CST 238: Memory Manager Project
Due: 3 December 2014
(Wednesday) (11:55 PM PST)

Garrett McGrath
gmcgrath@csumb.edu
(831) 656-3316

12 November 2014

Introduction

For this assignment, you will be responsible for implementing a dynamic memory manager. In our C++ programs this semester, whenever we invoked the `new` and `delete` operators, we called upon the standard library's memory manager. As you experienced it, memory management involves taking client requests to allocate a particular amount of memory and, later on, to free that memory for other uses. Memory management is a fundamental problem in computer science and features heavily in operating systems, systems programming and any memory constrained environment. As we have seen, it is essential to solving a large class of computational problems.

As you might imagine, memory management can get quite complicated; there are a number of different approaches to it. These approaches are differentiated by the strategies they implement for finding a free block of memory to service a client's request, and by their choice of data structures. Effective memory management requires a deep understanding of algorithms and of the proper data structures to use for a given task. For this project, you are asked to implement two strategies for memory management: *best fit* and *first fit*.

This page from cprogramming has a good overview of the topic. Moreover, this page from Virginia Tech will give you additional helpful explanation and includes a visualization of each approach. Lastly, a memory management reference page provides even more assistance and guidance.

What To Do

For this project you will be provided with skeleton code for implementing a memory manager. You will need to begin with the code provided on the course's Github page. Your job is to fill in code in MemoryManager.cpp, BestFit.h, BestFit.cpp, FirstFit.h, and FirstFit.cpp.[1]

Project 2's skeleton code consists of the following files:

1. ***MemoryManager.h***: This file contains the class definition for *MemoryManager*. It provides you as the implementer access to its private methods and attributes via the protected access modifier. In addition, the protected access modifier allows all derived classes to access these class members. To an invoking object, however, protected members are treated the same as private ones.

2. ***MemoryManager.cpp***: This file has blank method definitions which you need to fill in. Everywhere a comment says "Your code here" that is your cue to erase any statements currently there and replace them with your implementation. Each function is commented within and in the corresponding header file.

3. ***BestFit.h* and *BestFit.cpp***: The basic code for implementing the Best Fit algorithm (described below) for memory management. You must implement allocate.

4. ***FirstFit.h*** and ***FirstFit.cpp***: Same as with *BestFit*, but for the First Fit memory allocation algorithm.

Memory Allocation Implementation

When filling in code for the *MemoryManager* base class, the *BestFit* and *FirstFit* subclasses you must abide by the following:

- Throughout the skeleton code you will encounter comments saying, "Your code here" as a cue that you need to provide your own code. You can delete any statements below that comment as they are only there as a placeholder.

[1] In addition, you will need to write code to test your implementation enough to convince yourself that it is correct. However, you will not turn in anything but the .cpp and .h files listed above.

- Do not remove any of the methods or attributes currently there. You are allowed to add additional methods and attributes, but you must ***NOT*** remove any provided ones.

- Do not use *new* or *delete*. The only approved use of those operators occurs in the *MemoryManager* constructor.

- Memory is represented in the *MemoryManager* class as an array of *unsigned char*'s, which have been typedef'ed as *byte*. For every MemoryManager object there is a maximum amount of memory. There is also a block size representing the smallest unit of memory which can be allocated per client request. This means that every amount of memory allocated ***must be*** a multiple of the block size. One or more contiguous blocks are referred to as a *chunk*.[2]

- In order to implement *Best Fit* and *Worst Fit* you must implement the *allocate* and *free* methods. Both methods deal with *void **'s, which are generic pointers. They can potentially point to any type, and are found throughout the C standard library. It is the responsibility of the caller of *allocate* and *free* to type-cast any *void **'s to the appropriate type.

- *allocate* takes a number of bytes and returns back a pointer to the start of the block or chunk of memory allocated. Each base class requires its own implementation of this method. *allocate* ***MUST*** only allocate blocks of memory. If not enough memory is available to service the caller's request, you must throw the pre-defined *allocationException*.[3] The code to do this requires the single statement:

```
throw allocationException;
```

- *free* returns nothing but takes a pointer to the start of a previously allocated chunk or block. *free*'s implementation is constant across all memory allocation algorithms. You must figure out how to keep track of the size of each allocation as no code or hint is provided to you about how this should be done. If any error occurs, i.e., the client tries to free

[2]To the caller, the use of block sizes is invisible. This only has a bearing on you as the programmer.

[3]In the <new> header file, there is a *bad_alloc* exception. This project's use of a custom exception is for purposes of code demonstration. If you were using the standard library's exception you would use something like the following statement:
`throw std::bad_alloc();`

the same location twice, or the pointer is not in the range of allocated memory addresses, you must throw the pre-defined *freeException*. The code to do this requires the single statement:

```
throw freeException;
```

- The MemoryManager class maintains a linked-list of free blocks using the C++ standard library *list* class and a protected *struct* definition for a *FreeBlock*. Get familiar with this definition in *MemoryManager.h*.
- You must implement all missing method definitions found in *MemoryManager.cpp*.

Best Fit

Best Fit abides by the following strategy: when a request for allocation arrives, search the entire free list and allocate the smallest chunk/block possible.[4] This is meant to reduce fragmentation at the cost of slower allocation than other strategies. Allocation is then always a linear time operation, i.e., if memory doubled it would take twice as long to service an allocation request.

First Fit

First Fit abides by the following simple strategy: when a request for allocation arrives, return the first available block/chunk that satisfies the request. Like Best Fit, First Fit also always begins searching for a free block/chunk at the start of the free list.

How To Turn In

Turn in all your C++ source files, i.e., *MemoryManager.h*, *MemoryManager.cpp*, *BestFit.h*, *BestFit.cpp*, *FirstFit.h*, *FirstFit.cpp*, on **iLearn** in a single zip file named *Project2<Your Last Name>.zip*. You will be graded on identical criteria as previous homeworks and projects.

Points-wise 100 are available. There will be 50 points for successful compilation, 30 points for the MemoryManager, 10 points for Best Fit, and 10 points for First Fit.

[4]When deciding between two chunks, the smaller one is always chosen. For example, assume a block size of 64 bytes and that a client requests 50 bytes. After scanning the free list, the memory manager learns that there is a 2 block chunk and a single block available. Rather than split the chunk, the single block will be chosen by Best Fit. If the chunk came before the single block, First Fit would have chosen it.