Practice Midterm Solution

Problem 1: Code Generation

a)

```
   R1 = M[SP + 16]          ; load caspar.gremlin[1]
   R2 = M[SP + 4]           ; load bat
   R3 = R2 + 8              ; add 2 scaled by sizeof(char *)
   SP = SP - 8              ; make space for params to Treat
   M[SP + 4] = R3           ; assign 2nd param
   M[SP] = R1               ; assign 1st param
   CALL <Treat>
   SP = SP + 8              ; clean up params
   M[SP + 8] = RV           ; copy return value in caspar.troll
   RET
```

b)

```
   SP = SP - 12             ; make space for locals

candy[spook] = **witch;

   R1 = M[SP + 20]          ; load value of witch
   R2 = M[R1]               ; deref to load *witch
   R3 =.1 M[R2]             ; deref again to load **witch
   R4 = M[SP + 8]           ; load spook
   R5 = R4 * 2;             ; multiply offset by sizeof (short)
   R6 = SP + R5             ; add offset to base of array (which is SP)
   M[R6] =.2 R3             ; store short at that position

*(((struct ghost *)candy)->gremlin) = NULL;

   M[SP + 4] = 0            ; treat candy as ghost *, access gremlin

return *(int *)(&goblin + 5);

   R1 = SP + 16             ; compute address of goblin
   R2 = R1 + 20             ; add 5 scaled by sizeof(struct ghost)
   RV = M[R2]               ; deref to get int and store as return val
   SP = SP + 12             ; clean up locals
   RET                      ; return
```

Problem 2: C Coding

MTAdd first looks up the key to see if it already exists. In TableLookup, the client's hash and compare are called to find a matching element. Because we put the key first in the elements, the client's functions still work, even though they don't realize a DArray follows the key in memory.

If the key isn't in the table, we build a chunk of the key and a new DArray. Because we don't know the size of the key at compile-time, we must dynamically allocate. We copy the key from the client's pointer into the chunk then assign a new DArray into the location after the key. We enter the element in the Hashtable and free the storage, since the contents were copied into the table.

At this point we have a DArray of values, either the newly created one, or the one found under a pre-existing key, and we simply append the latest value and return the new count.

```
int MTAdd(MultiTable mt, const void *key, const void *value)
{
   void *foundElem, *newElem;
   DArray values;

   foundElem = TableLookup(mt->elements, key);
   if (foundElem == NULL) {   // need to create new entry
      newElem = GetBlock(mt->keySize + sizeof(DArray));
      memcpy(newElem, key, mt->keySize);
      values = ArrayNew(mt->valueSize, 10, NULL);
      *(DArray *)((char *)newElem + mt->keySize) = values;
      TableEnter(mt->elements, newElem);
      free(newElem);
   } else {                          // key already exists, just append value
      values = *(DArray *)((char *)foundElem + mt->keySize);
    }
    ArrayAppend(values, value);
    return ArrayLength(values);
}
```

Doing MTMap required a little bit of careful thought. You need to translate from one format to another which required using the clientData pointer of the original TableMap to pass three things, the original client mapping function, the original clientData, and the keySize (so that we can pick apart the element into the key and values DArray). The easiest way to do this is to define a simple struct and pass a pointer to it.

```
struct mapData {
    int keySize;
    MTMapFn clientFn;
    void *clientData;
};
```

```
void MTMap(MultiTable mt, MTMapFn fn, void *clientData)
{
     struct mapData data;

     data.keySize = mt->keySize;
     data.clientFn = fn;
     data.clientData = clientData;
     TableMap(mt->elements, MapAllValues, &data);
}

static void MapAllValues(void *elem, void *mapData)
{
    struct mapData *md = (struct mapData *)mapData;
    DArray values;
    int i;

    values = *(DArray *)((char *)elem + md->keySize);
    for (i = 0; i < ArrayLength(values); i++) // could ArrayMap here
        md->clientFn(elem, ArrayNth(values, i), md->clientData);
}
```

Problem 3: ADT Client

```
MultiTable ScanIntoTable(Scanner scanner)
{
   char word[1024], *heapCopy;
   int len;
   MultiTable mt;

   mt = MTNew(sizeof(int), sizeof(char *), 20, HashInt, CompareInt);
   while (ReadNextToken(scanner, word, sizeof(word))) {
      heapCopy = CopyString(word);
      len = strlen(word);
      MTAdd(mt, &len, &heapCopy);
   }
   MTMap(mt, PrintEntry, NULL);
   return mt;
}


static int HashInt(const void *elem, int numBuckets)
{
   return *(int *)elem % numBuckets;
}

static int CompareInt(const void *elem1, const void *elem2)
{
   return *(int *)elem1 - *(int *)elem2;
}

static void PrintEntry(void *key, void *value, void *cData)
{
   printf("%s %d\n",*(char **)value, *(int *)key);
}
```

Problem 4: Short answer

- Both 1 and 3 are fine, but 2 is not, since an array name is not an assignable L-value. Remember that arrays as always passed as pointers when used as parameters, no matter what the declaration may lead you to believe.

- One thing to be careful with here is that the original pointer must be passed by reference if it is going to potentially be changed during reallocation (this happens when the block cannot stretch in its correct location).

```
bool SafeRealloc(void **ptr, int newSize)
{
   void *result = realloc(*ptr, newSize);
   if (result == NULL)
      return false;
   else {
      *ptr = result;
      return true;
   }
}
```

- The code creates its own (erroneous) prototype for the `strlen` function. However, it doesn't include the `<string.h>` header file that has the correct prototype, so there is no clash about the declaration. Since the use of the function is consistent with the prototype given beforehand, the compiler is completely content. You always link with the standard libraries by default but since linker has no type information, it will match up the unresolved `strlen` symbol to the `strlen` implementation in the standard libraries. So it compiles and links without trouble.

  When executing, the generated code pushes two arguments on the stack and jumps to the generated code for `strlen` which only expects there to be one. The extra argument doesn't cause any trouble, since the code won't access the contents beyond the first. However, the first parameter passed was actually pointing to an integer, not a sequence of characters, as `strlen` expects. The function can't tell we've made this mistake, so it just starts examining the bytes at that location until it hits the null that is interpreted as the string terminator. For the integer value 1 on a big-endian machine, the first byte will be zero and thus `strlen` returns zero and that is what is printed. On a little-endian, it goes past the first byte to find a zero in the second, and returns and prints 1. This was a subtle and tricky question. Few people were totally correct, but several were on the right track.

- There are a couple of different ways to go about this. The key is realizing that you want to look at the bits without converting them – as soon as you convert the float to int or vice versa, you are no longer computing whether the original bit patterns are the same. It turns out that all bit operations (&, |, ^) require operands of type of unsigned int and so attempting to use one on a float will first convert it. Given I didn't expect you would know that, I counted answers using bit operators as correct as long as the logic in the use of the bit operators would work out.

```
bool AreSame(void)
{
   int i = 0;
   float f = 0.0;
```

```
    return ((sizeof(int) == sizeof(float)) && i == *(int *)&f);
    // or compare with memcmp(&i, &f, sizeof(i) == 0);
    // could use loop to compare char-by-char, too
    }
}
```

- As you should be well aware of by now, the DArray always refers to elements by address, so you pass pointers to elements to `ArrayAppend`, get pointers to elements from `ArrayNth`, and pointers to elements are passed as arguments to the various callback functions. This is always true even if the element itself is already a pointer.

  The bug was the use of `free`. The standard `free` function can never be correct as the element free function, no matter what is stored in the DArray, since it is guaranteed that a wrapper is needed to dereference the void * pointer element to free its contents. As it stands, `free` will attempt to free pointers into the Darray's storage and that is bad news. The fix in this case is not to provide the necessary wrapper, but use NULL for the free function since the strings being stored are string constants. They didn't come from the heap and thus don't need to be deallocated.