

## CS107 Practice Midterm Exam

---

### Exam Facts

Normal Time: Wednesday, May 7<sup>th</sup> at 7:00 p.m. in Hewlett 200.

The exam is open-notes, open-book, closed-computer. You should bring your own lecture notes, the course handouts, and any printouts of assignment code you'll want to consult during test time.

This practice exam is constructed from actual midterms I've given in previous quarters. You will effectively have as much time as you want, although we will need to pull the exams at 10:00 p.m., since that's when the room needs to be locked back up. Those on campus who need to take the exam earlier in the day on Wednesday may do so, provided you start some time after 9:00 a.m. (but not during the hour that I teach. ☺)

### Material

The exam will focus on material like that covered on the first five assignments. Be prepared for C/C++ coding questions requiring strong understanding of pointers, references, arrays, function pointers, and low-level memory manipulation, as well as questions on code generation, function call and return, variable layout, stack and heap implementation, and the compilation process as covered in Assignment 5.

In general, a good way to study for the coding questions is to take a problem for which you have a solution (lecture or section example, homework problem, sample exam problem) and write out your solution under test-like conditions. This is much more valuable than a passive review of the problem and its solution where it is too easy to conclude "ah yes, I would have done that" only to find yourself adrift during the real exam when there is no provided solution to guide you!

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on your midterm. Understand that I am under no obligation to mimic this exam when writing yours. That being said, any material covered in lecture or on an assignment is fair game.

**Problem 1: San Francisco Fine Dining**

Consider the following **struct** definitions:

```

typedef struct {          typedef struct {
    int **garydanko;      short ame[2];
    int aqua[3];          appetizer farallon;
    char *quince;         char bacar[8];
} appetizer;             } dessert;

dessert *dinnerisserved(short *boulevard, appetizer *jardiniere);
int *bonappetit(dessert azie, char **indigo)
{
    appetizer oola;
    dessert *catch;

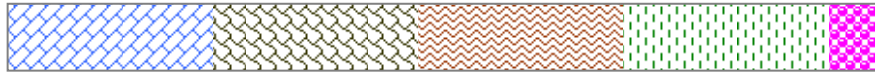
line 1    azie.bacar[azie.ame[2]] += catch->farallon.aqua[4];
line 2    ((appetizer *)(((dessert *)(&oolo.quince))->farallon.garydanko))->quince = 0;
line 3    return (*dinnerisserved((short *) &indigo, &oolo).farallon.aqua;
}

```

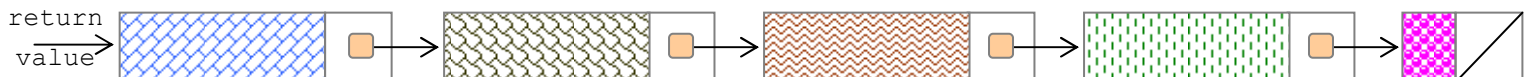
Generate code for the entire **bonappetit** function. Be clear about what assembly code corresponds to what line. You have this and the next page for your code.

## Problem 2: Breaking Larger Data Images Into Packets

Given the address of some binary image, its size in bytes, and a packet size in bytes, write a function called **packetize** that creates a linked list of packets. So, given the following address of the following, 1700-byte data image:



a call to **packetize(image, 1700, 400)** would return the address of the lead node of a NULL-terminated list that looks like this:



Note that each node is a **400**-byte portion of the original image, following by the four-byte address of the next node in the list. The last node is smaller than the others, because it only needs to store the last 100 bytes of the full image.

A call to **packetize(image, 1700, 500)** would return the head of a different list—one that looks like this:



Each node contains **500** bytes of data, save for the last one, which contains  $1700 \% 500 == 200$  bytes of data. (Note that the pointers are not drawn to scale.)

Use the next page to present your implementation of **packetize**, which takes the address of the first byte of the full data image, the number of bytes in the image, and the size that each packet of data should be. You needn't worry about alignment restrictions.

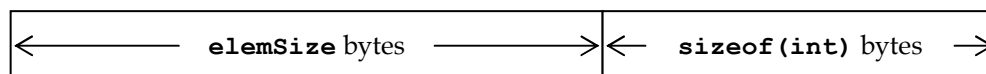
```
/**
 * Function: packetize
 * -----
 * Takes the specified block of data and builds a linked list
 * of nodes where each node is packetSize + sizeof(void *) bytes. The
 * first packetSize bytes of each node stores some portion of the original
 * image, and the final sizeof(void *) bytes store the address of the next
 * node in the packet list. The intent is that the sequence of bytes held
 * by all of the nodes is the same sequence of bytes held by the original
 * image.
 */

void *packetize(const void *image, int size, int packetSize)
{
```

### Problem 3: The C multiset

A **multiset** operates much like the **hashset** from Assignment 3, except that the **multiset** maintains a count on how many times each element has been inserted. The similarities between **hashset** and **multiset** are otherwise so great that it makes sense to layer the **multiset** right on top of the **hashset**.

Our **multiset** is fully generic, so the client should be able to use one to store any type he or she wants. But each item needs to be accompanied by its multiplicity. So each element is packed up against a **sizeof(int)**-byte integer and stored in the encapsulated **hashset**.



The difficult part here is the manual construction of these variably-sized figures so that they can be inserted into the encapsulated **hashset**.

Here's the reduced interface file for the **multiset**:

```
typedef int (*MultiSetHashFunction)(const void *elem, int numBuckets);
typedef int (*MultiSetCompareFunction)(const void *elem1, const void* elem2);
typedef void (*MultiSetMapFunction)(void *elem, int count, void *auxData);
typedef void (*MultiSetFreeFunction)(void *elem);

typedef struct {
    hashset elements;
    int elemSize;
    MultiSetFreeFunction free;
} multiset;

void MultiSetNew(multiset *ms, int elemSize, int numBuckets,
                MultiSetHashFunction hash, MultiSetCompareFunction compare,
                MultiSetFreeFunction free);
void MultiSetDispose(multiset *ms);
void MultiSetEnter(multiset *ms, const void *key);
void MultiSetMap(multiset *ms, MultiSetMapFunction map, void *auxData);
```

Your job is to implement all four functions, leveraging as much as possible off of the **hashset** routines implemented in Assignment 3.

Some additional information:

- You needn't worry about alignment restrictions.
- You don't need to bother with any error checking whatsoever.
- The manner in which client elements and their count get stored in the enclosed **hashset** must be consistent with the drawing specified above.
- **MultiSetHashFunctions** know how to hash client elements, not element/integer pairs. The client has no knowledge how things are stored behind the scenes.
- **MultiSetCompareFunctions** know how to compare two client elements. It's the comparison of the two client elements that guides the search through the encapsulated **hashset**.
- Since you're technically a client of the **hashset**, you shouldn't break the abstraction wall by accessing the **hashset**'s fields.

- a. Present implementations of **MultiSetNew** and **MultiSetDispose**. These should be short and straightforward.

```

typedef int (*MultiSetHashFunction)(const void *elem, int numBuckets);
typedef int (*MultiSetCompareFunction)(const void *elem1, const void* elem2);
typedef void (*MultiSetFreeFunction)(void *elem);

/**
 * Function: MultiSetNew
 * -----
 * Initializes the raw space addressed by ms to be an empty otherwise
 * capable of storing an arbitrarily large number of client elements of the
 * specified size. The numBuckets, hash, compare, and free parameters
 * are all supplied with the understanding that they'll be passed right
 * right through to HashSetNew. You should otherwise interact with the
 * embedded hashset using only those functions which have the authority
 * to access the hashset's fields.
 */

void MultiSetNew(multiset *ms, int elemSize, int numBuckets,
                MultiSetHashFunction hash, MultiSetCompareFunction compare,
                MultiSetFreeFunction free)
{

/**
 * Function: MultiSetDispose
 * -----
 * Disposes of all previously stored client elements by calling
 * HashSetDispose.
 */

void MultiSetDispose(multiset *ms)
{

```

- b. Now implement the more complicated **MultiSetEnter**, which ensures that the specified item is included in the **multiset**.

```
/**
 * Function: MultiSetEnter
 * -----
 * Ensures that the client element addressed by elem is included
 * in the multiset.  If the element isn't present, then the element
 * is inserted for the very first time and its multiplicity is registered
 * as 1.  If the element matches one already present, then the old one
 * is disposed of (if the free function is non-NULL), the new element
 * is replicated over the space of the one just disposed of, and
 * the multiplicity is incremented by 1.
 */

void MultiSetEnter(multiset *ms, const void *elem)
{
```

- c. Finally, implement **MultiSetMap**, which applies the specified **MultiSetMapFunction** to each element/count pair maintained by the **multiset**. Notice that **MultiSetMapFunctions** and **HashSetMapFunctions** have different signatures. You'll need to write a helper struct in order to manage the function mapping.

```
typedef void (*MultiSetMapFunction)(void *elem, int count, void *auxData);

/**
 * Function: MultiSetMap
 * -----
 * Applies the specified MultiSetMapFunction to every single element/int
 * pair maintained by the multiset, passing the supplied auxData argument as
 * the third argument to every single application.
 */

void MultiSetMap(multiset *ms, MultiSetMapFunction map, void *auxData)
{
```

#### Problem 4: The Queen Of Parking Infractions

Parking on campus has become so laughably difficult that some have resigned themselves to parking illegally and just paying whatever parking tickets they get. In the interest of crowning the Queen of Parking Infractions, you've been handed a fully constructed and populated **multiset** of license plate numbers, where the multiplicity is understood to be the number of parking tickets recorded for a particular license plate. Because all license plate numbers are strings of at most 7 characters, the **multiset** stores eight-byte figures, which are really static character arrays storing null-terminated C strings of length seven or less.

Your job is to complete the implementation of **FindQueenOfParkingInfractions**, which takes the address of a fully constructed **multiset** of license plates, and populates the specified character buffer with the license plate for which the maximum number of parking tickets has been recorded. You should use the convenience **struct** I've provided.

```
typedef struct {
    const char *licensePlate;
    int numTickets;
} maxTicketsStruct;

void FindQueenOfParkingInfractions(multiset *ms, char licensePlateOfQueen[])
{
```

#### Problem 5: Short Answers

For each of the following questions, we expect short, insightful answers, writing code or functions only when necessary. Clarity and accuracy of explanations are key.

- Assuming all instructions and pointers are 4 bytes, explain why instructions of the form  $\mathbf{M}[\mathbf{x}] = \mathbf{M}[\mathbf{y}] + \mathbf{M}[\mathbf{z}]$ , where  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mathbf{z}$  are legitimate but otherwise arbitrary memory addresses, aren't included in the instruction set.
- Our activation record model wedges the return address information below the function arguments and above the local parameters. Why not pack all variables together, and place this return address at the top or the bottom of the activation record?
- Write a short function called **IsLittleEndian**, which returns **true** if and only if the computer architecture is little endian – that is, if the least significant byte of a multi-byte figure is stored at the lowest address instead of the highest one.