

CS2604 (Summer II 2001)
PROGRAMMING ASSIGNMENT #1

Due Friday, July 13 @ 11:00 PM for 100 points

Late penalty: 10 points/day with final due date Monday, July 16 at 11:00 PM

Assignment:

You will write a memory management package for storing **variable-length records** in a large memory space. For background on this project, read Section 12.4 of the textbook.

Your **memory pool** will consist of a large array of characters. You will use a doubly linked list to keep track of the free blocks in the memory pool, which will be referred to as the **freeblock list**. The freeblock list should store the free blocks in descending order by size of the free block. If two multiple blocks are of the same length, then they should appear in ascending order of their position in the memory pool. You will use the worst fit rule for selecting which free block to use for a memory request. That is, the first free block in the linked list (which is the largest block) will be used to service the request if possible. If not all space of this block is needed, then the remaining space will make up a new free block and be returned to the free list.

Be sure to merge adjacent free blocks whenever a block is released. To do the merge, it will be necessary to search through the freeblock list, looking for blocks that are adjacent to either the beginning or the end of the block being returned. Do **not** merge the free blocks at the beginning and end of the memory pool. That is, the memory pool itself is not considered to be circular.

The records that you will store will contain the *xy*-coordinates and name for a city. Aside from the memory manager's memory pool and freeblock list, the other major data structure for your project will be the **record array**, an array that stores the "handles" to the data records that are currently stored in the memory pool. A handle is the value returned by the memory manager when a request is made to insert a new record into the memory pool. This handle is used to recover the record. Note that the record array is something of an artificial construct that is being used to simplify the testing procedures for the project. The idea is that the record array gives us an easy way to give an identification number to the records independent of their placement in the memory pool.

Invocation and I/O Files:

The program will be invoked from the command-line as:

```
memman <pool-size> <num-recs>
```

The name of the program is **memman**. Parameter **<pool-size>** is the size of the memory pool (in bytes) that is to be allocated. Parameter **<num-recs>** is the size of the record array that holds the handles to the records stored in the memory pool. Your program will read from standard input a series of commands, with one command per line. The program should terminate after reading the EOF mark. No command line will require more than 80 characters. The formats for the commands are as follows. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. All coordinates will be signed values small enough to fit in a 32-bit **int** variable.

```
insert recnum x y name
```

Parameter *recnum* specifies which slot in the record array will hold the handle for this record. An error should be reported if the value of *recnum* is outside of the range 0 to **num-recs** - 1. Parameters *x* and *y* are the *xy*-coordinates for the record, and *name* is the name of the city for this record. *name* may consist of upper and lower case letters and the underscore symbol. If there is

already a record stored at position *recnum* in the record array, then that earlier record should first be removed from the memory pool. If there is no room in the memory pool to handle the request, print a suitable message and do not modify the memory pool in any way. If the insert command is to a *recnum* that is already used, then the first step will be to delete the old record, and the second step will be to attempt to insert the new record. Should this attempt to insert fail, then the old record will remain deleted.

remove *recnum*

Remove the record whose handle is stored in position *recnum* of the record array. If there is no record there, print a suitable message. An error should be reported if the value of *recnum* is outside of the range 0 to `num-recs - 1`.

print *recnum*

Print out the record (coordinates and name) whose handle is stored in position *recnum* of the record array. If there is no record there, print a suitable message. An error should be reported if the value of *recnum* is outside of the range 0 to `num-recs - 1`.

print

Dump out a complete listing of the contents of the memory pool. This listing should contain two parts. The first part is the listing of city records currently stored in the memory pool, in order of the record number. Print the value of the position handle along with the record. The second part is a listing of the free blocks, in order of their occurrence in the freeblock list.

Design Considerations:

Your main design concern for this project will be how to construct the interface for the memory manager class. While you are not required to do it exactly this way, we recommend that your memory manager class include something equivalent to the following methods.

```
// constructor
MemManager(int poolsize);

// Insert a record and return its position handle.
// space contains the record to be inserted, of length size.
int insert(void* space, int size);

// Free a block at posHandle. Merge adjacent blocks if appropriate.
void remove(int posHandle);

// Return the record with handle posHandle, up to size bytes.
// Place the record into space.
void get(void *space, int posHandle, int size);

// Dump a printout of the freeblock list
void dump();
```

Another design consideration is how to deal with the fact that the records are variable length. One option is to store the record's handle and length in the record array. An alternative is to store the record's length in the memory pool along with the record. Both implementations have advantages and disadvantages. We will adopt the second approach.

The records stored in the memory pool **must** have the following format. The first byte will be the length of the record, in bytes. Thus, the total length of a record may not be more than 256 bytes. The next four bytes will be the x -coordinate. The following four bytes will be the y -coordinate. Note that the coordinates are stored in binary format, not ASCII. The city name then follows the coordinates. You should **not** store a NULL terminator byte for the string in the memory pool.

Programming Standards:

You must conform to good programming/documentation standards, as described in the Elements of Programming Style. Some specifics:

- You must include a header comment, preceding `main()`, specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation, unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point. You may not use code from STL, MFC, or a similar library in your program.

Testing:

A sample data file will be posted to the website to help you test your program. This is not the data file that will be used in grading your program. While the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

You will submit this project electronically. In particular, you will create a zip'ed archive file containing the following items (and nothing else):

- all source code files necessary to build an executable
- either the project workspace files (.dsw and .dsp) for Visual C++ users, or a makefile for g++ users.

Windows users should be sure to use a modern zip tool which preserves long file names. A suitable freeware command-line zip tool will be posted on the course website. UNIX users should submit a gzip'ed and tar'ed file.

Once you have assembled the archive file for submission, for your own protection, please move it to a location other than your development directory, unzip the contents, build an executable, and test that executable on at least one input file. Failure to do this may result in delayed evaluation of your program, and a loss of points.

You will submit your project to the automated Curator server. The instructions and necessary software are available at: <http://ei.cs.vt.edu/~eags/CuratorGuides.html>. If you make multiple submissions, only your last submission will be evaluated.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement in the header comment preceding the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor :
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - All C++ language code and documentation used in my program
//   is either my original work, or was derived, by me, from the source
//   code published in the textbook for this course.
//
// - I have not discussed coding details about this project with anyone
//   other than my instructor, ACM/UPE tutors or the GTAs assigned to this
//   course. I understand that I may discuss the concepts of this program
//   with other students, and that another student may help me debug my
//   program so long as neither of us writes anything during the discussion
//   or modifies any computer file during the discussion. I have violated
//   neither the spirit nor letter of this restriction.
//
```

Programs that do not contain this pledge will not be graded.