Assignment #4

CS 180 Fall 2020

Due Date:	As specified on the moodle
Topics covered:	Contiguous Memory Allocation
Deliverables:	Your submission should consist of two files: the header (interface) file, and the source (implementation) file. The header file must be named `MemoryManager.h` (with capital letters as indicated); this file should declare the class `MemoryManager` whose public interface is identical with that given at beginning of this handout. The source file must be named `MemoryManager.cpp` (again, with capital letters as indicated). These files must be placed in a directory and further zipped according to the specifications detailed in the syllabus. For this assignment, you may only use the standard C++ header files `iostream`, `list`, `iomanip`, `cstdlib`, and `cstdio`; in particular, the `windows.h` header file may not be included. Your submission should compile without errors or warnings using `cl /W4 /EHsc /c MemoryManager.cpp` on the command line.
Objectives:	To demonstrate an understanding of how a simple memory manager allocates memory using the contiguous allocation scheme.

1 Programming Statement

In this assignment you will write a simple memory manager for dynamically allocated memory. The memory manager will make use of a linked list to keep track of allocated and unallocated blocks of memory; the algorithm for this was discussed in class (see the class lecture notes).

2 Public interface

The *public* interface for the memory manager must be as follows (it assumes that the file `iostream` has been included):

```
class MemoryManager {
  public:
    MemoryManager(int total_bytes);
    ~MemoryManager(void);
    void *allocate(int bytes);
    void deallocate(void *pointer);
    void dump(std::ostream& out);
};
```

The public member functions listed above are defined in detail below. You are free to specify the *private* portion of the class as you see fit. Also, you may implement the `MemoryManager` using either the STL `list` class, or you may wish to create the linked list yourself from scratch; note however, that you will need a doubly linked list.

2.1 Member functions

`MemoryManager` — create a memory manager instance of a specified heap size (in bytes). Your memory manager should use `new` (or `malloc`) only once to create the entire heap of memory that it will use for memory allocation (note: if you decide to make your own linked list, you may also use `new` (or `malloc`) to create new nodes for the list). The construction parameter `total_bytes` gives the total size of the heap in bytes. For example, the code fragment

```
MemoryManager mm(1<<20);
```

creates an instance of the memory manager, named `mm`, with a heap size of 1 Mb. Each subsequent call to the `allocate` member function will return a pointer to a chunk of memory within this 1 Mb block.

`~MemoryManger` — destroys a memory manager instance. As a bare minimum, the destructor will deallocate the heap created by the memory manager constructor. This should be done with a single `delete` (or `free` if you used `malloc` to allocate the heap). Again, if you are making your own linked list, you may invoke `delete` (or `free`) for the nodes in the list that you remove).

`allocate` — get a block of memory of a specified size (in bytes) from the heap. On success, the function returns a pointer to a memory block of the requested size; on failure, a zero (null) pointer is returned. Note that the pointer should be recast to a pointer of the desired type. For example, the code fragment

```
int *array = (int*)mm.allocate(100*sizeof(int));
```

will allocate space on the heap for an array of 100 integers.

`deallocate` — frees a block of memory (that was allocated using `allocate`) from the heap. The value of `pointer` should be a non-zero pointer that was returned by the `allocate` member function. The code fragment

```
mm.deallocate(array);
```

frees the space on the heap used by the array of integers allocated above. If `pointer` has a value of zero, refers to an address that was not returned by `allocate`, or refers to the addres of a block that was previously deallocated, then the effect of `deallocate` is undefined; no error checking is performed.

`dump` — prints information about the current structure of the heap to the specified output stream. This function is intended for debugging purposes; it prints a list of the current blocks on the heap: the starting address of the block, the block size (in bytes), and whether the block is currently in use (allocated) or not (deallocated/free). The code fragment

```
mm.dump(std::cout);
```

prints infomation about block structure of the heap to the standard output. An example of the ouput from this function might be something like this

```
start address: 804c008
  byte count: 78
  allocated? false
start address: 804c080
  byte count: 28
  allocated? true
start address: 804c0a8
  byte count: f60
  allocated? false
```

which indicates that the heap is currently divided into three blocks of size 78h, 28h, and F60h bytes, respectively; only the middle block is allocated, the other two are free. Your code should emulate this output as much as possible; in particular, *all numerical values should be in hexadecimal.*

2.2 Details on `allocate` and `deallocate`

The `allocate` member function should search for a block of memory on the heap of the requested size. Other than this, you are free to implement the function as you see fit: you may choose the first available block, or you may wish to choose the *best fit* block (smallest block that is at least large as the requested size), or the *worst fit* block (the largest block).

Not only should the `deallocate` function mark the specified block as free (deallocated), but should also consolidate adjacent free blocks into a single free block. For example, if the heap currently has the structure indicated above in the description of the `dump` function, then the call

```
mm.deallocate(0x804c080);
```

would result in the heap having the structure

```
start address: 804c008
  byte count: 1000
  allocated? false
```

i.e., all three of the blocks on the heap are consolidated into a single free block.