Spring, 2017 Name: _____

(Please *don't* write your id number!)

Exam 4: C Programming with Structs and Pointers

Directions

This exam is closed book and notes.
If you need more space, use the back of a page. Note when you do that on the front.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Clarity is important; if your answers are sloppy and hard to read, you may lose some points.

For Grading

Question:	1	2	3	Total
Points:	10	40	50	100
Score:				

1. (10 points) [Programming] In C, define the function specified in the following header file:

```
// $Id: bag_count.h,v 1.3 2017/04/25 21:12:23 leavens Exp leavens $
#ifndef BAG_COUNT
#define BAG_COUNT 1
#include <stdlib.h>
#include <string.h>
typedef struct bag_s {
    int size;
    char **elems; // an array of strings, but C doesn't like char *elems[] here
} bag_t;
typedef bag_t *bag;
// requires: sought is allocated (and not NULL) and is a null-terminated string,
//           also b is allocated (and not NULL) and is such that b->elems has
//           at least b->size elements, each of which is allocated (and not NULL)
//           and is a null-terminated string.
// ensures: result is the number of times that a string equal to sought occurs
//          as an element of b.
extern int bag_count(char *sought, bag b);
#endif
```

that takes a string `sought` and a `bag` `b`, and returns the number of times that `sought` occurs as an element in the array referred to by the `elems` field of `b`. Tests for this problem appear below.

```
// $Id: test_bag_count.c,v 1.2 2017/04/25 20:57:02 leavens Exp $
#include <stdlib.h>
#include <string.h>
#include "tap.h"
#include "bag_count.h"
int main() {
    char *c_s = "cooked";
    int len = strlen(c_s);
    // make cooked be a string that is not pre-allocated
    char *cooked = (char *)malloc(len+1);
    strcpy(cooked, c_s);   // now cooked contains "cooked"
    bag testbag = (bag)malloc(sizeof(bag_t));
    if (testbag == NULL) { return 1; }
    char **testelems = (char **)malloc(sizeof(char *)*10);
    if (testbag == NULL) { return 1; }
    testbag->size = 10;
    testbag->elems = testelems;
    testbag->elems[0] = c_s;              testbag->elems[1] = cooked;
    testbag->elems[2] = "raw";            testbag->elems[3] = "soup";
    testbag->elems[4] = cooked;           testbag->elems[5] = "raw";
    testbag->elems[6] = "Zaphod";         testbag->elems[7] = c_s;
    testbag->elems[8] = "clams, stewed";  testbag->elems[9] = "stewed";
    ok(bag_count(c_s, testbag) == 4);
    ok(bag_count(cooked, testbag) == 4);
    ok(bag_count("raw", testbag) == 2);
    ok(bag_count("stewed", testbag) == 1);
    ok(bag_count("clams, stewed", testbag) == 1);
    ok(bag_count("fried", testbag) == 0);
    bag bag1 = (bag)malloc(sizeof(bag_t));
    char **elems1 = (char **)malloc(sizeof(char *));
    bag1->elems = elems1;
    bag1->elems[0] = cooked;
    bag1->size = 1;
    ok(bag_count("cooked", bag1) == 1);
    return exit_status();
}
```

Please put your code for `bag_count` below.

2. (40 points) [Programming] In C, define the functions specified in the following header file:

```
// $Id: complex.h,v 1.2 2017/04/25 20:57:02 leavens Exp $
#ifndef COMPLEX_H
#define COMPLEX_H 1
#include <stdlib.h>
#include <stdbool.h>
typedef struct complex_s {
    double real;
    double imaginary;
} complex_t;

typedef complex_t *complex;

// requires: neither a nor b is not-a-number
//           and malloc has enough space to succeed in allocating a complex_t struct.
// ensures: result is non-NULL and allocated with its real field equal to a and its
//          imaginary field equal to b.
extern complex make_complex(double a, double b);

// requires: c is allocated (and not NULL)
// ensures: result is the value of the real field of c
extern double complex_real_part(complex c);

// requires: c is allocated (and not NULL)
// ensures: result is the value of the imaginary field of c
extern double complex_imaginary_part(complex c);

// requires: both c and d are allocated (and not NULL)
//           and malloc has enough space to succeed in allocating a complex_t struct
// ensures: result is the sum of c and d, that is
//          result's real field is the sum of c and d's real fields,
//          and result's imaginary field is the sum of c and d's imaginary fields.
extern complex add_complex(complex c, complex d);

// requires: both c and d are allocated (and not NULL)
// ensures: result is true just when the real fields of c and d are equal
//          and the imaginary fields of c and d are also equal.
extern bool equal_complex(complex c, complex d);
#endif
```

These functions create, access, and manipulate complex numbers. The function `make_complex` creates the complex number that is written mathematically as a+bi from the arguments `a` and `b`, so that `a` is the real part and `b` is the imaginary part. The function `complex_real_part` returns the real part of a complex number (i.e., it returns `a` from a+bi). Similarly the function `complex_imaginary_part` returns the imaginary part of a complex number (i.e., it returns `b` from a+bi). The function `add_complex` returns the sum of its arguments; if the arguments represent the mathematical complex numbers a+bi and x+yi then the result is mathematically (a+x) + (b + y)i. The function `equal_complex` returns true just when the two arguments have the same real and imaginary parts. Tests for this problem appear below.

```
#include "tap.h"
#include "complex.h"
int main() {
    complex one = make_complex(1.0, 0.0);
    ok(one != NULL);
    ok(complex_real_part(one) == 1.0 && complex_imaginary_part(one) == 0.0);
    ok(equal_complex(one, one));
    complex i = make_complex(0.0, 1.0);
    ok(i != NULL);
```

```
    ok(complex_real_part(i) == 0.0 && complex_imaginary_part(i) == 1.0);
    ok(equal_complex(i, i));
    ok(!equal_complex(one, i));
    complex oneplusi = add_complex(one, i);
    complex iplusone = add_complex(i, one);
    ok(complex_real_part(one) == 1.0, "one's real part is unchanged");
    ok(complex_imaginary_part(one) == 0.0, "one's imag part is unchanged");
    ok(complex_real_part(i) == 0.0, "i's real part is unchanged");
    ok(complex_imaginary_part(i) == 1.0, "i's imaginary part is unchanged");
    ok(equal_complex(oneplusi, iplusone), "addition is commutative");
    ok(complex_real_part(oneplusi) == 1.0 && complex_imaginary_part(iplusone) == 1.0);
    complex twoi = add_complex(oneplusi, iplusone);
    ok(complex_real_part(twoi) == 2.0 && complex_imaginary_part(twoi) == 2.0);
    return exit_status();
}
```

There is space for your answer below and on the next page.

3. (50 points) [Programming] In C, define the functions specified in the following header file:

```
// $Id: car.h,v 1.2 2017/04/25 21:12:23 leavens Exp leavens $
#ifndef CAR_H
#define CAR_H 1
#include <stdlib.h>
#include <stdio.h>
// maximum length of names for make and model of cars
#define MAX_NAME_LEN 40
// format for reading names with scanf
#define NAME_FORMAT "%40s"

typedef struct car_s {
    char *make;
    char *model;
    int year;
} car_t;

typedef car_t *car;

// requires: manuf and mod are allocated (and not-NULL)
//           null-terminated strings and yr >= 1900 and malloc
//           has enough space to allocate a new car_t struct
// ensures: result is not NULL and points to a newly allocated
//          car_t struct with its make field equal to manuf,
//          its model field equal to mod, and its year field equal to yr.
extern car make_car(char *manuf, char *mod, int yr);

// requires: the make_prompt, model_prompt, and year_prompt arguments are
//           all allocated (and not NULL) and null-terminated strings,
//           stdin and stdout are open
//           and malloc has enough space to allocate a new car_t struct
// effect: prompts with make_prompt on stdout, reads the car's make from stdin,
//         then prompts with model_prompt on stdout, and reads the car's model
//         from stdin, then prompts with year_prompt, and reads the car's year
//         (as 4 digits) from stdin. If the year read is not at least 1900,
//         then print a message (on stdout) of the form
//         "You entered NNNN, please enter a year after 1899"
//         where NNNN is the year the user entered, and
//         then reprompt with year_prompt (on stdout) and read the year again
//         from stdin, until the year read is at least 1900.
//         Then creates a new car_t struct
//         and returns a pointer to it (which is not NULL).
//         The fields of the result, the new car_t struct, are
//         initialized to the values read from stdin; that is the make field
//         is equal to the make read from stdin, the model field is equal to
//         the model read from stdin, and the year field is equal to the year
//         read from stdin.
extern car read_car(const char *make_prompt, const char *model_prompt,
                    const char *year_prompt);

// requires: c is allocated and not NULL
// ensures: result is the value of c's make field
extern const char *car_make(car c);

// requires: c is allocated and not NULL
// ensures: result is the value of c's model field
extern const char *car_model(car c);
```

```
// requires: c is allocated and not NULL
// ensures: result is the value of c's year field
extern const int car_year(car c);

// requires: c is allocated and not NULL and malloc has enough space to
//           hold the result
// ensures: result is a string of the form "YEAR MAKE MODEL", where YEAR is
//          the value of c's year field, MAKE is the value in c's make field
//          and MODEL is the value in c's model field.
//          Result is not NULL and is null-terminated.
extern char *car_to_string(car c);
#endif
```

These functions create, access, and manipulate car structs. The function `make_car` creates a car struct and initializes its fields with the given arguments. The function `read_car` takes as arguments 3 prompt strings; using these it prompts (on stdout) for the car's make, model, and year, and after each prompt reads in the corresponding data (from stdin) and stores it in the corresponding field of a newly-allocated `car_t` struct; however, if the year read is 1899 or less, then it prints a message (on stdout) of the form "`You entered NNNN, please enter a year after 1899`", where `NNNN` is the year the user entered, and then reprompts with `year_prompt` and reads the year again, until the year read is at least 1900. The function `car_make` returns the make of its argument. The function `car_model` returns the model of its argument. The function `car_year` returns the year of its argument. Tests for this problem appear below.

```
// $Id: test_car.c,v 1.1 2017/04/25 18:46:37 leavens Exp $
#include <string.h>
#include "tap.h"
#include "testing_io.h"
#include "car.h"
static int test_read_not_for_you_to_implement() { // FOR TESTING ONLY
    for (int i = 0; i < 4; i++) {                 // NOT FOR YOU TO IMPLEMENT!
        car c = read_car("Car make (no spaces): ", "Car model (no spaces): ",
                         "Car year (4 digits, after 1899): ");
        printf("\nread car %d is: \"%s\"\n", i, car_to_string(c));
    }
    return EXIT_SUCCESS;
}

int main() {
    car camry = make_car("Toyota", "Camry", 2014);
    ok(camry != NULL);
    ok(strcmp(car_make(camry), "Toyota") == 0);
    ok(strcmp(car_model(camry), "Camry") == 0);
    ok(car_year(camry) == 2014);
    ok(strcmp(car_to_string(camry), "2014 Toyota Camry") == 0);
    car civic = make_car("Honda", "Civic", 2016);
    ok(civic != NULL);
    ok(strcmp(car_make(civic), "Honda") == 0);
    ok(strcmp(car_model(civic), "Civic") == 0);
    ok(car_year(civic) == 2016);
    ok(strcmp(car_to_string(civic), "2016 Honda Civic") == 0);
    car modelt = make_car("Ford", "Model T convertible", 1915);
    ok(strcmp(car_make(modelt), "Ford") == 0);
    ok(strcmp(car_model(modelt), "Model T convertible") == 0);
    ok(car_year(modelt) == 1915);
    ok(strcmp(car_to_string(modelt), "1915 Ford Model T convertible") == 0);
    testproc(test_read_not_for_you_to_implement, "test_car1");
    return exit_status();
}
```

In these tests the input (for testing `read_car`) is the following (in the file `test_car1.in`)

```
Chevrolet
Chevette
1987
DeLorean
DMC-12
1985
DeDion
La-Marquise
1884
1884
1904
Ferrari
250-GTO
1962
```

which produces the following output (on stdout).

```
Car make (no spaces): Car model (no spaces): Car year (4 digits, after 1899):
read car 0 is: "1987 Chevrolet Chevette"
Car make (no spaces): Car model (no spaces): Car year (4 digits, after 1899):
read car 1 is: "1985 DeLorean DMC-12"
Car make (no spaces): Car model (no spaces): Car year (4 digits, after 1899): You entered 1884, please enter a year
Car year (4 digits, after 1899): You entered 1884, please enter a year after 1899
Car year (4 digits, after 1899):
read car 2 is: "1904 DeDion La-Marquise"
Car make (no spaces): Car model (no spaces): Car year (4 digits, after 1899):
read car 3 is: "1962 Ferrari 250-GTO"
```

There is space for your answer below and on the next page.