Heap Memory Manager

Version 1.1

Copyright (c) 2002-2016 Walter William Karas

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1 Introduction

This document describes a portable heap memory manager, also known as a dynamic memory allocator. The manager consists of several functions implemented in ANSI C. These functions have worst-case time complexity of O(log N), where N is the number of distinct free memory block sizes. Allocation is done on a strict best-fit basis. Adjacent free memory blocks are always coalesced into a single, larger free block.

This document, as well as the source code it describes, is in the public domain.

The heap facility provided by this manager is similar to the heap facility provided by the `malloc` and `free` functions in the C standard library. A successful allocation returns a contiguous range of bytes in memory of at least the requested size. Allocated memory has to be explicitly freed/deallocated (no garbage collection).

The manager code (optionally) performs limited self-consistency checks in order to detect heap corruption.

This manager could be used to implement the `malloc`/`free` functions in the C standard library. It could also be used to implement a heap memory manager for shared memory, which in turn could be used to implement allocators of shared memory for C++ Standard Template Library containers.

2 Source Files

The files containing the source code for the manager are:

File	**Contents**
heapmm.h	The header file that defines the interface to the heap memory manager.
hmm_cnfg.h	The header file containing definitions that may have to be changed for a particular port of the heap manager. Included by heapmm.h.
hmm_intrnl.h	The internal header file for the manager. Files external to the manager should not include

	this file.
hmm_base.c	The implementation file for basic functions that are always needed when using the heap manager.
hmm_alloc.c	The implementation of the `hmm_alloc` function.
hmm_grow.c	The implementation of the `hmm_grow_chunk` function.
hmm_true.c	The implementation of the `hmm_true_size` function.
hmm_largest.c	The implementation of the `hmm_largest_available` function.
hmm_resize.c	The implementation of the `hmm_resize` function.
hmm_shrink.c	The implementation of the `hmm_shrink_chunk` function.
hmm_dflt_abort.c	The implementation of the function that aborts execution if the manager's (optional) self-auditing finds corruption of the heap.
test_hmm.c	Test suite for the manager.
test_hmm.h	Header file for test suite (only).
speed_hmm.c	Program that (very crudely) measures the speed of manager functions versus the `malloc` and `free` functions of the standard library.

These files, along with this document in PDF (heapmm.pdf) are in the zip archive heapmm.zip.

The source code for the manager includes the header files cavl_if.h and cavl_impl.h from the C AVL Tree Generic Package (version 1.4 or later).

3 Assumptions and Definitions

The first assumption is that `sizeof(char)` is `1`. This is required by the ISO C++ standard. I'm not sure it's required by ANSI C, but I've never seen a C implementation where this wasn't the case.

If a pointer p of type `char *` contains an address such that `(* (T *) p)` is a valid lvalue for any type T, then p contains an *aligned address*.

The *address align unit (AAU)* is some positive number of bytes N such that, if p, a pointer of type `char *`, contains an aligned address, then the expressions `p + N` and `p – N` also evaluate to aligned addresses.

In order for the manager to be ported to a processor architecture, a valid address align unit for the processor must exist. The smaller the AAU, the fewer wasted "pad" bytes will be necessary. Generally, to avoid skewed memory reads and writes, you should use the width in bytes of your processor's data bus as the AAU. For that reason, the default AAU is `sizeof(int)`. The preprocessor symbol

`HMM_ADDR_ALIGN_UNIT`, defined in hmm_cnfg.h, translates to the address align unit. If eliminating pad bytes concerns you more than avoid skewed memory operations, you can make the AAU even smaller. An AAU of 1 can be used for the Intel x86, VAX and other architectures which do not generate bus errors. An AAU of 2 can be used for the Motorola 68000 and the PDP-11 architectures.

A *chunk* of memory is a large contiguous range of addresses, which the manager subdivides to satisfy allocation requests. The subdivided pieces of a chunk are called *blocks*. The first byte of a chunk must be at an aligned address. The number of bytes in a block or chunk must be a whole number of *block align units (BAUs)*. A block align unit is a whole number of AAUs. The preprocessor symbol `HMM_BLOCK_ALIGN_UNIT`, defined in hmm_cnfg.h, translates to the block align unit. The default BAU is simply one AAU. Using a larger BAU causes the sizes of blocks to be rounded up with pad AAUs, thus wasting some space. The advantage is that the number of distinct block sizes is reduced. The time it takes to allocate and free memory increases as the number of distinct block sizes increases.

The implementation depends on the assumption that `sizeof(T *) == sizeof(void *)` for any type `T`. It also assumes that, if `p` is a pointer variable of type `void *`, and p is null, then `*(T **) &p == (T *) 0` for any type `T`.

4 Interface

The prototypes for these functions and the typedefs are provided by heapmm.h.

4.1 Typedef `hmm_size_aau`

An unsigned integral type with enough precision to hold the size of a block in AAUs. `unsigned long` by default.

4.2 Typedef `hmm_size_bau`

An unsigned integral type with enough precision to hold the size of a block or chunk in BAUs. `unsigned long` by default.

Let N be the number of bits of precision in `hmm_size_bau`. Due to details of the implementation, the maximum chunk size is the largest unsigned number of precision N – 1. For example, if `hmm_size_bau` is a 32-bit unsigned integer, the maximum chunk size is $2^{31} – 1$. If you attempt to add or grow a chunk in excess of the maximum size, the results are undefined.

4.3 Typedef `hmm_descriptor`

The descriptor for a heap that is managed by this manager. The fields `num_baus_can_shrink` and `end_of_shrinkable_chunk` should be treated as read-only by the user code. See the descriptions of the `hmm_free` and `hmm_shrink_chunk` functions for the usage of these two fields. The other fields of this structure should be treated as private (neither readable nor writable) by the user code.

4.4 Function `hmm_init`

```
void hmm_init(hmm_descriptor *desc);
```

Initializes the descriptor to an empty state. A descriptor must be initialized before passing it to any other function.

4.5 Function `hmm_new_chunk`

```
void hmm_new_chunk(hmm_descriptor *desc, void *first_byte_of_chunk,
hmm_size_bau num_block_align_units_in_chunk);
```

Assigns a chunk to the heap with the given descriptor. The chunk can then be subdivided to satisfy allocation requests. A chunk can only be assigned to a single heap, but a heap can have multiple chunks assigned to it. The first byte of the chunk has to be at an aligned address.

Assigning very small chunks to a heap will result in corruption. ("Very small" means smaller than `MIN_BLOCK_BAUS + DUMMY_END_BLOCK_BAUS`. These values are defined in hmm_intrnl.h.)

4.6 Function `hmm_alloc`

```
void * hmm_alloc(hmm_descriptor *desc, hmm_size_aau
num_addr_align_units);
```

Allocate a contiguous range of AAUs of size `num_addr_align_units` from a heap. A pointer to the first byte of the allocated memory is returned if the function is successful. A null pointer is returned if the function fails.

4.7 Function `hmm_free`

```
void hmm_free(hmm_descriptor *desc, void *mem);
```

Frees (deallocates) memory previously allocated from the given heap. The `mem` parameter has to be same value returned by the `hmm_alloc` function.

If, after the free operation, it would be possible to shrink the size of a chunk, the descriptor field `num_baus_can_shrink` will be set to the maximum number of block align units the chunk could be shrunk by. If it is not possible to shrink any chunk as a result of the free operation, `num_baus_can_shrink` will be set to zero.

If `num_baus_can_shrink` is set to a non-zero value, the (void) pointer field `end_of_shrinkable_chunk` (also in the descriptor) will point to one byte after the last byte in the chunk that can be shrunk.

4.8 Function `hmm_grow_chunk`

```
void hmm_grow_chunk(hmm_descriptor *desc, void *end_of_chunk,
hmm_size_bau num_block_align_units_to_add);
```

Adds space to a chunk at the end. The chunk must already have been assigned to the heap whose descriptor is given. `end_of_chunk` should point to the first byte after the last byte in the chunk (prior to growing).

Growing a chunk by a very small amount may result in corruption. ("Very small" means smaller than `MIN_BLOCK_BAUS`. This value is defined in hmm_intrnl.h.)

This function is redundant, in the sense that you could add the additional heap space as a new chunk, rather than growing an existing chunk. The reason to grow the existing chunk instead is that the manager is not smart enough to coalesce adjacent free blocks in different chunks, even when the chunks are adjacent. There is also a small amount of per-chunk overhead.

4.9 Function `hmm_true_size`

`hmm_size_aau hmm_true_size(void *mem);`

Sometimes, when you allocate some memory, you will actually get a few more AAUs than you requested. This function returns the number of AAUs that are actually usable in a currently allocated block. `mem` has to be pointer value that was returned by `hmm_alloc`.

4.10 Function `hmm_largest_available`

`hmm_size_aau hmm_largest_available(hmm_descriptor *desc);`

Returns the largest number of AAUs that could successfully be allocated (by a single call to `hmm_alloc`) from the heap with the given descriptor.

4.11 Function `hmm_shrink_chunk`

```
void hmm_shrink_chunk(hmm_descriptor *desc, hmm_size_bau
num_block_align_units_to_shrink);
```

Shrinks a chunk. (The starting location of the chunk does not change, only the ending location.) This function should only be called immediately after a call to `hmm_free` that set the `num_baus_can_shrink` descriptor field to a non-zero value. No other manager functions should be called between the calls to `hmm_free` and `hmm_shrink_chunk`. The value passed for `num_block_align_units_to_shrink` has to be less than or equal to `num_baus_can_shrink`. The (void) pointer `end_of_shrinkable_chunk` in the descriptor, as set by `hmm_free`, points to the first byte past the last byte in the chunk that will be shrunk. (User code must not change the value of `end_of_shrinkable_chunk`.)

If the descriptor field `num_baus_can_shrink` contains the number of BAUs in the chunk, the chunk can be made to "disappear" by this function. But, if an attempt is made to shrink the chunk to a very small size, corruption will result. ("Very small" is smaller than `MIN_BLOCK_BAUS + DUMMY_END_BLOCK_BAUS`. These values are defined in hmm_intrnl.h.)

4.12 Function `hmm_resize`

```
int hmm_resize(hmm_descriptor *desc, void *mem, hmm_size_aau
new_num_addr_align_units);
```

Attempts to change the size of a previously-allocated block. `mem` must be pointer that was returned by `hmm_alloc`. The function returns 0 if it was able to resize the block to the desired size. Otherwise, the function returns -1.

Note that this function will not move the block the way the C Standard Library function `realloc` will.

5 Overview of Implementation

Each block in a chunk starts on an aligned address, and consists of a whole number of block align units. At the beginning of each block is the block *head*. The block head consists of the following structure:

```
typedef struct head_struct
  {
    hmm_size_bau previous_block_size, block_size;
  }
head_record;
```

`hmm_size_bau` is the (configurable) unsigned type for holding sizes of blocks and chunks. The most significant bit of `previous_block_size` and the most significant bit of `block_size` are combined together to form a *block status field*. If both bits in the block status field are zero, this indicates that the block has been allocated (by a call to `hmm_alloc`). Any of the other 3 possible combinations of values of the bits in the block status field indicate that the block is free. The remaining bits is `block_size` contain the number of block align units in the block. The remaining bits in `previous_block_size` contain the block size of the preceding block, or zero if the block is the first block in a chunk. The size of the block head is the minimum number of address align units required to hold a `head_record`.

Each chunk ends with a *dummy end block*. In the dummy end block, both bits in the block status field are zero (although it cannot be allocated). `block_size` contains zero, and `previous_block_size` contains the size of the last (non-dummy) block in the chunk. The size of a dummy end block is the minimum number of block align units need to hold a `head_record`.

The remainder of a (non-dummy) block after the head is the block *payload*. In an allocated block, the payload contains user data. When you call `hmm_alloc`, the pointer returned is a pointer to the first byte of the payload of the block that was allocated for you. In a free block, the payload contains a *pointer record*. A pointer record has this structure:

```
typedef struct ptr_struct
  {
    struct ptr_struct *self, *prev, *next;
  }
ptr_record;
```

Free blocks are held in a data structure called the *free collection*. The pointers in the pointer record are used to link a free block into the free collection. Each free block is located in a *bin* along with all other free blocks of exactly the same size. The first block in each bin is a node in an AVL tree keyed by block size. The `self` and `prev` pointers in the block's pointer record are the node's child pointers. The node's balance factor (-1, 0 or 1) is encoded using the 3 possible bit value combinations for a free block in the block status field. The `next` pointer points to the pointer record of the next block in the bin (or is null if there is only one block in the bin).

For blocks in a bin after the first, `prev` and `next` point to the previous and the next block in the bin (making the bin a doubly-linked list). `self` points to the pointer record it is inside of. (This will make sense in a minute.) The block status field contains any of the 3 possible "free" values (doesn't matter which).

When allocating a block, you search the AVL tree for the smallest block whose payload is the same size or greater than the size requested. If there are other blocks in the bin of the block you find, you grab the second block in the bin. This avoids a (relatively) time-consuming change to the AVL tree structure. But if there is only one block in the bin, you have to delete it from the AVL tree. If the "extra" space in the block is big enough, you split the extra space off as a new block. You then put this new block into the free collection.

When freeing a block, you check to see if the adjacent blocks are free. (Since the previous block's size is in the head of the block you're freeing, it's easy to locate the previous block's head.) If (one or both) adjacent blocks are free, you coalesce them into a single free block. Before coalescing an adjacent block, you have to take it out of the free collection. You look at the `self` pointer in the pointer record of the adjacent block. If the `self` pointer is actually pointing to itself, the adjacent block can't be the first block in a bin, because a node in an AVL tree cannot be its own child. In this case, taking the adjacent block out of the free collection is simply a matter of deleting it from the doubly-linked list for the bin. If you're not lucky, and the adjacent block is an AVL tree node, it's better to substitute another block from the bin into the adjacent block's position in the tree. (Deleting a node from an AVL tree can result in a lot of complex rebalancing.) But if the adjacent free block is alone in its bin, deleting it from the AVL tree is unavoidable. After any coalescing is complete, you insert the newly-freed block into the free collection.

It's a common pattern in programs that a bunch of blocks are allocated at about the same time, then freed later at about the same time. A group of successive allocations are often satisfied by subdividing a single large block, causing the allocated blocks to be located adjacently in the order of allocation. If these blocks are freed together, they coalesce back into the original big block. To optimize for this case, when a block is freed by a call to `hmm_free`, the block is not immediately put into the free collection. This give the block a chance to coalesce with succeeding blocks that are freed before taking the time to do an insertion into the free collection. (There is a pointer in the heap descriptor to keep track of the last freed block.)

I chose to implement this in C rather than C++ because of the widespread (but unjustified) resistance to using C++ in low-level programming. There are a few macros in the code that translate to expressions with multiple occurrences of macro parameters. I tried to not pass expressions as parameters to these macros, but it wasn't always easy to avoid it. So, if you're using C++, and your faith in your compiler's repeated subexpression detection is weak, you might want to convert these macros to inline functions.

6 Version History

Version 1.0

- Initial version.

Version 1.1

- Added `hmm_shrink_chunk` and `hmm_resize` functions.