *khash.h fast and light-weighted hash table library in c.* `http://attractivechaos.awardspace.com/` (Retrieved June 2011).

Deniau, L. (2010). *Pp narg macro.* `https://groups.google.com/group/comp.std.c/browse_thread/thread/77ee8c8f92e4a3fb` (Retrieved June 2011).

Lustick, I. (2002). Ps-i: A user-friendly agent-based modeling platform for testing theories of political identity and political stability. *Journal of Artificial Societies and Social Simulation*, *5*(3).

Michie, D. (1968). Memo functions and machine learning. *Nature*, *218*(1), 19–22.