Documentation for MCell Memory Utilities

Rex Kerr

6th February 2004

The memory utilities can be found in the files `mem_util.h` and `mem_util.c`

1 stack_helper

The `stack_helper` struct and functions implement a hybrid array/linked-list stack of a set of items of the same size. The main struct has an array of data of a specified size, plus a pointer to the next part of the stack should the first part overflow. There are seven functions for dealing with stacks:

```
struct stack_helper* create_stack(int size,int length);
void stack_push(struct stack_helper *sh,void *d);
void stack_pop(struct stack_helper *sh, void *d);
void stack_dump(struct stack_helper *sh);
inline int stack_size(struct stack_helper *sh);
void* stack_access(struct stack_helper *sh,int n);
void delete_stack(struct stack_helper *sh);
```

To start, one creates a new stack using the `create_stack` function. The first argument is the size of your data structure (e.g. `sizeof(struct my_struct)`). The second argument is the number of elements in an array of that data structure. This number should be chosen small enough to not overburden memory, but large enough so the stack can function primarily with array access rather than with list-traversal.

You can then push and pop items of that data type onto and off of the stack using the `stack_push` and `stack_pop` functions. Note that these *copy* the data, so stacks are best used for small structs or other types.

If you wish to clear out the stack, use `stack_dump`. To see the current size of the stack, use `stack_size`. (Zero means the stack is empty.) To get a pointer to one element of the stack, use `stack_access`. The oldest item on the stack has an index of `0`; the most recent has an index of `stack_size(...)-1`.

Calling `delete_stack` will delete everything you've pushed onto the stack, and will free the stack_helper itself. If you wish to only empty the stack but keep using it, use `stack_dump` instead.

Note: stacks are slow in the current implementation if the stack is many times longer than the length of the array, as it has to wade down a long linked list. Need to fix this. (Easy enough, just swap so the first thing always is the one with space!) Index-based access will always be slow, though (can't avoid traversing the list).

2 mem_helper

The `mem_helper` struct and functions implement a hybrid array/linked-list block-memory allocation specifically for linked lists. The main struct has an array of data of a specified size, plus a pointer to the next allocation block should the first part run out of space. It also maintains a linked list of list elements that have been deallocated so that they can be reused. There are five functions for this utility:

```
struct mem_helper* create_mem(int size,int length);
void* mem_get(struct mem_helper *mh);
void mem_put(struct mem_helper *mh,void *defunct);
void mem_put_list(struct mem_helper *mh,void *defunct);
void delete_mem(struct mem_helper *mh);
```

To start, one creates a new helper with the create_mem function. The first argument is the size of your data structure (e.g. `sizeof(struct my_struct)`) and the second is the number of those structures to allocate in each chunk.

You then can use `mem_get` in place of `malloc` to get a pointer to the start of a data structure, and `mem_put` instead of `free` when you are done with one of your list elements. If you have a linked list and you wish to free all of them, use `mem_put_list` on the head of the linked list. When you're done with everything you've created with that helper, call `delete_mem` and all memory you have allocated, plus the `mem_helper` struct itself, will be freed.

3 temp_mem

If you want to create a bunch of objects using `malloc` and don't want to worry about freeing them all individually, use the `temp_mem` struct and functions. There are only three functions:

```
struct temp_mem* setup_temp_mem(int length);
void* temp_malloc(size_t size,struct temp_mem *list);
void free_temp(struct temp_mem *list);
```

Start off by calling `setup_temp_mem` with an argument that estimates the number of separate items you'll be mallocing (the pointers will be stored on a `stack_helper` stack). Then, just use `temp_malloc` instead of `malloc`, and when you're done with everything you've `temp_malloc`'ed, call `free_temp`. Simple!

4 counter_helper

The `counter_helper` struct and functions are a way to make a set (in the mathematical sense) out of a list of items. In particular, `counter_helper` will find identical items and keep track of the number of that type of item rather than storing each one individually. This numbering is kept track of in the `counter_header` struct. The following functions are for use with counter_helper:

```
struct counter_helper* create_counter(int size,int length);
void counter_add(struct counter_helper *ch,void *data);
void counter_reset(struct counter_helper *ch);
struct counter_header* counter_iterator(struct counter_helper *ch);
struct counter_header* counter_next_entry(struct counter_header *c);
void counter_read(struct counter_helper *ch,struct counter_header *c,void *data);
void delete_counter(struct counter_helper *ch);
```

As usual, you start with `create_counter` and specify the size of your struct and the number of items to allocate at once. (`counter_helper` uses `mem_helper`.) You can then add items using counter_add, where the items will be binned into groups and counted as you go. This method *copies* the data from the individual items. (This is implemented using linked lists and therefore is slow for large numbers of items! If you want to throw away the items you've collected so far, use `counter_reset`.

Once you've added all the items you wish to (or before, if you please), you can traverse the counted set of items by calling `counter_iterator` to point to the first item in the set (returns a `counter_header` as an iterator), and then `counter_next_entry` on that iterator to get the next one. If you want to read out the data stored at a particular location, use `counter_read` to copy the data in the counter into the pointer you provide.

Finally, when you're done, `delete_counter` will delete the `counter_helper` and everything contained within. None of the items you added will be deleted, since `counter_helper` creates copies of the data rather than using the originals.