Assignment 4

Due Date: Monday April 8 at 11:59pm

Objectives

- Implementing object manager
- Applying principles of Design by Contract

Note

- Please follow the programming standards (available in UMLearn); not doing so will result in a loss of marks.
- If you write your code on your own computer, remember to verify your code on the lab machines before submission.
- Your assignment code must be handed in electronically, see the Assignment Submission Guideline (under the Admin module in UM Learn) for details.
- This is a detailed assignment so rather than writing an essay, most of the details will be discussed in class.

Summary

In this assignment, you are to implement a simple object manager. An object manager is another form of table, but we now store generic objects in a managed buffer (the internal implementation of a table varies based on need). Using a buffer means that we must directly manage the memory and references to objects. This means that our object manager must implement reference counting and a garbage collector, so that we properly handle creation and deletion of objects. The object manager's interface is given by the file *ObjectManager.h* and the implementation will be in the file *ObjectManager.c*. Your task is to implement the functions required for the object manager. This includes all the functions listed in *ObjectManager.h*, along with any (private) static functions that will be useful. You will also need to define appropriate data types and data structures for keeping track of the objects that have been allocated. You are NOT allowed to change the prototypes of the functions that are provided to you in *ObjectManager.h*. To summarize, the functionalities you have to implement are:

- *initPool()* - Initialize the object manager upon starting.
- *destroyPool()* - Clean up the object manager upon quitting.
- *insertObject(size)* - request a block of memory of given size from the object manager.
- *retrieveObject(id)* - retrieve the address of an object, identified by the reference id.
- *addReference(id)* - increment reference count for the object with reference id.
- *dropReference(id)* - decrement reference count for the object with reference id.
- *compact()* - initiate garbage collection (see below).
- *dumpPool()* - print (to stdout) info about each object that is currently allocated including its id, start address, size and reference count.

A more thorough discussion of the functionality of these routines will be discussed in class.

Garbage Collection

You will implement a Mark and Sweep Defragmenting garbage collector, as described in class. This function is called *compact()* in the object manager. So that we can evaluate your implementation, every time the garbage collector runs print out the following statistics (to stdout):

- The number of objects that exist
- The current number of bytes in use
- The number of bytes collected

Data Structures

You must manage the tracking of allocated objects using an index, which will be implemented using a linked list. Each node in your linked list contains the information needed to manage its associated object. Note that the index is implemented inside the object manager, you don't need any additional files.

Testing your Object Manager

You can use this *main.c* file for an example of how the Object Manager can be used correctly. Use design by contract to ensure that your code is solid. Write a thorough set of unit tests. You will be given additional main programs closer to the due date. Your code must be able to handle all of the situations we might throw at it! The more you anticipate errors, the easier it will be to deal with those additional test cases.

Notes

- Do not wait until additional main programs are posted to think about error checking or you will not have enough time to finish.
- Test each feature as it is implemented. Create your own test cases to test your code.
- Use the name *main.c* for your main program (do not change it).
- Your code must be "reasonably" efficient. There will be main programs that create lots of objects, but they will not stress all aspects simultaneously. They must run in a reasonable amount of time. Linear-order algorithms will be fine (i.e. linear search is OK, binary search is not required).
- Hand in your source code and a *Makefile* capable of compiling all your code. Include a file named *main.c*, which contains all of your test cases.