CS107 Handout 25
Autumn 2000 October 23, 2000

CS107 Practice Midterm

Exam Facts

Monday, October 30th at 7 p.m.
Location: TCSeq200 (right next door to our normal lecture room.)

Any student with a conflict should **send me e-mail by tonight.** Be sure to tell me what other times you have available on Monday. After I have collected all the responses, I will e-mail those students later this week with the plan for the alternate. This midterm will be closed-book/closed-notes, but I will give you as much time as you need (within reason.) The practice exam was open-book/open-note, but was two hours long. My exam will be similiar in length and scope, covering the same key elements except those where memorization would now be needed because of my closed book/closed notes policy.

Material

The exam will focus on material like that of homework 1 and 2. Be prepared for C coding questions requiring strong understanding of pointers, arrays, function pointers, and low-level memory manipulation, as well as questions on code generation, activation records, variable layout, stack and heap implementation, and the compilation process like the work you did for HW2.

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 the midterm. fact, this is the exact exam given last fall, so it is a good example of what to expect. Some of our section problems have been taken from previous exams and are a useful source of study material as well.

You are encouraged to come to Thursday's section or any of our office hours with questions you have about these problems, the HW2 problems, or any other course material on which you need clarification. Good luck preparing!

Exam Instructions

The exam instructions will read something like this:

> This is an open-book, open-note exam. Details of syntax are not critical as long as it is clear what you mean. When asked to "draw the state of memory" please use the same format we did in class and on the handout examples: Draw the stack on the left, growing down and the heap on the right. Indicate the end of the stack by showing where the stack pointer, SP, is pointing. Label the fields in records and activation records. Label which procedure an activation record represents. Sit up straight. Draw pointers as arrows. Call your mother. For code generation, use the RISC assembly language and "R1 = R2 + R3" syntax. You may assume that integers and pointers are 4 bytes. When generating code, do not assume that the values of registers are preserved between snippets. You are strongly encouraged to comment your machine code, since it is basically unreadable in its native form. Comments are not required, but they help you receive partial credit if your intentions are correct but your code is wrong.

Problem 1: Code Generaton

You are to generate code for the following nonsense program. Don't be concerned about optimizing your instructions or conserving registers. We don't ask that you draw the activation records, but it will probably help you get the correct answer if you do.

```
struct ghost {
   int troll;
   struct ghost *gremlin[2];
};

static int Treat(struct ghost *goblin, char **witch);

static void Trick(char *bat[], struct ghost caspar)
{
     caspar.troll = Treat(caspar.gremlin[1], bat + 2);
}
```

a) Generate code for the entire `Trick` function.

```
static int Treat(struct ghost *goblin, char **witch)
{
   short candy[4];
   int spook;

   candy[spook] = **witch;                        // line 1
   *(((struct ghost *)candy)->gremlin) = NULL;  // line 2
   return *(int *)(&goblin + 5);                  // line 3
}
```

b) Generate code for the entire Treat function. Please mark your assembly code with the corresponding line numbers from the C source to aid us in understanding your work.

Problem 2: C Coding

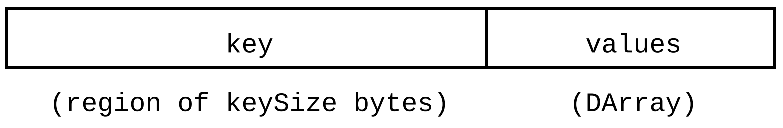
The HW1b HashTable manages a one-to-one mapping of keys to values. For this problem, you will implement a MultiTable variant for a one-to-many mapping where an entire set of values is associated with each key. A MultiTable could be used for a university database where the key was the class name and the associated values were the enrolled students. You could also use a MultiTable to map a token to the web pages containing that token, or from a grammar non-terminal to its list of productions, and so on. In short, it is a handy extension to the usual table.

In our tradition of building generic modular components, both the keys and the values of the MultiTable can be of any type. The ordinary HashTable lumps the key and value together, but the MultiTable separates them since it needs to operate on the key and value independently. The size of each type is specified when creating a new MultiTable and both key and value are referred to by address when entering and retrieving them. Just as for the HashTable, the client is responsible for providing functions that hash and compare the keys.

We provide you with the function that creates a new empty MultiTable, it will be your job to implement the Add and Map operations on such a table.

```
MultiTable MTNew(int keySize, int valueSize, int numBuckets,
                     TableHashFn keyHashFn, TableCompareFn keyCmpFn);
       void MTAdd(MultiTable mt, const void *key, const void *value);
       void MTMap(MultiTable mt, MTMapFn fn, void *clientData);
```

In implementing the MultiTable, you will be a client of both the HashTable and the DArray. Store the list of values for a key in a DArray, the ideal data structure for such an unbounded collection. Bundle together a key and its DArray of values and store in a HashTable so you can quickly lookup by key as needed. Thus, your MultiTable is a HashTable wherein each element is a chunk of memory that combines a key with its DArray of values, like this:



The operations you need to write are Add and Map. The Add operation enters a new key-value mapping. If the key is already in the table, you just append the new value to its DArray. If the key isn't in the table, enter a new element with the key and its DArray of a single value into the table. The Map operation calls a client-supplied function on every key-value pairing in the table.

The MultiTable should not re-implement the functionality of either the HashTable or DArray, but should be built on top of both. The DArray and HashTable prototypes are reproduced on the last page of the exam, refer to your assignment handouts for the full header files. The interface file for the MultiTable is given beginning on the next page.

A few comments before you start:

- Be sure you understand the data structure we require before writing any code. See drawing above and note how we configure things in MTNew, since you have to start from there.
- Read the header file very carefully. Your implementation must conform to the published interface, including handling of exceptional situations.
- As a client of the HashTable and DArray, make use of everything in the public interface, but nothing more. You should not recreate functionality already supplied by the provided routines and should not assume any implementor-level knowledge.
- We are going to ignore implementing any free operation for the MultiTable.
- You do not need to be concerned with any alignment restrictions.

```
/* File: multitable.h
 * ------------------
 * Defines the MultiTable ADT interface. This ADT manages a one-to-many mapping
 * between keys and values. The client specifies size (in bytes) of the key and
 * values being stored. All keys and values are referred to via void* pointers.
 */

/* Type: MultiTable
 * ------------------
 * Defines the MultiTable type itself. The struct declaration below is
 * "incomplete"- the details are literally not visible in the .h file.
 */
typedef struct MTImplementation *MultiTable;

/* MTNew
 * -----
 * Creates a new MultiTable with no elements and returns it. The keySize and
 * valueSize parameters specify the number of bytes that keys and values stored
 * in this table require. The numBuckets is used to partition during hashing.
 * The keyHashFn is used to hash the key. The keyCmpFn function is used for
 * testing equality of two keys. MTNew raises an asserts if keySize, valueSize,
 * or numBuckets is non-positive or if either function pointer is NULL.
 */
MultiTable MTNew(int keySize, int valueSize, int numBuckets,
                     TableHashFn keyHashFn, TableCompareFn keyCmpFn);

/* MTAdd
 * -----
 * Adds a value for the given key.  The contents of the value are copied from
 * the memory pointed to by the value parameter. If the key is not already in
 * the table, its contents are copied from the memory pointed to by the key
 * parameter. The function returns the count of values stored under this key
 * (including the one just added).
 */
int MTAdd(MultiTable mt, const void *key, const void *value);


/* MTMapFn
 * ----------
 * MTMapFn is the typedef for functions that can map over all the entries
 * in a MultiTable. The function is called for each pairing with a pointer
 * to the key, a pointer to the value, and the user’s clientData pointer.
 */
typedef void (*MTMapFn)(void *key, void *value, void *clientData);


/* MTMap
 * ------
 * Iterates through all key-value pairs (in any order) and calls fn
 * once for each distinct key-value entry that has been stored in the table.
 */
void MTMap(MultiTable mt, MTMapFn fn, void *clientData);
```

First go over the code on this page that we start you with. We provide the completed struct and the implementation of the `MTNew` function. Your ADT must work with this code without any changes. Because we are not implementing free for the MultiTable, we pass NULL as the element free function when creating a HashTable inside `MTNew`.

```
struct MTImplementation {
   HashTable elements;
   int keySize, valueSize;
};

MultiTable MTNew(int keySize, int valueSize, int numBuckets,
                     TableHashFn keyHashFn, TableCompareFn keyCmpFn)
{
   MultiTable mt;

   assert(keySize > 0 && valueSize > 0 && numBuckets > 0);
   assert(keyCmpFn != NULL && keyHashFn != NULL);

   mt = GetBlock(sizeof(*mt));
   mt->keySize = keySize;
   mt->valueSize = valueSize;
   mt->elements = TableNew(keySize + sizeof(DArray), numBuckets,
                             keyHashFn, keyCmpFn, NULL);
   return mt;
}

static void *GetBlock(int size)     // a utility routine you can use as needed
{
   void *ptr;

   ptr = malloc(size);
   assert(ptr != NULL);
   return ptr;
}
```

From here, your job is to implement the Add and Map functions to work properly with the ADT as designed above. You are free to write helpers as needed.

```
int MTAdd(MultiTable mt, const void *key, const void *value)
```

```
void MTMap(MultiTable mt, MTMapFn fn, void *clientData)
```

Problem 3: MultiTable ADT Client

Take the new MultiTable ADT out for a spin to store all the words in a file grouped by length. The function below should create a table for integer keys with string values, tokenize words from the file (using the Scanner, of course) and store each word into the table using the word length as its key (so all 3-letter words are under key 3, all 4-letter under 4, and so on). After the file has been scanned, map over the table and print out all the words and lengths and return the filled table.

Assume there is a completely working version of the MultiTable ADT at hand. Your job is to add the necessary client code as described and write any necessary helper functions. You can assume the `CopyString` helper function exists if you need it.

```
MultiTable ScanIntoTable(Scanner scanner)
 {
   char word[1024];
   MultiTable mt;
         create new MultiTable
         use 20 buckets
         assume no fixed length
         when storing words

   while (ReadNextToken(scanner, word, sizeof(word)) {

         add word to table
         use length as key









   }
         print contents of table
         one per line, word & len
              apple 5

   return mt;
 }
```

Probelm 4: Short answer

At most 1-2 sentences please! These questions are purposely not assigned many points, so don't burn a lot time here that might be better spent on the more important coding questions.

· For each of the following three cases, indicate Yes / No if the ++ operation is allowed at compile-time.

```
void Winky(char c[10])
{
   c++;                 // 1) allowed?
}

void Binky()
{
   char a[10], *b;

   b = a;
   a++;                 // 2) allowed?
   b++;                 // 3) allowed?
   Winky(a);
}
```

a) Write a function `SafeRealloc` that takes a currently allocated pointer and a new size and attempts to reallocate the pointer to the new size or leaves it unchanged if the request cannot be satisfied. The function should return a boolean result of whether the resizing was successful.

b) Consider an attempt to compile and link the following **entire** program in the usual manner for a single file:

```
#include <stdio.h> // has prototype for printf, nothing else relevant

int strlen(int *ptr, int val);

int main(int argc, char *argv[])
{
    int num = 1;

    printf("Result is %d\n", strlen(&num, num));
    return 0;
}
```

Identify any problems that will be detected during the compilation process. For each problem, indicate whether it is a fatal error or just a warning and which tool (preprocessor, compiler, linker) reports it. If you believe the compilation succeeds without any fatal errors, describe what happens when you execute the program.

c) Write a function that returns a boolean result of whether the bit pattern representing floating point 0.0 is the same as the bit pattern of integer 0 for the architecture on which it is executed. If integers and floats are not the same size on this machine, the function should return false.

d) The most insidious errors are those that go unnoticed by the compiler and linker, but deliver nasty runtime surprises. Assume the fragment below compiles and links cleanly in some larger context, despite the fact that it contains a lurking runtime land mine. Identify the bug and explain what would be required to correct it. If you cannot spot the bug, leaving the question blank will be worth more than an answer that makes incorrect assertions.

```
void DoStrings(void)
{
   int i;
   char *s[4] = {"miney", "meenie", "eenie", "mo"};
   DArray strings;

   strings = ArrayNew(sizeof(char *), 4, free);
   for (i = 0; i < 4; i++) {
       ArrayAppend(strings, &(s[i]));
   }
   ArrayFree(strings);
}
```