Midterm Practice

Midterm: Monday, May 1st 7-9 pm
Gates B01(next door to our usual room)

We'd be most appreciative if all of you could attend the regular exam, but if you have an unavoidable conflict, please e-mail Andrew (atoy@cs) by 5pm on Wednesday April 26 in order to make arrangements for an alternate exam on Monday.

Coverage

The midterm will cover all of the 106A material (the first 3 chapters of the text), along with linked lists. Recursion will not be on the midterm. It is a 2-hour exam, open-book /open-notes, no computers allowed!

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. This is, in fact, the exact exam I gave the last time I taught CS106X. You are highly encouraged to work through the problems in test-like conditions to prepare yourself for the actual exam. Some of our section problems have been taken from previous exams and chapter exercises from the text often make appearances in same or similar forms on exams, so both of those resources are a valuable source of study material as well.

Note: To conserve trees, I have cut back on answer space. The actual exam will have much more room for your answers and for any scratch work.

General instructions that will given on the real exam

Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

When writing programs for the exam, you do not need to be concerned with `#include` lines, just assume any of the libraries that you need (both standard C or 106-specific) are already available to you. If you would like to use a function from a handout or textbook, you do not need to repeat the definition on the exam, just give the name of the function and the handout or chapter number in which its definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Problem 1: Strings Revealed

Eric Roberts wants **you** to implement a new function for his string library, to be added to the "strlib.h" interface. The prototype for this new function is:

```
string Weave(string s1, string s2);
```

This function takes two strings and returns a third string composed of the characters from the first two strings woven together. The first character of the new string is the first character of the first string, the second character of the new string is the first character of the second string, the third character of the new string is the second character of the first string, and so on. Once the characters from the shorter string are all in the result string, the remaining characters of the longer string are appended to the new string.

Here are some examples:

s1	s2	Weave(s1,s2)
"Hello"	"World"	"HWeolrllod"
"banana"	"Go"	"bGaonana"
"why"	"not!"	"wnhoyt!"

Since you are implementing this function as part of "strlib.h" (and because we want to test your knowledge of the underlying representation of strings,) **you must not using any functions from "strlib.h" (e.g. IthChar, FindString, Concat) in your implementation**. You may use any of the functions from the ANSI string library <string.h>. Also note that like other functions in "strlib.h", Weave should not change the two strings passed as arguments in any way.

Problem 2: Linked Lists

Write the function RemoveOddNumbers that takes a linked list of numbers using the cell definition below and destructively modifies the list to remove and free all odd numbers in the list. The remaining list entries should contain the even numbers in the same order they were previously.

```
typedef struct __cell {
   int value;
   struct __cell *next;
} Cell;


void RemoveOddNumbers(Cell *head)
```

Problem 3: Pointers and Program Tracing

For the following code segment, you are asked to draw the state of the computer's memory twice during code execution. Trace through, starting from `main` and stop at the first point marked before the call to `Oliver` and draw the state of memory. Then execute the call to `Oliver` and draw the state of memory afterwards. Differentiate between stack memory and heap memory, label the names of variables, and indicate whenever memory is orphaned.

```
typedef enum {
   Mike, Carol, Alice, Sam
} BradyAdult;

typedef struct {
   string greg, marsha[4] *peter[4];
   int jan, *bobby, cindy;
} BradyKids;

main()
{
   BradyKids   bunch;
   BradyAdult  parent;

   bunch.marsha[0] = CopyString("Marsha, ");
   bunch.marsha[1] = bunch.marsha[0];
   bunch.marsha[2] = CopyString("Marsha!");
   bunch.marsha[3] = (string) GetBlock(strlen(bunch.marsha[2]));

   for (parent = Mike; parent < Sam; parent++) {
      bunch.peter[parent] = bunch.marsha + parent + 1;
      strcpy(*bunch.peter[parent], "Mart");
   }
   bunch.bobby = &bunch.jan;
                          <—Draw state of memory before call
   Oliver(bunch, bunch.peter);
}                         <—And draw state after call


void Oliver(BradyKids bunchCopy, string **marshaPtr)
{
   bunchCopy.jan = 7;
   *bunchCopy.bobby = 3;
   *(bunchCopy.marsha[1] + 4) = 'h';
   *bunchCopy.peter[0] = Concat(*bunchCopy.peter[0], "Bertha!");
   *marshaPtr -= 1;
   sprintf(bunchCopy.marsha[3], "%s%d","Wilma", bunchCopy.jan);
}
```

Problem 4: Data structures

You are given the following constant definitions and data structure declarations:

```
#define MAX_RECIPES 10
#define MAX_PEOPLE 100

#define NUM_DISHES 5
typedef enum {Salad, Casserole, Vegetable, Bread, Dessert} DishType;

typedef struct {
      string name;                  /* name of ingredient i.e. "Butter" */
      int quantity;                 /* amount in some generic unit */
} Ingredient;

typedef struct {
      string name;                  /* name of recipe:  "Lasagna" */
      DishType type;                /* type of dish: Bread, Salad, etc. */
      Ingredient *ingreds;          /* dynamic array of Ingred structs */
      int numIngreds;               /* num entries in ingredient array */
} Recipe;

typedef struct {
      string name;                  /* name of person */
      Recipe *recipes[MAX_RECIPES]; /* array of pointers to recipes we
like */
      int numRecipes;               /* num entries in recipe array */
      Ingredient *pantry;           /* dynamic array of Ingred structs */
      int numPantry;                /* num entries in pantry array */
} Person;

typedef struct {
      Person *chef;                 /* person preparing this dish */
      Recipe *recipe;               /* recipe for the dish */
} Dish;

typedef struct {
      Person attendees[MAX_PEOPLE]; /* array of Person structs */
      int numAttending;             /* num entries in attendees array */
      Dish menu[NUM_DISHES];        /* assignments of people to dishes */
} Potluck;
```

You are trying to plan a potluck dinner. You want to have exactly one dish of each of the five types (Salad, Casserole, etc.), so you need to recruit five attendees to whip up a dish for the dinner. The catch is that you want to make it easy on your volunteers and only ask them to make a recipe for which they already have all the ingredients. Good thing your data structure includes a list of all the ingredients each person has in their pantry as well as all the ingredient information for each recipe. Matching them up should be right up your alley!

Take careful stock of the data structure definitions given to you, in particular, take note of where you have pointers and where you have structures.

We have already decomposed the problem into four parts which are described below. You can assume all data is well-formed (i.e. you do not need to do any error-checking).

We'll start you with the FindIngredient function.
This is a lookup routine that takes an ingredient name, expressed as a string, and searches an array of ingredient structs for an ingredient of that name. The function returns the pointer to the ingredient struct if found and NULL if no such ingredient exists in the array.

```
Ingredient *FindIngredient(Ingredient ingreds[], int count, string
name)
{
   int i;

   for (i = 0; i < count; i++) {
      if (StringEqual(ingreds[i].name, name))
         return &(ingreds[i]);
   }
   return NULL;
}
```

(4a) Write the HasIngredients predicate.
This function will determine if a person has the resources to make a particular recipe. The function parameters are a pointer to the person and a pointer to the recipe. Check the person's pantry list to see if they have sufficient quantities of all the ingredients required by the recipe. Don't forget you have the above `FindIngredient` routine to search the person's pantry list for an ingredient by name. You will return a boolean value to indicate whether the person has enough of all the necessary ingredients.

```
bool HasIngredients(Person *person, Recipe *recipe)
```

(4b) Write the FindRecipeOfType function.
This function will determine if a given person can be assigned responsibility for bringing a particular type of dish to the potluck. Search the person's list of favorite recipes to find one of the desired type that they have the ingredients for. If you successfully find such a recipe, return a pointer to it, otherwise return NULL to indicate failure.

```
Recipe *FindRecipeOfType(Person *person, DishType type)
```

(4c) Write the ArrangePotluck function.
This function is passed a pointer to a Potluck structure and will assign responsibilities for the five types of dishes on the menu by signing up a particular person with their recipe. For each type of dish, search the list of attendees for the first person you find who can make a dish of that type. When you've got one, assign that person and recipe to the entry for that dish in the menu array. Don't worry if a person is asked to bring more than one dish. Return TRUE if you were able to assign volunteers for all the dishes and FALSE if you were not.

```
bool ArrangePotluck(Potluck *dinner)
```

Problem 5: File Processing

Using the data structure from Problem 4, you are going to write the function `ReadOneRecipe` which will read one recipe's information from a file and return a pointer to a newly allocated record containing the information for the recipe just read in or NULL if at end of the file.

Assume the file is open and ready for reading. The datafile is in the following format:

```
Chocolate Chip Cookies          <—recipe name
Type: 4 Num Ingredients: 2      <— type of dish & num ingredients
Eggs                            <—name of ingredient
4                               <—quantity of ingredient
Butter
2
```

The dish type is represented using the integer values of the `DishType` enum declared earlier. Assume the file is correctly structured (i.e. you don't have to do error-checking on the file format). If at the time `ReadOneRecipe` is called, it is at the end of file with no more recipes to read (i.e. the first line is blank), the function should return NULL. You do not need to decompose your solution unless you find it helpful for your own use.

```
Recipe *ReadOneRecipe(FILE *in)
```