Section Solutions 3: Files and Strings

1) String Implementation Redux.

```
string SubString(string s, int start, int stop)
{
   int length, i;
   string result;

   length = strlen(s);
   if (start < 0)      // if start before beginning of string, reset
         start = 0;
   if (stop > length - 1)  // if stop past end, reset to end
         stop = length - 1;
   if (start > stop)       // if start past stop, return empty string
         return "";

   result = (string)GetBlock(stop - start +2);//alloc space, include null
   for (i = 0; i <= stop - start; i++)
         result[i] = s[start + i];
   result[i] = '\0';                        // put null at end of string
   return result;
}
```

2) Quick Data Structure Question.

We will get to talk about this a bit more in lecture, but here are some things to consider:

Static array: + No dynamic allocation, don't have to work with pointers, can grow/shrink number of elements used (within bounds of the array)
- The fixed upper bound means potentially too small or too large, can waste a lot of space or limit utility value

Dynamic array: + Exactly the size you need, as little or as large
- Have to remember to allocate & thus must know size in advance, can't grow/shrink easily once allocated

Static array of ptrs: + Moderately conservative in use of memory, can grow/shrink within bounds of array, pointers are easier to swap and move around inside array
- Lots of allocations means lots of opportunities to forget, still have fixed upper limit problems, forces you to deal with pointers

Any decision among these three choices should consider such factors as the size of the structure itself, how large the variance is in the number of elements needed, how comfortable you feel about working with pointers, whether this is a known upper bound that is never exceeded, whether you need to grow and shrink the array, and so on.

3) Structures.

```
typedef enum {engineering, humanities, science, business, law,
              medical} schoolType;

typedef struct {
   string      title;
   int         units;
   string      instructor;
   bool        pncGrading;
   schoolType  school;
} courseInfo;

typedef struct {
   courseInfo  *course;
   bool        takingForGrade;
} courseTaken;

typedef struct {
   string      name;
   long        id;
   int         numCourses;
   courseTaken courses[STUDENT_MAX_COURSES];
} studentInfo;

typedef struct {
   courseInfo  *arr[MAX_COURSES];      // arr of ptrs to courseInfo
structs
   int         numCourses;
} courseList;

typedef struct {
   studentInfo *arr[MAX_STUDENTS];     // arr of ptrs to studentInfo
struct
   int         numStudents;
} studentList;

typedef struct {
   studentList students;
   courseList courses;
} schoolDB;



int FindUnits(schoolDB *db, string studentName)
{
   int units;
   studentInfo *student;
   int i;

   student = FindStudent(db->students, studentName);
   if (student == NULL) Error("Student name not in database");

   units = 0;
   for (i = 0; i < student->numCourses; i++)
         if (student->courses[i].takingForGrade)
               units += student->courses[i].course->units;
   return units;
```

```
}


/* Function: FindStudent
 * ------------------------
 * Returns a studentInfo pointer if the student with given name is
in
 * the database, otherwise returns NULL.  This sort is a binary
search,
 * and requires that the database be in sorted (by name) order.
 */
static studentInfo *FindStudent(studentList students, string name)
{
   int left, right, mid;

   left = 0;
   right = students.numStudents -1;

   while (left <= right) {
      mid = (left + right) / 2;
      if (StringEqual(name, students.arr[mid]->name)) {
         return students.arr[mid];
      } else if (StringCompare(name, students.arr[mid]->name) < 0) {
         right = mid - 1;
      } else {
         left = mid + 1;
      }
   }
   return NULL;
}
```

4) Structure & Pointer Craziness.

*The problem is that each CDtype variable contains a dynamic array of strings, which is actually a **pointer**. So when we copy CD1 into CD2, everything gets copied exactly, including the base address of the array, which means that **both CD1 and CD2 are pointing to the same array of trackTitles**. In changing the title of the first song on CD2, we will inadvertently change the title for CD1 also. In order to create a copy of a CDtype which is entirely distinct, we really need to allocate memory for the new trackTitle array, and then copy each title individually. (This structure copy operation would make a really good function!)*

5) Super Quick File Questions.

a) *The problem is that there is no memory allocated for result, and thus the fscanf call writes into some unknown space. One possible way to fix it is as follows:*

```
string GetNameFromFile(FILE *infile)
{
   char buffer[80];
   fscanf(infile, "%s", buffer);

   // make a copy in heap we can return
   return CopyString(buffer); }
```

To ensure we don't overrun the buffer's fixed size, we can add a field width to the string specifier in the fscanf call that will at most read 79 characters (leaving space for NULL char at end).

```
   fscanf(infile, "%79s", buffer);
```

b)
```
PlayerRec *GetBaseballPlayer(FILE *infile)
{
   PlayerRec *newPlayer;

   newPlayer = New(PlayerRec *);
   fscanf(infile, "Hits: %d AtBats: %d Walks %d", &newPlayer->hits,
               &newPlayer->atBats, &newPlayer->walks);
   return newPlayer;
}
```

6) Files, files, files.

```
            typedef struct {
   string proprietor;
   double amount;
} chargeRec;

typedef struct {
   string name;
   int numCharges;
   chargeRec *chargeArray; // dynamically-allocated array of charges
} cardHolder;

typedef struct {
   int numCardHolders;
   cardHolder *cardHolderArray[MAX_RECORDS]; // array of ptr to
cardHolders
} billingDB;
```

```
chargeRec ReadCharge(FILE *infile)
{
   chargeRec charge;

   charge.proprietor = ReadLine(infile);
   fscanf(infile, "%g\n", &charge.amount);
   return charge;
}

cardHolder *ReadCardHolder(FILE *infile)
{
   string line;
   int i;
   cardHolder *client;

   line = ReadLine(infile);
   if (StringEqual(line, "")) return NULL;
   client = New(cardHolder *);
   client->name = line;
   fscanf(infile, "%d\n", &client->numCharges);

/* dynamically allocate the charge array now that we know the size
*/

   client->chargeArray = (chargeRec *) GetBlock(sizeof(chargeRec)

   *client->numCharges);
   for (i = 0; i < client->numCharges; i++)
         client->chargeArray[i] = ReadCharge(infile);
   return client;
}


billingDB *ReadBillingDB(FILE *infile)
{
   int i;
   cardHolder *client;
   billingDB *db;

   db = New(billingDB *);
   db->numCardHolders = 0;
   for (i = 0; i < MAX_RECORDS; i++) {
         client = ReadCardHolder(infile);
         if (client == NULL) break;          // no more clients
         db->cardHolderArray[i] = client;
         db->numCardHolders++;
   }
   return db;
}
```

This data structure has lots of storage to free: all the strings in the charge arrays, the charge arrays themselves, the card holder records, etc. Be sure to free things in the correct order (free things pointed to out of structures before freeing those structures themselves)

```
 void FreeCardHolder(cardHolder *client)
 {
    int i;

    FreeBlock(client->name);
    for (i = 0; i < client->numCharges; i++) // free string in each
 charge
          FreeBlock(client->chargeArray[i].proprietor);
    FreeBlock(client->chargeArray);           // free dynamic array
 itself
    FreeBlock(client);            // free cardHolder structure
 }

 void FreeBillingDB(billingDB *db)
 {
    int i;

    for (i = 0; i < db->numCardHolders; i++)
          FreeCardHolder(db->cardHolderArray[i]);
    FreeBlock(db);
}
```