Dynamic memory allocation

In this course, we have discussed how a call to

```
int *a_data{ new int[20] };
```

makes a request to the operating system for 20 × 4 bytes = 80 bytes of memory. This memory could be almost anywhere in main memory: the operating system sets aside a significant block of memory available for such requests. This is in contrast to the call stack, where the memory for the next function call sits right on top of the memory for the current function call. Each executing program has a block of memory allocated to it for the call stack, and this is often fixed in size, especially for embedded systems. At the same time, however, the operating system must respond to requests for dynamically allocated memory from all executing programs, and the operating system must flag that memory as allocated to that specific executing program, and at any time, an executing program may free up (or *deallocate*) that memory, in which case, the operating system needs to flag that memory as once again available for future requests.

In this high-level discussion on this, we will introduce a very simple design that can be used to manage the allocation of memory dynamically. This is not a scheme that would be used in any general-purpose operating system such as Linux, OS X or Windows, but it, or variations thereupon, may be useful in a small embedded system.

Design

We will suppose that all the memory that is available for dynamic memory allocation starts as a single contiguous block of N bytes. Like an array, we can refer to this block by the addresses of the first byte. To simplify the design, when a requests comes in for n bytes, we will allocate a block of $n + 8$ bytes, where the extra eight bytes are used to store the size of the request. Additionally, we will round $n + 8$ up to the next-highest multiple of 16 bytes. This can be done with a simple program as follows:

```
std::size_t make_multiple_of_four( std::size_t request ) {
    assert( request != 0 );

    request += 0b1000;          // Add 8 bytes
    std::size_t last_four_bits{ request & 0b1111 };

    if ( last_four_bits == 0 ) {
        // It is already a multiple of 16
        return request;
    } else {
        // Round up to the next highest multiple of 16
        return (request & ~0b1111) + 0b10000;
    }
}
```

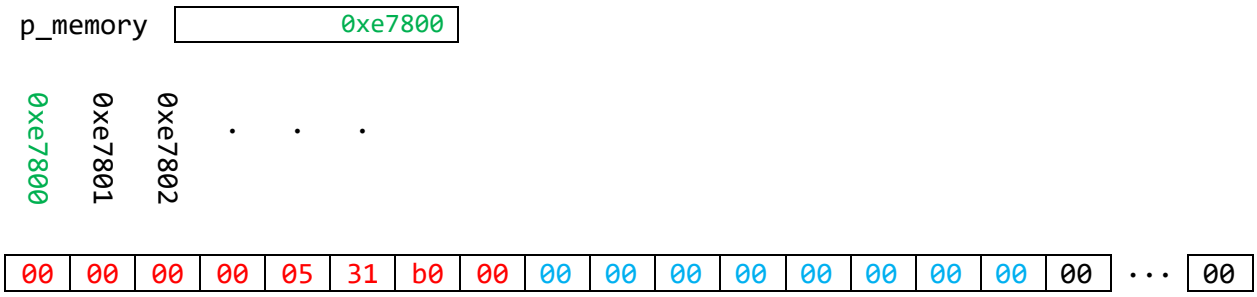
You will recall that we can fully describe an array by storing the address of the first byte of the array together with the number of entries *n* and the type `typename`. We can get the memory allocated by calculating `n*sizeof( typename )`. For our design here, we will start with two variables, one that stores the address of the first byte, and the second that stores the total number of bytes of memory:

```
void *p_memory{ 0xe7800 };
```

We could have a second variable that stores the number of bytes of memory that is available, but as this entire block is memory that is currently not being used, why not just use the first eight bytes to store the size of the block? Additionally, we will use the next eight bytes to store a pointer, but we will start with that pointer being the null pointer (all zeros). We will now look at an example of such a set-up and the describe how we can use this for dynamic memory allocation.

Example

Let us assume that there are 85,100 KiB of memory available, so 85,100 × 1024 = 87142400 bytes (= `0x531b000` bytes), and the first byte of this is at address `0xe7800`. Thus, starting at this address, we would see the following bytes (remember, one byte is two hexadecimal characters):



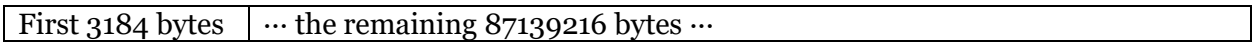
The exact details about how we can arbitrarily interpret the first eight bytes as a size, and the next eight bytes as an address is a little beyond this course. (In short, it requires a `struct` and casting, and is used in the accompanying program.)

A first request

Suppose a request for 3168 bytes is made; for example, a program executed:

```
double *a_data{ new double[396] };   // 396 x 8 = 3168
```

After adding eight bytes to this request and rounding it up to the next multiple of 16, we have a request for 3184 bytes of memory. We look at the memory available in the first block of memory, and notice that there is more than enough memory to satisfy this request. Thus, we will split the 87142400 bytes into one block that 3184 bytes and the rest which is the remaining 87139216 bytes:

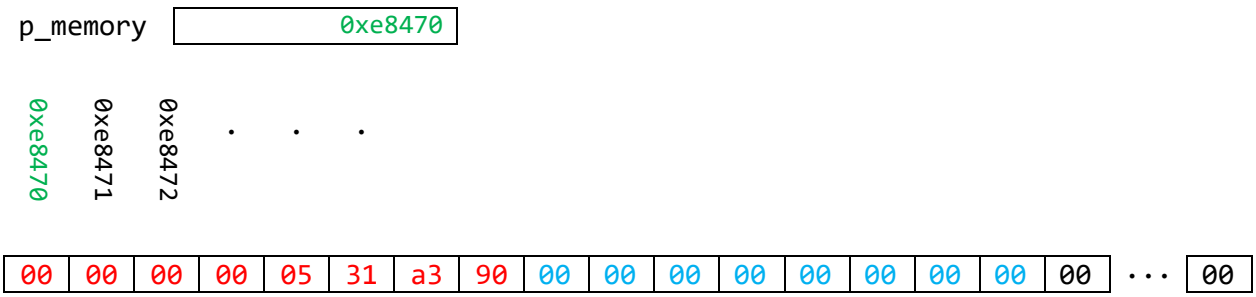


We will modify the 3184 byte block as follows:

1. The size of this block will be stored in the first eight bytes, so 3184 = `0xc70`.
2. We will return the address of the byte immediately following these eight bytes. Thus, the user will have access to the next 3168 bytes needed for the array.

The address that will be returned is `0xe7808`, so `a_data` will be assigned this address, and any access to `a_data[k]` will access memory in these next 3168 bytes.

As for the remaining block of 87139216 bytes = `0x531a390` bytes, we will store in the first eight bytes of that block how much memory is left, and we will update `p_memory` to store the first address of that block:



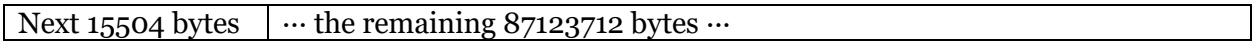
Thus, the block of free memory is now smaller, and because the first 3184 bytes were allocated, the address of this block is now `0xe7800` (the original address) plus 3184 (= `0xc70`) = `0xe8470`. In this design, we do not track the memory that was allocated; instead, we assume that the executing program will return that memory when it is no longer required. This may be a foolish assumption in a general purpose multi-user environment; however, it may be quite appropriate for an embedded system where all code is appropriately designed and reviewed.

A second request

Suppose a second request comes in for 30992 bytes comes in; for example, someone requested:

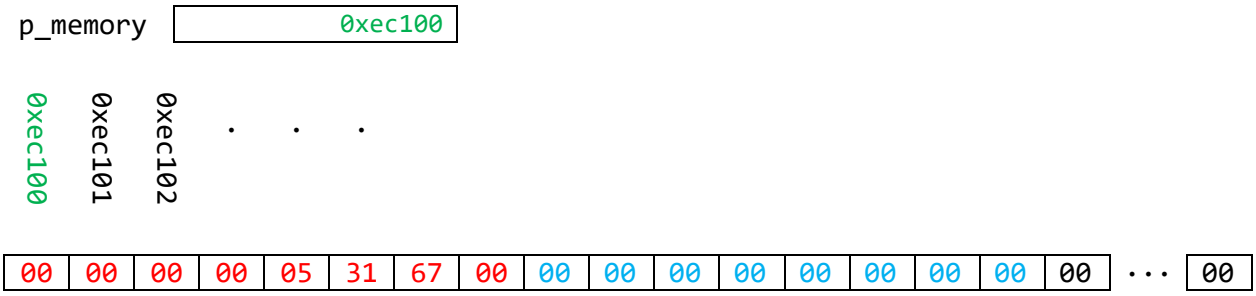
```
a_readings = new double[1937];   // 1937 x 8 = 15496
```

After adding 8 bytes to this request, it is now a multiple of 16, so the request is for a block of 15504 bytes. We look at the first block of memory, and notice that there is still more than enough memory to satisfy this request. Thus, we will split the 87139216 bytes into one block that 15504 bytes (= `0x3c90` bytes) and the rest which is the remaining 87123712 bytes:



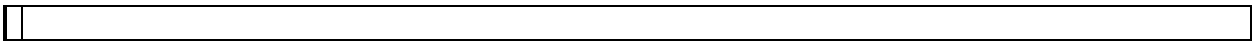
The size `0x3c90` will be stored in the first eight bytes (starting at address `0xe8470`), and the address that will be returned is `0xe8478`, so `a_readings` will be assigned this address, and any access to `a_readings[k]` will access memory in these next 15496 bytes.

As for the remaining block of 87123712 bytes, we will store in the first eight bytes of that block how much memory is left, and we will update `p_memory` to store the first address of that block:



Thus, once again, the block of free or available memory is now smaller.

To give an overview, our initial block of memory looks a little like:



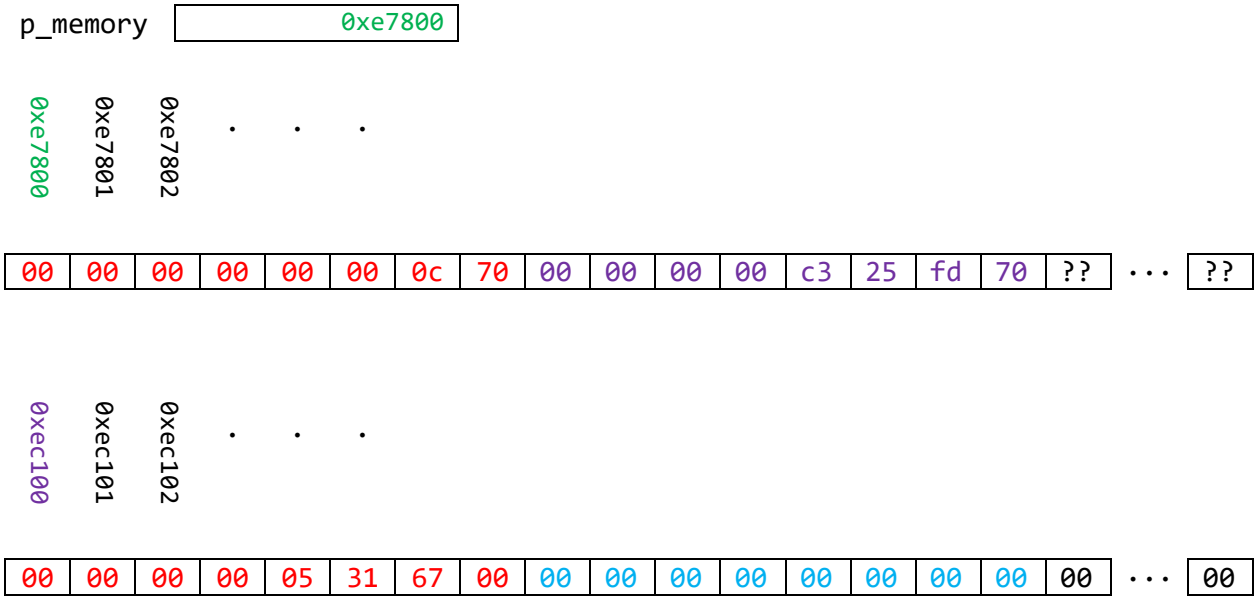
where the first sliver was allocated to the first request, the second to the second request, and all that is left over is available for future requests.

De-allocating or freeing memory

Suppose the first request now frees the memory that was allocated it, and in C++, this would be through a call to `delete[] a_data`. We now have to record that there are two blocks of memory that are available: a smaller block that is 3184 bytes, and a larger block that is 87123712 bytes.

In order to allow us to store this memory, we will use a *linked list*. First, recall that the size of the block is in the eight bytes prior to the address that was returned. When a program calls `delete[]`, it is necessary to subtract eight from that address to get the start of the block, where the first eight bytes store the size. Remember that second eight bytes that we set to all zeros (the null pointer)? We do the following:

1. We will leave the current block alone.
2. With the memory that was freed, we will now use it again as follows:
    a. The first eight bytes will store how much memory is available in that block, and 3168 bytes = `0xc60` bytes.
    b. The next eight blocks will store the address of the larger block of memory that is still available.



Strategies

Thus, each block of available memory stores how much memory is available in that block, and the address of any subsequent block of available memory. When a future request comes in for memory, we can use the following algorithm, we will proceed as follows:

1. Look for an unallocated block that is equal to or larger than the request for memory.
2. If the block is equal to what is requested, just update the pointers to skip this block, and return the address of the block.
3. If the block is smaller than what is requested, split the block into two, return the address of the first, which is of the size requested, and replace that block with what is left over.

When a block is freed, it can simply be inserted at the front of the linked list.

Thus, after a few allocations and deallocations, the available memory map may look as shown in Figure 1.

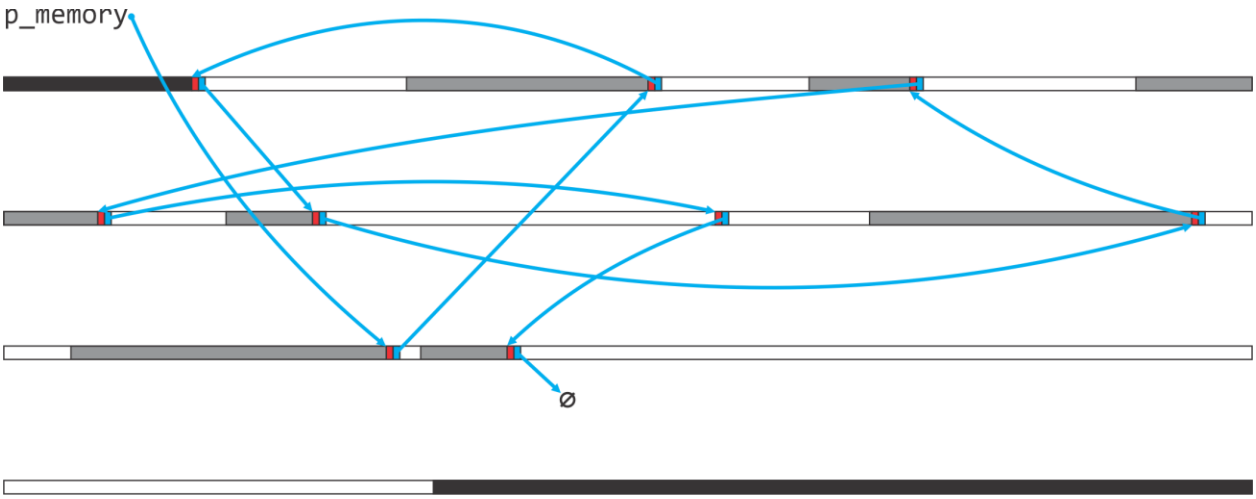


Figure 1. An example of a possible memory map.

Any memory not available for dynamic memory allocation is shown in black. All memory that has been allocated in Figure 1 is grayed out. At the start of each available block (shown in white), there are eight bytes allocated for sorting the size of the block (shown in red) and the next eight bytes store the address of the next available block in the linked list (shown in blue). The arrows graphically show the address that is stored in those eight bytes by pointing to the address that is stored there. You will notice that two of the available blocks are juxtaposed; this may have been a result of a block of memory being split but then later having one of those blocks freed. The `p_memory` variable stores the address of what is likely the last deallocated block of memory, and that block of memory stores its size and the next block of memory, etc. By design, the last block of memory in this linked lists of available memory blocks is the last, and its next pointer stores the null pointer, or all zeros.

What is convenient about this design is that the available memory is used to store information about the available memory, so the only additional memory is that of the variable `p_memory`. There are also strategies that can be used when a new request for memory arrives:

1. First fit, where we walk through the linked list until we find a block that is greater than or equal in size to the request.
2. Best fit, where we find that block that is smallest in size while still being greater than or equal to the request.
3. Worst fit, where, unless an exact match is found for the request, it is the largest block that is used to satisfy the request.

For example, if we were to use the first-fit algorithm, then if a request came in for more memory than was available in the first block, but exactly the size of the second block, we would designate that second block as allocated, and we would just remove that block from the linked list, as shown in Figure 2 (you should contrast this with Figure 1).

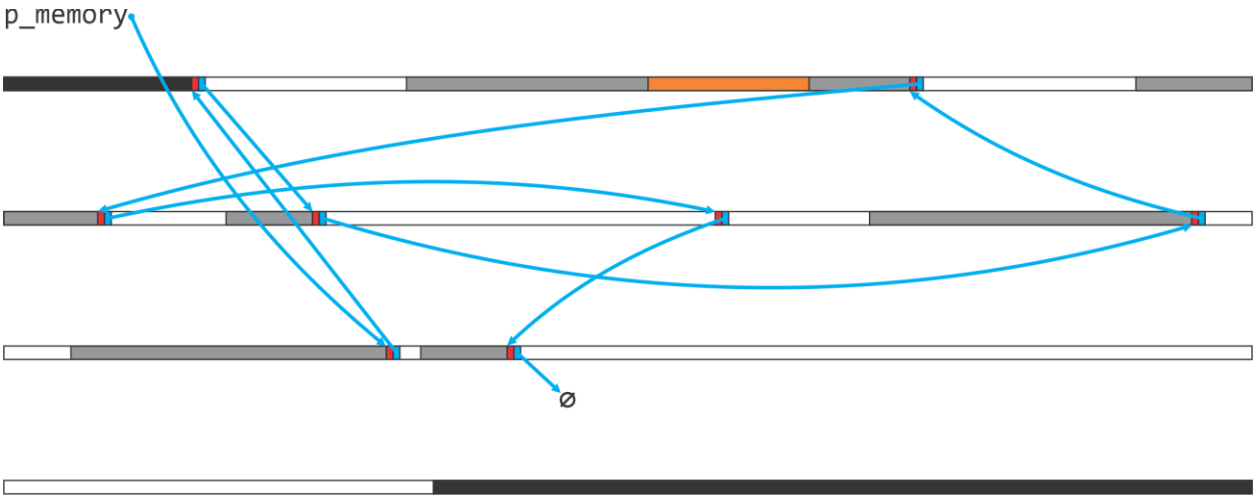


Figure 2. The change if the second block perfectly matches the next request.

Alternatively, if the next request was larger than that first block, but less than the second, we could split the second, and what appears in Figure 3 as orange, that would be the allocated block, and the balance would remain in the linked list of blocks of available memory.

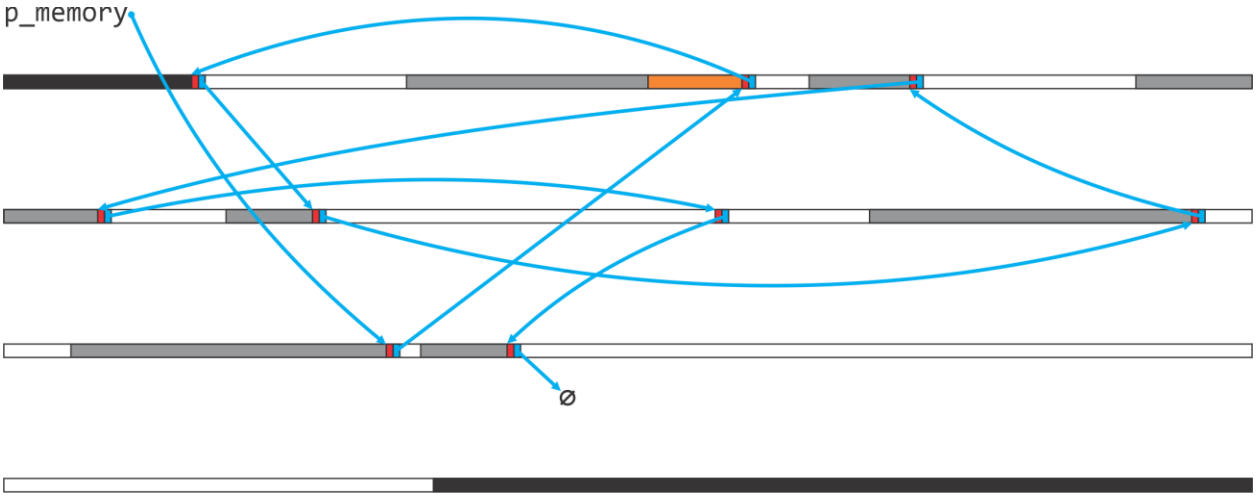


Figure 2. The change if the second block perfectly matches the next request.

This approach, however, has some weaknesses and there are some aspects we don't discuss (e.g., how does the operating system know how much memory was allocated when an address is returned and marked as available?), and there are other better designs available. However, as a quick introduction to how memory management *could* be implemented (and there are circumstances where the above design is appropriate), once you the follow-on course in algorithms and data structures, you'll be able to appreciate some of the more modern designs.

Implementation

An implementation of the first-fit memory allocation algorithm is shown at

https://replit.com/@dwharder/2-Dynamic-memory-allocation.

You are welcome to execute and even use it. The map() member function has an ASCII diagram of you have seen above as Figures 1 through 3.