Programming Assignment #1 FAQ

Here are some popular Questions and Answers for assignment #1.

1. **Question: Can I modify the private section of the header file?**
   Answer: Yes. In fact, you *must* modify it. You can put whatever you want in the private section. This includes fields and methods. (A lot of functionality is usually placed into the private methods.) However, you must not touch the public section at all (and there is no reason you should need to). I gave you the header file so you could see the interface that the client expects. Unfortunately, C++ does not (currently) allow you to have 2 header files for a single class.

   Also, as stated in the handout, you are not allowed to put any implementation in the header file. All implementation goes in the .cpp file. Repeat: **NO IMPLEMENTATION, HOWEVER SLIGHT, IS ALLOWED IN THE HEADER FILE.** The exception to this is the implementation that I've already provided.

2. **Question: How should we calculate the page size?**
   Answer: You should return the size of the entire page. For example, if you had a page with 5 objects, each object 20 bytes, the actual size of the page is 104 bytes. (Number of objects) * (Size of object) + (Size of link pointer) = 5×20 + 4. If you have padding, header blocks, and/or alignment bytes, you'd have to include those as well. The 4 bytes for the pointer is for a 32-bit program. If it is a 64-bit program, it will be 8 bytes.

   You should also look at the sample client code and output which demonstrates a client interacting with the ObjectAllocator's interface.

3. **Question: How should we structure the free list? Normally I would just do it my own way, but the GetFreeList function implies that it should be in some specific format.**
   Answer: This is a good point. "Normally", we don't want to expose any of the internal properties of our class because we want to be able to change them at will without affecting any of the clients. This is the reason that the *GetFreeList* method is listed under the comment "Testing/Debugging methods". It is meant to be used to debug the class from outside. There are other approaches to this such as making it protected and then creating a derived class whose sole purpose is to debug the implementation. Or we could make public iterators so that clients can request the "next" object in the list. It turns out that this unnecessarily complicates an otherwise "simple" class. (Plus, the primary purpose of this assignment is to understand the memory-manager concept.) Since both the free list and page list are implemented as simple, single-linked lists, it is trivial for everyone to implement them the same way.

   The sample driver gets the pointer to the free list so it can dump out an object and see that the correct debugging information is present (predefined values, padding, header blocks). The *GetPageList* method is similar in that it allows the driver to dump the entire set of pages to examine their values. The GUI driver uses the page list so it can display a "memory dump" of your allocated pages in real time.

4. **Question: What are the specific text strings you want to have thrown? Since you seem to be intending on sending those straight to the output, and that's one of the things you grade on.**
   Answer: The text strings should be informative. This means if there is no memory available, you should say why. (e.g. because the system is out of memory or because the maximum page count has been reached.)

   The driver that I use to grade the assignments (automatically using a diff program) won't print out your text strings simply because it is very difficult to get everyone to emit the same strings. (I've tried!) If you look at the sample driver, you'll see that there is a `#define` named `SHOW_EXCEPTIONS` which is used in order to show the error messages that your code emits. If this is not defined, generic error messages will be displayed by the driver instead.

5. **Question: How are you grading this assignment? In other words, since the implementation is private, will we get full credit if we get the same output as your example driver?**
   Answer: Obviously, your object allocator needs to work correctly to get full credit. But, like you said, the implementation is private so there is no one "right" implementation. However, some implementations are better or worse than others. Think in terms of complexity and memory usage. For example, if one implementation runs slower or uses more memory than another, it is less optimal. For this assignment, the scope of the memory manager's functionality is purposely limited so that you can focus on implementing it using the linked list technique shown in class. This will give you adequate practice with linked lists (for something other than the sake of linked lists) and very efficient memory usage when implementing your own memory manager. There are certainly many other ways to implement the private portions. If you choose to invent your own algorithms, you need to ask yourself "Given the functionality of the Object Allocator, is this superior to the one we discussed in class?"

   As it turns out, for memory efficiency, the technique shown in the diagrams is just about "As Good As It Gets". ™

6. **Question: Do we need to handle objects smaller than 4 bytes (which is the size of a 32-bit pointer)?**
   Answer: No. In fact, you shouldn't attempt to do so. Managing such small objects will lead to poor performance. In the real world, we wouldn't dynamically allocate individual chunks like that. (We'd probably use an array of small chunks. However, our array could be allocated by the Object Allocator, so we still might benefit.)

   Because we require a linked list to track each block of memory (object) that the client requests, we need at least 4 bytes (32 bit pointers) for the "next" pointer in each block. The linked list benefits from not having to allocate its own storage for the next pointer, which would have made our memory manager less efficient. Even if we chose, say, to keep an array of pointers to the blocks (a poor strategy for this assignment), we still would have to consume extra memory to hold these pointers. To see why it's not a good approach, compare the complexity for memory usage using an

array to track the blocks with the complexity for memory usage using the approach outlined in the diagrams.

Again, in the big picture, the limitation that each object must be at least 4 bytes in size is not really a limitation at all.

Note: In a 64-bit program, the smallest object you would be able to deal with is 64 bits (8 bytes). It is important that you **DO NOT** hard code the value 32 or 64 anywhere in your code. Use the expression **`: sizeof(void *)`** instead. This will allow your code to work correctly in both 32-bit and 64-bit environments (or anything else that might come along in the future). It will also prevent you from receiving a lower grade on the assignment!

7. **Question: What kinds of checks are enabled when the `DebugOn` flag is set?**
   Answer: There are two areas that are controlled by this flag: Patterning the memory (writing a specific signature) and checking addresses/pad bytes when the client calls Free.
   1. Setting the memory blocks to a specific pattern (signature) is only done when `DebugOn` is true. This includes writing the pattern on the pad bytes as well.
   2. Checking the address when the client frees a block is done only when `DebugOn` is true. This includes checking for bad boundaries and multiple frees.

   Note that adding pad bytes is not part of the debug routine (it's controlled by the value of the PadBytes configuration parameter). However, writing a signature to those pad bytes is a debug task.

8. **Question: Can we use `memset` to initialize the bytes with signatures?**
   Answer: Yes, you should. This is a situation where you don't want to write your own **`memset`** function or use your own loop to set each byte of memory. Not using **`memset`** will cause you to lose points. Note that **`memset`** is in `string.h` (or `cstring` for the C++ interface).

9. **Question: There seems to be 4 different *modes* for the header blocks functionality. How do they differ?**
   Answer: There are 4 modes: none, basic, extended, and external. There is an enumeration in the OAConfig class that lists them:

```
enum HBLOCK_TYPE{hbNone, hbBasic, hbExtended, hbExternal};
```

   Descriptions:
   1. **None** - There are no header blocks, meaning this feature is disabled.
   2. **Basic** - A basic header is always 5 bytes in length. The first 4 bytes (from left to right) are the allocation number of this block. The 5th (right-most) byte is a *flags* byte and is used to track various attributes of the memory block.

      The allocation counter is kept internally by the object allocator. It is incremented each time the allocator successfully returns a block of memory to the client. At start, the counter is 0, and continues to increment for the life of the allocator. Currently, the counter is a 4-byte

unsigned value, meaning we can track over 4 billion allocations. We won't come anywhere near that in this class!

Currently, only one flag is used and the other 7 bits are reserved. The right-most bit of the flag byte indicates whether the memory block is free, or is in-use. A value of 0 means the block is free (**00000000**), and a value of 1 indicates the block is currently in-use (**00000001**).

3. **Extended** - An extended header is similar to a basic header, but adds two more fields: a 2-byte use-counter and a user-defined field of any number of bytes.

   The 2-byte use-counter is used to count *how many times this block has been used*. This means, that each time the Object Allocator provides a block to the client, this counter is incremented. When a page is first allocated, the use-counter and flag byte are set to 0. The first time the block is given to the client, the value of the counter is 1. When the client later frees the block, the counter is kept with the free block. When the client asks for a block of memory and the Object Allocator gives back a previously used (but now, free) block, the use-counter is incremented. See the sample driver and sample output for more information.

   The user must specify how many additional bytes are to be reserved in the header. This is supplied to the ObjectAllocator's constructor. Currently, the values of these bytes are 0 and must remain 0. See the driver for details.

   So, from left to right, the extended header might look like this in hexadecimal (assuming 5 additional user-defined bytes, for a total size of 12 bytes):

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
|          A           | |   B   | |        C          ||D |
```

   A is the user-defined bytes, B is the use-counter, C is the allocation number, and D is the flags byte. An actual header would have values other than just 0x00 for each byte of the header.

4. **External** - An external header is different from the others in that it is simply a pointer to a chunk of memory outside of the block itself. Hence the term, *external*. The size of the header block will depend on the size of a pointer. It will be 4 bytes for a 32-bit computer and 8 bytes for a 64-bit computer. This pointer is stored in the header block and points to the information about this block of memory. Many real-world memory managers use this type of header because it is much easier to extend and keeps all header blocks the same size, regardless of the feature set.

   At the time a block of memory is requested by the client, you will allocate (using **new**, not the ObjectAllocator) memory for the external header. The external header currently has 3 fields and looks like this:

```
struct MemBlockInfo
{
  bool in_use;     // Is the block free or in use?
  char *label;  // A dynamically allocated NUL-terminated string
  unsigned alloc_num; // The allocation number (count) of this block
};
```

Be sure not to allocate this struct until the client calls the *Allocate* method. You will have to initialize the actual header block to zeros when you create a page, and update it when the block is actually allocated and given to the client. When the block is freed (returned to the Object Allocator), you must set the header back to zeros.

Finally, be sure to free the struct (and the dynamically-allocated *label*) when the client frees the block of memory. Failure to do so will end up causing memory leaks.
See these Diagrams for more help.

10. **Question: What is being validated when the user calls ValidatePages?**
    Answer: There are several things that can be validated by this call. However, we are only going to be validating one aspect: the pad bytes. You need to walk each of the pages in the page list checking the pad bytes of each block (free or not). You don't need to check to see if any of the blocks are on the free list or not because you're just concerned about the padding. It's possible (in the future) to add more functionality to the*ValidatePages* method, such as checking the signatures of the free blocks and unallocated blocks. But for this assignment, checking the pad bytes is sufficient to get an understanding of how you would verify the heap.

    Also, if the debug flag is OFF or the size of the padding is 0, you will simply return 0 from the *ValidatePages* method without doing anything. This is simply because you can't check the pad bytes if they don't exist or if there are no signatures (in debug only) written to them.

11. **Question: If the user disables our allocator (by setting UseCPPMemManager to true), do we still throw an exception if the allocation from new fails?**
    Answer: Yes. However, you won't throw a std::bad_alloc exception, since the user isn't expecting that. You should catch that exception and throw an *OAException*, setting the appropriate error code (E_NO_MEMORY).

12. **Question: If the user disables our allocator (by setting *UseCPPMemManager* to true), what kinds of statistics can we still track?**
    Answer: Since all of the functionality of the object allocator is disabled, you won't be able to do much with debugging stuff. However, you can still keep counts of certain things. You can keep track of the number of free objects, the number of objects in use, and the most objects ever in use.