Programming Assignment #1

Files – Due Friday, September 19, 2014 at 2:30pm

Checklist – Due same day at the start of class

This assignment gives you some experience with memory management by applying linked list techniques to implement a rudimentary, but highly-effective, object allocator.

The task is to implement a class named ObjectAllocator which will be used to allocate and deallocate fixed-sized memory blocks for a client. The public interface for the ObjectAllocator class is very simple, almost trivial. In addition to the public constructor (and destructor), there are essentially only 2 public methods that the client requires. These are *Allocate* and *Free.* As far as the client is concerned, these methods replace the C++ **new** and **delete** operators, respectively. There are several other public methods that are present to help aid in testing, debugging, and measuring your free list management. The (partial) interface file is as follows:

```
class ObjectAllocator
{
  public:
      // Creates the ObjectAllocator with the specified values
      // Throws an exception if the first page can't be allocated. (Out of memory)
    ObjectAllocator(size_t ObjectSize, const OAConfig& config) throw(OAException);
    ~ObjectAllocator();   // Destroys the ObjectAllocator

      // Take an object from the free list and give it to the client (simulates new)
      // Throws an exception if the object can't be allocated. (Out of memory)
    void *Allocate(const char *label) throw(OAException);

      // Returns an object to the free list for the client (simulates delete)
      // Throws an exception if the the object can't be freed. (e.g. invalid object)
    void Free(void *Object) throw(OAException);

      // Frees all empty pages (pages that only contain blocks on the freelist)
    unsigned FreeEmptyPages(void);

      // Calls the callback fn for each block in use by the client
    unsigned DumpMemoryInUse(DUMPCALLBACK fn) const;

      // Calls the callback fn for each block that is potentially corrupted
    unsigned ValidatePages(VALIDATECALLBACK fn) const;

      // Returns true if FreeEmptyPages and data alignment functionality are implemented
    static bool ImplementedExtraCredit(void);

      // ********** Testing/Debugging/Statistic methods *************
    void SetDebugState(bool State);      // true=enable, false=disable
    const void *GetFreeList(void) const; // a pointer to the internal free list
    const void *GetPageList(void) const; // returns a pointer to internal page list
    OAConfig GetConfig(void) const;      // returns copy of configuration parameters
    OAStats GetStats(void) const;        // returns copy of usage statistics

  private:
      // Private data and methods
};
```

All implementations must adhere to this interface. You will use this file, **ObjectAllocator.h** in your project and implement the methods in a file called **ObjectAllocator.cpp.** The amount of code required to implement this interface is really not that great. You will need to spend sufficient time testing your code to make sure it handles all cases (allocating pages, providing debugging data, allocating objects, freeing objects, validating objects, error handling, etc.) A simple example test program and output is available to help get you started with testing. It is not an exhaustive test driver, but it is a start. You should use this sample driver as a starting point for a client program that you will create to sufficiently test your ObjectAllocator class with varying sizes of data. Also, if you are unsure of how something is supposed to work, you should ask a question. In the real world you are rarely given all of the information before starting a project (although I've tried to give you everything you will need.)

Extra Credit

There are two additional features that you can implement to earn an additional 20 points of extra credit. The first feature is to implement the method *FreeEmptyPages.* The second feature is data alignment. The details of these features will be discussed in class. Both of these features must be implemented and work correctly for you to receive the points. If you implemented these features correctly and you want them to be tested, you must return **true** from the *ImplementedExtraCredit* method. Otherwise, make sure to return **false** so you don't lose points for failing to implement the functionality.

Notes (more will be discussed in class)

1. There is a structure called *OAConfig* which contains the configuration parameters for the ObjectAllocator. You should not need to modify this.
2. There is a structure called *OAStats* which contains the statistical data for the ObjectAllocator. Do not modify this.
3. Setting *MaxPages* to 0 effectively sets the maximum to unlimited (limited only by what **new** can allocate)
4. The *DebugOn_* field is a boolean that determines whether or not the debugging routines are enabled. The default is **false**, which means that no checking is performed. When **true**, the memory allocated has certain values and depends on the current status of the memory (e.g. unallocated, allocated, freed, or padding). These values are specified in the header file on the web page for this assignment. Also, if *DebugOn_* is **true**, validation during calls to *Free* is enabled (checking for double-frees, corruption, page boundaries, etc.)
5. The *UseCPPMemManager_* field is a flag that will enable/disable the memory management system. Specifically, it allows you to use the C++ new and delete operators directly. (Effectively bypassing all memory management you've written.) With this flag enabled, you will only be able to count the total allocations/deallocations and the most objects in use at any one time. The other counts are undefined and should be ignored (e.g. pages in use).
6. The *PadBytes_* field is the number of bytes that you will "pad" before **and** after the each block that is given to the client. These bytes are used to detect memory overruns and underruns.
7. There is a struct in the implementation file called *GenericObject*, to help with the implementation (casting).
8. The exception class is provided and should not be modified.
9. There are several public methods for testing/debugging the ObjectAllocator. These will also help you in creating flexible driver programs and to test your implementation.
10. The *DumpMemoryInUse* method returns the number of blocks in use by the client.
11. The *FreeEmptyPages* method returns the number of pages that were freed.
12. The *ValidatePages* method returns the number of blocks that are corrupted. Only pad bytes are validated.
13. When checking for boundaries, your code should perform the operation in constant time. This means you don't need to "walk" through the page to determine if the block is at the proper boundary. (Hint: Use the modulus operator: %)
14. Since you're dealing with raw memory, it is natural that you will need to cast between pointers of different types. You must use the C++ named cast (e.g. **`reinterpret_cast`**). Do not use the old, unnamed casting. See the sample driver for examples. Also, look up -Wold-style-cast for GNU's compilers to help you find these old C-style casts. The graded drivers will be using this..
15. When using **new** to allocate memory, you must wrap it in a try/catch block. Do not use std::nothrow. If you don't know what that is, then you're probably not going to use it.
16. Do not use void pointers. In other words, do not create any members or variables of type **`: void *.`** They are unnecessary and will complicate your code with lots of unnecessary casting.
17. To help understand how the ObjectAllocator works, several diagrams are posted on the course website with the other documents for this project. Make sure you understand them before starting to code the assignment.
18. The purpose of this course is to learn about and understand data structures and their interfaces. That's the reason we provide public methods for accessing implementation details, which normally would remain hidden.
19. Finally, there is a wealth of additional information on the web site for this assignment. Please read all of it before beginning to implement your solution.

Deliverables

You must submit your header file, implementation file, and compiled help file (ObjectAllocator.h, ObjectAllocator.cpp, index.chm) by the due date and time to the proper submission page as described in the syllabus.

Files	Description
ObjectAllocator.h	The interface to the ObjectAllocator. You will need to modify this to help you implement the algorithm, so be very careful not to remove or change any of the public interface. I will be using drivers that expect the current interface. **No implementation is permitted in this file** (with the exception of OAException which is already provided).
ObjectAllocator.cpp	The implementation file. **All implementation goes here**. You must document this file (file header comment) and functions (function header comments) using Doxygen tags as previously
index.chm	The compiled help file.

Your code must compile cleanly with all 3 compilers to receive full credit. You should do a "test run" by extracting the files into a folder and verifying that you can compile and execute what you have submitted (because that's what I'm going to do.) Details about how to submit are described in the syllabus.

Make sure your name and other info is in all documents.