COMP 2401 -- Assignment #3

Due: Thursday, November 6, 2014 at 9:00 PM

Goal

You will implement, in C using the Ubuntu Linux environment, a suite of functions to help the user manage the memory that they dynamically allocate in their programs. This means that, for this assignment, your user will be another programmer! The *Memory Hunter* functions that you implement will still use **`malloc`** and **`free`**, but they will also keep track of every *memory block* that is allocated and freed. That way, the user will be able to print to the screen information about these allocated memory blocks, get a count of the number of bytes that are currently allocated, and garbage collect all the memory blocks at once.

In this assignment, you will:

- implement functions to allocate, deallocate, garbage collect and report upon memory allocated at runtime
- augment existing code to provide additional testing

Learning Objectives

- work with dynamically allocated memory
- perform more complex pointer manipulations
- develop software for other programmers rather than end users

Instructions

1. Memory Hunter functions

The Memory Hunter functions keep track of all blocks of dynamically allocated memory, by storing information about each block as it is allocated. This allows the user to query a **`HeapType`** structure to see how much memory is currently used, dump to the screen information about the allocated memory blocks, and perform garbage collection on all dynamically allocated memory. You must use the header files and **`main`** function provided here: a3Posted.tar

Implement the following functions:

- an initialization function with prototype: **`void mh_init(HeapType *heap)`**
  - this function initializes the **`heap`** variable declared in **`main`**, including the dynamic allocation of the maximum number of blocks
- a cleanup function with prototype: **`void mh_cleanup(HeapType *heap)`**
  - this function frees the memory allocated for all the blocks in the **`heap`** parameter
- a memory allocation function with prototype: **`void *mh_alloc(HeapType *heap, int n, char *label)`**
  - using **`malloc`**, this function dynamically allocates a new block of memory of size **`n`** bytes
  - it adds the new block of memory as the next element of the **`blocks`** array in the **`heap`** parameter
  - it saves the address of the new memory in that **`blocks`** element, as well as its size and a user-defined **`label`** to indicate the use for this block, and it sets the **`blocks`** element to reserved
  - it returns the pointer returned by **`malloc`**
- a memory deallocation function with prototype: **`void mh_dealloc(HeapType *heap, void *addr)`**
  - this function finds the **`blocks`** element in the **`heap`** parameter that corresponds to the block of memory pointed to by **`addr`**
  - it sets the **`blocks`** element to not reserved and deallocates the memory pointed to by **`addr`** using **`free`**

- a memory count function with prototype: **`int mh_count(HeapType *heap)`**
  - this function returns the total number of bytes currently reserved in all blocks of the **`heap`** parameter
- a screen dumping function with prototype: **`void mh_dump(HeapType *heap)`**
  - this function prints to the screen information about every block in the **`heap`** parameter
  - a sample output can be found below
- a garbage collection function with prototype: **`void mh_collect(HeapType *heap)`**
  - using **`free`**, this function deallocates all memory tracked in the **`blocks`** array of the **`heap`** parameter

Notes:

- you **must** use the header files and **`main`** function provided
- you **must** use the function prototypes exactly as stated above, without modification
- do **not** make any changes to the header files provided, nor to the existing code in the **`main`** function provided

2. Additional testing

Modify the **`main`** function provided to add new tests for the Memory Hunter functions. At minimum, your program will use the Memory Hunter functions to:

- perform memory allocations and deallocations of at least five new data types not already tested in **`main`** (you can define new structures for these data types, as needed)
- dump the content of the heap after every allocation and every deallocation, and print out the current byte count

Sample Output

Below is the output of a sample execution:

```
Don't Panic ==> valgrind a3
==2582== Memcheck, a memory error detector
==2582== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==2582== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==2582== Command: a3
==2582==

DUMP 1, byte count = 1116
               ints:     20 bytes stored at 0x41ee350
            doubles:     80 bytes stored at 0x41ee398
              chars:      8 bytes stored at 0x41ee418
           Students:   1008 bytes stored at 0x41ee450

DUMP 2, byte count = 1008
           Students:   1008 bytes stored at 0x41ee450

DUMP 3, byte count = 0


==2582==
==2582== HEAP SUMMARY:
==2582==     in use at exit: 0 bytes in 0 blocks
==2582==   total heap usage: 6 allocs, 6 frees, 1,828 bytes allocated
==2582==
==2582== All heap blocks were freed -- no leaks are possible
==2582==
==2582== For counts of detected and suppressed errors, rerun with: -v
==2582== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Don't Panic ==>
```

Constraints

- you **must** use the header files and **`main`** function provided, without modification, unless otherwise instructed
- do **not** use any global variables
- compound data types **must** be passed by reference, not by value
- you must manage your memory! use valgrind to find and fix memory leaks
- you must reuse functions everywhere possible
- your program must be thoroughly commented

Submission

You will submit in *cuLearn*, before the due date and time, one **`tar`** file that includes all the following:

- all source code, including the code provided
- a readme file, which must include:
  - a preamble (program author, purpose, list of source/header/data files)
  - exact compilation command
  - launching and operating instructions

Grading

- **Marking breakdown:** The grading scheme is posted in cuLearn.
- **Deductions:**
  - Up to 50 marks for any of the following:
    - the code does not compile using gcc in the VM provided for the course
    - unauthorized changes have been made to the code provided for you
    - code cannot be tested because it doesn't run
  - Up to 20 marks for any of the following:
    - your program is not broken down into multiple reusable, modular functions
    - your program uses global variables (unless otherwise explicitly permitted)
    - your program passes compound data types by value
    - the readme file is missing or incomplete
  - Up to 10 marks for missing comments or other bad style (indentation, identifier names, etc.)
- **Bonus marks:**
  - Up to 5 extra marks are available for fun and creative additional features