Memory 1/mem

Purpose

To allocate memory suitable for the dynamic creation of objects of different sizes, placing some larger objects automatically into doubly linked lists and assigning each a unique allocation ID number.

Definitions

¶1. This section is slightly simplified, but essentially copied, from the memory allocator used in the main Inform 7 compiler.

It allocates memory as needed to store the numerous objects of different sizes, all typedef'd structs. There's no garbage collection because nothing is ever destroyed. Each type has its own doubly-linked list, and in each type the objects created are given unique IDs (within that type) counting upwards from 0.

¶2. Before going much further, we will need to anticipate what the memory manager wants. In order to keep the doubly linked lists and the allocation ID, every structure subject to this regime will need extra elements holding the necessary links and ID number. We define these elements with a macro (concealing its meaning from all other sections).

Smaller objects are stored in arrays, and their structure declarations do not use the following macro.

```
define MEMORY_MANAGEMENT
    int allocation_id;                    Numbered from 0 upwards in creation order
    void *next_structure;                 Next object in double-linked list
    void *prev_structure;                 Previous object in double-linked list
```

¶3. There is no significance to the order in which structures are registered with the memory system, but `NO_MEMORY_TYPES` must be 1 more than the highest MT number, so do not add to this list without incrementing it. There can in principle be up to 1000 memory types.

```
define auxiliary_file_MT 0
define skein_node_MT 1
define chunk_metadata_MT 2
define placeholder_MT 3
define heading_MT 4
define table_MT 5
define segment_MT 6
define request_MT 7
define template_MT 8
define template_path_MT 9
define NO_MEMORY_TYPES 10             must be 1 more than the highest _MT constant above
```

---

§**1.** For each type of object to be allocated, a single structure of the following design is maintained. Types which are allocated individually, like world objects, have `no_allocated_together` set to 1, and the doubly linked list is of the objects themselves. For types allocated in small arrays (typically of 100 objects at a time), `no_allocated_together` is set to the number of objects in each completed array (so, typically 100) and the doubly linked list is of the arrays.

```
typedef struct allocation_status_structure {
    actually needed for allocation purposes:
    int objects_allocated;                    total number of objects (or arrays) ever allocated
    void *first_in_memory;                                  head of doubly linked list
    void *last_in_memory;                                   tail of doubly linked list
    used only to provide statistics for the debugging log:
    char *name_of_type;                                     e.g., "lexicon_entry_MT"
    int bytes_allocated;          total allocation for this type of object, not counting overhead
    int objects_count;                        total number currently in existence (i.e., undeleted)
    int no_allocated_together;                number of objects in each array of this type of object
} allocation_status_structure;
```

The structure allocation_status_structure is private to this section.

§**2.** The memory allocator itself needs some memory, but only a fixed-size and fairly small array of the structures defined above. The allocator can safely begin as soon as this is initialised.

```
allocation_status_structure alloc_status[NO_MEMORY_TYPES];

void start_memory(void) {
    int i;
    for (i=0; i<NO_MEMORY_TYPES; i++) {
        alloc_status[i].first_in_memory = NULL;
        alloc_status[i].last_in_memory = NULL;
        alloc_status[i].objects_allocated = 0;
        alloc_status[i].objects_count = 0;
        alloc_status[i].bytes_allocated = 0;
        alloc_status[i].no_allocated_together = 1;
        alloc_status[i].name_of_type = "unused";
    }
}
```

The function start_memory is called from 1/main.

§**3. Architecture.** The memory manager is built in three levels, with its interface to the rest of `cblorb` being entirely at level 3 (except that when it shuts down it calls a level 1 routine to free everything). Each level uses the one below it.

(3) Managing linked lists of large objects, within which objects can be created at any point, and from which objects can be deleted; and providing a way to create new small objects of any given type.
(2) Allocating some thousands of memory frames, each holding one large object or an array of small objects.
(1) Allocating and freeing a few dozen large blocks of contiguous memory.

§**4. Level 1: memory blocks.** Memory is allocated in blocks of 100K, within which objects are allocated as needed. The "safety margin" is the number of spare bytes left blank at the end of each object: this is done because we want to be paranoid about compilers on different architectures aligning structures to different boundaries (multiples of 4, 8, 16, etc.). Each block also ends with a firebreak of zeroes, which ought never to be touched: we want to minimise the chance of a mistake causing a memory exception which crashes the compiler, because if that happens it will be difficult to recover the circumstances from the debugging log.

```
define SAFETY_MARGIN 64
define BLANK_END_SIZE 128
```

§**5.** At present `MEMORY_GRANULARITY` is 100K. This is the quantity of memory allocated by each individual `malloc` call.

After `MAX_BLOCKS_ALLOWED` blocks, we throw in the towel: we must have fallen into an endless loop which creates endless new objects somewhere. (If this ever happens, it would be a bug: the point of this mechanism is to be able to recover. Without this safety measure, OS X in particular would grind slowly to a halt, never refusing a `malloc`, until the user was unable to get the GUI responsive enough to kill the process.)

```
define MAX_BLOCKS_ALLOWED 15000
define MEMORY_GRANULARITY 100*1024*4          which must be divisible by 1024

int no_blocks_allocated = 0;
int total_objects_allocated = 0;              a much larger number, used only for the debugging log
```

§**6.** Memory blocks are stored in a linked list, and we keep track of the size of the current block: that is, the block at the tail of the list. Each memory block consists of a header structure, followed by `SAFETY_MARGIN` null bytes, followed by actual data.

```
typedef struct memblock_header {
    int block_number;
    struct memblock_header *next;
    char *the_memory;
} memblock_header;

memblock_header *first_memblock_header = NULL;          head of list of memory blocks
memblock_header *current_memblock_header = NULL;        tail of list of memory blocks
int used_in_current_memblock = 0;                       number of bytes so far used in the tail memory block
```

The structure memblock_header is private to this section.

§**7.** The actual allocation and deallocation is performed by the following pair of routines.

```
void allocate_another_block(void) {
    unsigned char *cp;
    memblock_header *mh;
    ⟨Allocate and zero out a block of memory, making cp point to it 8⟩;
    mh = (memblock_header *) cp;
    used_in_current_memblock = sizeof(memblock_header) + SAFETY_MARGIN;
    mh->the_memory = (void *) (cp + used_in_current_memblock);
    ⟨Add new block to the tail of the list of memory blocks 9⟩;
}
```

The function allocate_another_block is.

§**8.** Note that `cp` and `mh` are set to the same value: they merely have different pointer types as far as the C compiler is concerned.

⟨Allocate and zero out a block of memory, making cp point to it 8⟩ ≡

```
int i;
if (no_blocks_allocated++ >= MAX_BLOCKS_ALLOWED)
    fatal(
        "the memory manager has halted cblorb, which seems to be generating "
        "endless structures. Presumably it is trapped in a loop");
check_memory_integrity();
cp = (unsigned char *) (malloc(MEMORY_GRANULARITY));
if (cp == NULL) fatal("Run out of memory: malloc failed");
for (i=0; i<MEMORY_GRANULARITY; i++) cp[i] = 0;
```

This code is used in §7.

§**9.** As can be seen, memory block numbers count upwards from 0 in order of their allocation.

⟨Add new block to the tail of the list of memory blocks 9⟩ ≡

```
if (current_memblock_header == NULL) {
    mh->block_number = 0;
    first_memblock_header = mh;
} else {
    mh->block_number = current_memblock_header->block_number + 1;
    current_memblock_header->next = mh;
}
current_memblock_header = mh;
```

This code is used in §7.

§**10.** Freeing all this memory again is just a matter of freeing each block in turn, but of course being careful to avoid following links in a just-freed block.

```
void free_memory(void) {
    memblock_header *mh = first_memblock_header;
    while (mh != NULL) {
        memblock_header *next_mh = mh->next;
        void *p = (void *) mh;
        free(p);
        mh = next_mh;
    }
}
```

The function free_memory is called from 1/main.

§**11. Level 2: memory frames and integrity checking.** Within these extensive blocks of contiguous memory, we place the actual objects in between "memory frames", which are only used at present to police the integrity of memory: again, finding obscure and irritating memory-corruption bugs is more important to us than saving bytes. Each memory frame wraps either a single large object, or a single array of small objects.

```
define INTEGRITY_NUMBER 0x12345678                    a value unlikely to be in memory just by chance

typedef struct memory_frame {
    int integrity_check;                              this should always contain the INTEGRITY_NUMBER
    struct memory_frame *next_frame;                  next frame in the list of memory frames
    int mem_type;                                     type of object stored in this frame
    int allocation_id;                                allocation ID number of object stored in this frame
} memory_frame;
```

The structure memory_frame is private to this section.

§**12.** There is a single linked list of all the memory frames, perhaps of about 10000 entries in length, beginning here. (These frames live in different memory blocks, but we don't need to worry about that.)

```
memory_frame *first_memory_frame = NULL;              earliest memory frame ever allocated
memory_frame *last_memory_frame = NULL;               most recent memory frame allocated
```

§**13.** If the integrity numbers of every frame are still intact, then it is pretty unlikely that any bug has caused memory to overwrite one frame into another. `check_memory_integrity` might on very large runs be run often, if we didn't prevent this: since the number of calls would be roughly proportional to memory usage, we would implicitly have an $O(n^2)$ running time in the amount of storage n allocated.

```
int calls_to_cmi = 0;
void check_memory_integrity(void) {
    int c;
    memory_frame *mf;
    c = calls_to_cmi++;
    if (!((c<10) || (c == 100) || (c == 1000) || (c == 10000))) return;
    for (c = 0, mf = first_memory_frame; mf; c++, mf = mf->next_frame)
        if (mf->integrity_check != INTEGRITY_NUMBER)
            fatal("Memory manager failed integrity check");
}
void debug_memory_frames(int from, int to) {
    int c;
    memory_frame *mf;
    for (c = 0, mf = first_memory_frame; (mf) && (c <= to); c++, mf = mf->next_frame)
        if (c >= from) {
            char *desc = "corrupt";
            if (mf->integrity_check == INTEGRITY_NUMBER)
                desc = alloc_status[mf->mem_type].name_of_type;
        }
}
```

The function check_memory_integrity is.
The function debug_memory_frames is.

§**14.** We have seen how memory is allocated in large blocks, and that a linked list of memory frames will live inside those blocks; we have seen how the list is checked for integrity; but we not seen how it is built. Every memory frame is created by the following function:

```
void *allocate_mem(int mem_type, int extent) {
    unsigned char *cp;
    memory_frame *mf;
    int bytes_free_in_current_memblock, extent_without_overheads = extent;
    extent += sizeof(memory_frame);                    each allocation is preceded by a memory frame
    extent += SAFETY_MARGIN;                  each allocation is followed by SAFETY_MARGIN null bytes
    ⟨Ensure that the current memory block has room for this many bytes 15⟩;
    cp = ((unsigned char *) (current_memblock_header->the_memory)) + used_in_current_memblock;
    used_in_current_memblock += extent;
    mf = (memory_frame *) cp;                                              the new memory frame,
    cp = cp + sizeof(memory_frame);                        following which is the actual allocated data
    mf->integrity_check = INTEGRITY_NUMBER;
    mf->allocation_id = alloc_status[mem_type].objects_allocated;
    mf->mem_type = mem_type;
    ⟨Add the new memory frame to the big linked list of all frames 16⟩;
    ⟨Update the allocation status for this type of object 17⟩;
    total_objects_allocated++;
    return (void *) cp;
}
```

The function allocate_mem is.

§**15.** The granularity error below will be triggered the first time a particular object type is allocated. So this is not a potential time-bomb just waiting for a user with a particularly long and involved source text to discover.

⟨Ensure that the current memory block has room for this many bytes 15⟩ ≡

```
if (current_memblock_header == NULL) allocate_another_block();
bytes_free_in_current_memblock = MEMORY_GRANULARITY - (used_in_current_memblock + extent);
if (bytes_free_in_current_memblock < BLANK_END_SIZE) {
    allocate_another_block();
    if (extent+BLANK_END_SIZE >= MEMORY_GRANULARITY)
        fatal("Memory manager failed because granularity too low");
}
```

This code is used in §14.

§**16.** New memory frames are added to the tail of the list:

⟨Add the new memory frame to the big linked list of all frames 16⟩ ≡

```
mf->next_frame = NULL;
if (first_memory_frame == NULL) first_memory_frame = mf;
else last_memory_frame->next_frame = mf;
last_memory_frame = mf;
```

This code is used in §14.

§**17.** See the definition of `alloc_status` above.

⟨Update the allocation status for this type of object 17⟩ ≡

```
if (alloc_status[mem_type].first_in_memory == NULL)
    alloc_status[mem_type].first_in_memory = (void *) cp;
alloc_status[mem_type].last_in_memory = (void *) cp;
alloc_status[mem_type].objects_allocated++;
alloc_status[mem_type].bytes_allocated += extent_without_overheads;
```

This code is used in §14.

§**18. Level 3: managing linked lists of allocated objects.** We define macros which look as if they are functions, but for which one argument is the name of a type: expanding these macros provides suitable C functions to handle each possible type. These macros provide the interface through which all other sections of `cblorb` allocate and leaf through memory.

Note that `inweb` allows multi-line macro definitions without backslashes to continue them, unlike ordinary C. Otherwise these are "standard" macros, though this was my first brush with the `##` concatenation operator: basically `CREATE(thing)` expands into `(allocate_thing())` because of the `##`. (See Kernighan and Ritchie, section 4.11.2.)

```
define CREATE(type_name) (allocate_##type_name())
define CREATE_BEFORE(existing, type_name) (allocate_##type_name##_before(existing))
define DESTROY(this, type_name) (deallocate_##type_name(this))
define FIRST_OBJECT(type_name) ((type_name *) alloc_status[type_name##_MT].first_in_memory)
define LAST_OBJECT(type_name) ((type_name *) alloc_status[type_name##_MT].last_in_memory)
define NEXT_OBJECT(this, type_name) ((type_name *) (this->next_structure))
define PREV_OBJECT(this, type_name) ((type_name *) (this->prev_structure))
define NUMBER_CREATED(type_name) (alloc_status[type_name##_MT].objects_count)
```

§**19.** The following macros are widely used (well, the first one is, anyway) for looking through the double linked list of existing objects of a given type.

```
define LOOP_OVER(var, type_name)
    for (var=FIRST_OBJECT(type_name); var != NULL; var = NEXT_OBJECT(var, type_name))
define LOOP_BACKWARDS_OVER(var, type_name)
    for (var=LAST_OBJECT(type_name); var != NULL; var = PREV_OBJECT(var, type_name))
```

§**20. Allocator functions created by macros.** The following macros generate a family of systematically named functions. For instance, we shall shortly expand `ALLOCATE_INDIVIDUALLY(parse_node)`, which will expand to three functions: `allocate_parse_node`, `deallocate_parse_node` and `allocate_parse_node_before`.

Quaintly, `#type_name` expands into the value of `type_name` put within double-quotes.

```
define NEW_OBJECT(type_name) ((type_name *) allocate_mem(type_name##_MT, sizeof(type_name)))
define ALLOCATE_INDIVIDUALLY(type_name)
type_name *allocate_##type_name(void) {
    alloc_status[type_name##_MT].name_of_type = #type_name;
    type_name *prev_obj = LAST_OBJECT(type_name);
    type_name *new_obj = NEW_OBJECT(type_name);
    new_obj->allocation_id = alloc_status[type_name##_MT].objects_allocated-1;
    new_obj->next_structure = NULL;
    if (prev_obj != NULL)
        prev_obj->next_structure = (void *) new_obj;
    new_obj->prev_structure = prev_obj;
    alloc_status[type_name##_MT].objects_count++;
    return new_obj;
}
void deallocate_##type_name(type_name *kill_me) {
    type_name *prev_obj = PREV_OBJECT(kill_me, type_name);
    type_name *next_obj = NEXT_OBJECT(kill_me, type_name);
    if (prev_obj == NULL) {
        alloc_status[type_name##_MT].first_in_memory = next_obj;
    } else {
        prev_obj->next_structure = next_obj;
    }
    if (next_obj == NULL) {
        alloc_status[type_name##_MT].last_in_memory = prev_obj;
    } else {
        next_obj->prev_structure = prev_obj;
    }
    alloc_status[type_name##_MT].objects_count--;
}
type_name *allocate_##type_name##_before(type_name *existing) {
    type_name *new_obj = allocate_##type_name();
    deallocate_##type_name(new_obj);
    new_obj->prev_structure = existing->prev_structure;
    if (existing->prev_structure != NULL)
        ((type_name *) existing->prev_structure)->next_structure = new_obj;
    else alloc_status[type_name##_MT].first_in_memory = (void *) new_obj;
    new_obj->next_structure = existing;
    existing->prev_structure = new_obj;
    alloc_status[type_name##_MT].objects_count++;
    return new_obj;
}
```

§**21.** `ALLOCATE_IN_ARRAYS` is still more obfuscated. When we `ALLOCATE_IN_ARRAYS(X, 100)`, the result will be definitions of a new type `X_block` and functions `allocate_X`, `allocate_X_block`, `deallocate_X_block` and `allocate_X_block_before` (though the last is not destined ever to be used). Note that we are not provided with the means to deallocate individual objects this time: that's the trade-off for allocating in blocks.

```
define ALLOCATE_IN_ARRAYS(type_name, NO_TO_ALLOCATE_TOGETHER)
typedef struct type_name##_array {
    int used;
    struct type_name array[NO_TO_ALLOCATE_TOGETHER];
    MEMORY_MANAGEMENT
} type_name##_array;
ALLOCATE_INDIVIDUALLY(type_name##_array)
type_name##_array *next_##type_name##_array = NULL;
struct type_name *allocate_##type_name(void) {
    if ((next_##type_name##_array == NULL) ||
        (next_##type_name##_array->used >= NO_TO_ALLOCATE_TOGETHER)) {
        alloc_status[type_name##_array_MT].no_allocated_together = NO_TO_ALLOCATE_TOGETHER;
        next_##type_name##_array = allocate_##type_name##_array();
        next_##type_name##_array->used = 0;
    }
    return &(next_##type_name##_array->array[
        next_##type_name##_array->used++]);
}
```

The structure type_name##_array is private to this section.

§**22. Expanding many macros.** Each given structure must have a typedef name, say `marvel`, and can be used in one of two ways. Either way, we can obtain a new one with the macro `CREATE(marvel)`.

Either (a) it will be individually allocated. In this case `marvel_MT` should be defined with a new MT (memory type) number, and the macro `ALLOCATE_INDIVIDUALLY(marvel)` should be expanded. The first and last objects created will be `FIRST_OBJECT(marvel)` and `LAST_OBJECT(marvel)`, and we can proceed either way through a double linked list of them with `PREV_OBJECT(mv, marvel)` and `NEXT_OBJECT(mv, marvel)`. For convenience, we can loop through marvels, in creation order, using `LOOP_OVER(var, marvel)`, which expands to a `for` loop in which the variable `var` runs through each created marvel in turn; or equally we can run backwards through using `LOOP_BACKWARDS_OVER(var, marvel)`. In addition, there are corruption checks to protect the memory from overrunning accidents, and the structure can be used as a value in the symbols table. Good for large structures with significant semantic content.

Or (b) it will be allocated in arrays. Once again we can obtain new marvels with `CREATE(marvel)`. This is more efficient both in speed and memory usage, but we lose the ability to loop through the objects. For this arrangement, define `marvel_array_MT` with a new MT number and expand the macro `ALLOCATE_IN_ARRAYS(marvel, 100)`, where 100 (or what may you) is the number of objects allocated jointly as a block. Good for small structures used in the lower levels.

Here goes, then.

```
ALLOCATE_INDIVIDUALLY(auxiliary_file)
ALLOCATE_INDIVIDUALLY(skein_node)
ALLOCATE_INDIVIDUALLY(chunk_metadata)
ALLOCATE_INDIVIDUALLY(placeholder)
ALLOCATE_INDIVIDUALLY(heading)
ALLOCATE_INDIVIDUALLY(table)
ALLOCATE_INDIVIDUALLY(segment)
ALLOCATE_INDIVIDUALLY(request)
ALLOCATE_INDIVIDUALLY(template)
ALLOCATE_INDIVIDUALLY(template_path)
```