CS3723 Pgm Assignment #1: Storage Management in C (60 pts)

**© Copyright 2020 Larry Clark, this document must not be copied to any other website**

In this assignment, you will create **storage management functions** which manage memory and implement garbage collection. Your code must understand metadata to support the overall garbage collection process. Why is that necessary?

There are several important phases of memory management:

**Initialize Phase** Initialize the memory to one huge free node. The storage manager's pFreeHead will then point to the starting address in that huge node having a cGC of 'F' (free). Although the free memory list initially has only one node, it can have many free nodes after a Garbage Collection Phase completes. The initialize phase only happens once - prior to any usage.

**Usage Phase** In this phase:

* When memory allocation is requested, your memory allocation function (gcAllocate) will return a suitable "In Use" node from a free node.
* The "In Use" node will be marked with a cGC of 'I' (in use).
* Your memory allocation function might return the left over portion of the free node to the free list or include it in the "In Use" node. (see below)
* In this phase, we will also associate nodes to each other.
* Other than returning the left-over portion of an allocation request to the free list, we will NOT free any nodes (i.e., we won't change any nodes from 'I' to 'F') during the Usage Phase.

**GC Phase** There are three subphases to the Garbage Collection Phase :

**Mark Subphase** Walk through all memory from the beginning and mark **all** nodes to a cGC of 'C' (candidate). It is easier to combine adjacent free areas if all nodes are marked as 'C'.

**Follow Subphase** From any valid starting point (which we are simulating with a hash table), follow the nodes based on metadata and mark each referenceable node as 'I' (in use).

**Collection Subphase** We will build the free list from the entire memory. (We are affectively ignoring the old free linked list. Discard any knowledge of it.) Walk through all the memory looking for 'C' nodes, combining adjacent ones into a single free area. Each free area will be placed on the front of the new free list.

To help reduce the code that you have to write (allowing you to focus on the most important concepts) and to reduce debugging difficulty, I have provided the following (in **/usr/local/courses/clark/cs3723/2020Sp/Pgm1**)

cs3723p1Driver.c - reads the input file, calls your storage management functions, and uses a hash table to store the addresses of allocated memory (so that those can be subsequently associated/disassociated with other nodes). An important function is the setData function which uses metadata to set attributes in a node. The driver also provides functions for printing the heap memory to help with debugging.

cs3723p1.h - include file for this program. Some important typedefs:

NodeType – describes one node type: name, beginning subscript in metaAttrM array, total size

MetaAttr – describes one attribute in a node type: name, type, size in bytes, and offset

InUseNode - contains the node's size, node type, cGc, and the user's data

GCManager - a structure that contains the address of the heap (pBeginStorage), pointer to the end of the heap (to the address that immediately follows it), pointer to the head of the free memory list, an array of NodeType entries, and an array of MetaAttr entries. It does not have a count of the number of entries in those arrays. Instead, sentinels are used to mark the end of the arrays.

GCResult - used by your gcXXXXXXX functions to specify whether they executed successfully.

FreeNode - the first portion of a free node.

hashApi.cpp - C++ code to integrate C with the C++ Hash Table Class (unordered\_map). This supports functions getHash, putHash, eraseAll, and getAll. This is only used by the driver.

printNode.o - A function to print the contents of a node, showing each attribute and its values. This is used by the driver when printing the heap's contents.

p1Input.txt - Input text file suitable for the driver. The driver uses **stdin** so redirect input from this file.

p1InputGC.txt - Same as p1Input.txt except that it has a DEBUG command to stop calling the drDump function when reading commands; instead, it calls drDump during garbage collection. Once you get input file steps 1-11 working, you will probably want to use this input file during testing.

makefile - Please use this makefile to create your **p1** executable. Note that you should not use the hen servers. The makefile uses **g++** instead of gcc. To build the executable (with it automatically building the other pieces), type  
 **make p1**

**You will need to code** the following functions; however, due to modularity concerns, you may need to create 1 or more extra functions. Your code should be placed in cs3723p1.c.

void **gcInit**(GCManager \*pMgr)

This is part of the Init Phase. The **driver** already set pMgr->pBeginStorage to the address of the heap, pMgr->iHeapSize, pMgr->pEndStorage, and pMgr->iMinimumNodeSize. Initialize the memory to one huge free node. You need to set pMgr->pFreeHead to point to the starting address of that huge node.

* Zero out the entire heap.
* Set the cGC to 'F', (free node)
* Set the shNodeSize

void \* **gcAllocate**(GCManager \*pMgr, short shDataSize, short shNodeType, char sbData[]

, GCResult \*pGCResult)

This is part of the Usage Phase. This is satisfying a memory allocation request using the first node from the free list that is big enough.

* It is passed the user data size (which is less than the size to allocate due to the node overhead of size, node type, and cGC), the node type, and the user's data. How much do we allocate? the user data size plus ?
* Searches for a free node with enough memory to satisfy the needed size. If there isn't a large enough free node to satisfy the request:
  + Set pGCResult->rc to RC\_NOT\_AVAIL.
  + Copy an error message to pGCResult->szErrorMessage.
  + Functionally return NULL.
* The remaining bullets assume there was a large enough free node to satisfy the request.
* Remove that free node from the free list, updating the free list. Do this regardless of whether there is enough free space left in the free node to make a new free node
* The new InUseNode will **begin** at the address of the free node.
* Determine whether there is enough remaining space in the free node to make that remaining space a new free node (see pMgr->iMininumNodeSize). If there isn't enough space left, change the allocated size to be the free node's size. Otherwise, add the new free node (which uses the remaining space) to the free list. (We will **ALWAYS** add to the **FRONT** of the free list.)
* Initializes an InUseNode:
  + Set its cGc to 'I'.
  + Set its node type.
  + Set its size.
  + Set its sbData.
* Returns a pointer (from the user's perspective) to the allocated memory. This is not the address of the InUseNode

void **gcMark**(GCManager \*pMgr, GCResult \*pGCResult)

This is the first subphase of Garbage Collection. Beginning with the start of the heap, sequentially mark every node as 'C'. Look at the driver's drDump function to see how to advance through the heap touching every adjacent node.

void **gcFollow**(GCManager \*pMgr, void \*pUserData, GCResult \*pGCResult)

The Follow subphase of Garbage Collection is handled by the driver and **gcFollow**. The **driver will** actually **call this function multiple times** during this subphase. It is passed a user data pointer.

* If the current node's cGC is 'I', it has already been followed; therefore, return.
* Otherwise, set its cGC to 'I'. Recursively follow each non-null pointer referenced by this node. Your code must make certain that a cycle in the data doesn't cause an infinite loop. This code has to understand metadata to know where the pointers are located.

void **gcCollect**(GCManager \*pMgr, GCResult \*pGCResult)

This is the third subphase of Garbage Collection. In this phase, you will sequentially traverse the heap, collecting the 'C' nodes, combining adjacent 'C' nodes, and placing the nodes onto a new free list. Each insertion will be to the **front** of the free list. As you collect free space, call one of these two driver functions:

drPrintCollecting(pMgr, pCandidate);

drPrintCombining(pMgr, pPrecedes, pCandidate);

void **gcAssoc**(GCManager \*pMgr, void \*pUserDataFrom, char szAttrName[]

, void \*pUserDataTo, GCResult \*pGCResult)

sets a user pointer in the specified user data node to a new referenced user data node.

* Search for the specified attribute name in the meta data for the **from** node.
* If not found, set the pGCResult->rc to RC\_ASSOC\_ATTR\_NOT\_FOUND, provide an error message that contains the attribute name, and return. (Ignore the remaining bullets.)
* If the specified attribute is not a pointer, set the pGCResult->rc to RC\_ASSOC\_ATTR\_NOT\_PTR provide an error message that contains the attribute name, and return. (Ignore the remaining bullets.)
* Change the user pointer in the specified user data node to point to the new referenced user data node or NULL (if that was specified).
* Hint: once you know the offset in the user data:

void \*\*ppNode = (void \*\*)&(pInUseFrom->sbData[pAttr->shOffset]);

\*ppNode = pUserDataTo;

**Grading**:

* Your code must follow my **programming standards**.
* You must make certain your code works on a **fox server** and can be compiled by the specified **makefile**.
* Points:
  + Following program standards is worth 2-6pts.
  + Allocating nodes correctly from a free list and associating nodes (input steps 1-11, 15-17) is worth 10-30 pts.
  + Garbage collection, when empty, when not adjacent, and when adjacent is worth 30-40 pts.
  + If your program can’t execute for all of the input text, the grader won’t know if any subsequent code is working. Therefore, he/she will take off for all remaining input cases.

**Notes**:

1. To help understand how to use the metadata, examine the driver's setData function.
2. To simplify grading, please include your C code and output (generated on a **fox** server for p1Input.txt) in a zip file with your name in mixed case with last name followed by first (e.g., KingJoe.zip). Do not embed any directories within that zip file.
3. See the sample partial output on BlackBoard.
4. For **Netbeans** users, see the **Netbeans Considerations for Pgm1** Word document.
5. For **Microsoft Visual Studio** Users:

* Since I provided the printNode as a .o file, you can't use it. Inside the driver, rename dumbPrintNode to printNode. Please remember to delete those before running on the UTSA Linux server or submitting your program.
* If you need a wider Console Window:
  + Once the console window displays (you may want a break point in your code so that it doesn't disappear), click the top left corner of the console window.
  + Properties
  + Layout
  + Change the Screen Buffer Size to 120
  + Change the Window Size to 120