Homework 7: Structs (and Pointers) in C

See Webcourses and the syllabus for due dates.

General Directions

This homework should be done individually. Note the grading policy on cooperation carefully if you choose to work together (which we don't recommend).
In order to practice for exams, we suggest that you first write out your solution to each problem on paper, and then check it typing that into the computer.
You should take steps to make your code clear, including using symbolic names for important constants and using helping functions (or helping procedures) to avoid duplicate code.
It is a good idea to test your code yourself first, before running our tests, as it is easier to debug when you run the code yourself. To do your own testing for problems that ask for a function (instead of a whole program), you will need to write a main function to do your own testing. (Note that our tests are also in a main function.)
Our tests are provided in `hw7-tests.zip`, which you can download from the homeworks directory. This zip file contains several C files with names of the form `test_f.c`, where `f` is the name of a file you should be writing (such as `us_phone`), and also some other files, including `testing_io.h`, `testing_io.c`, `tap.h`, and `tap.c`. Continuing to use `f` to stand for the name of the file you should be writing, place the code for `f` in a file named `f.c`, with the functions as named in the header file `f.h`. These conventions will make it possible to test using our testing framework.

Testing on Eustis

To test on eustis.eecs.ucf.edu, send all the files to a directory you own on eustis.eecs.ucf.edu, and compile using the command given in a comment at the top of our testing file (e.g., at the top of `test_f.c` for a file named `f`), then run the executable that results (which will be named `test_f`) by executing a command such as:

```
./test_f >test_f.txt
```

which will put the output of our testing into the file `test_f.txt`, which you can copy back to your system to view and print. You can then upload that file and your source file `f.c` to webcourses.

Testing on Your Own Machine using Code::Blocks

You can also test on your own machine using Code::Blocks, which is probably more convenient.

Project Setup

To use our tests for a file `f` (where `f` should be replaced in all cases below with the name of the function being asked for in the problem you are working on, such as `us_phone`), with Code::Blocks, create a new "Console application" project (using the File menu, select the "New" item, and then select "Project ..."). Make the project a C project (not C++!). We suggest giving the project the title "`test_f`", and putting it in a directory named, say, `cop3223h/test_f`.

Putting Our Tests into Your Project

You should unzip the contents of our hw7-tests.zip file into a directory you own, say `cop3223h/hw7testing`. You will need to add the project tests for `f` (i.e., the file `test_f.c`) and the files `testing_io.h`, `testing_io.c`, `tap.c` and `tap.h`, as well as any other files that the problem directs (such as `test_data_ivec.c` and `test_data_ivec.h` or `test_data_string_set.c` and `test_data_string_set.h`).

We recommend that you do this by copying these files into the project's directory (i.e., into the directory `cop3223h/test_f`). (You can do this using the File Explorer (on Windows) or the Finder (on a Mac), or by using the command line. Once this is done, use the "Project" menu in Code::Blocks and select the item "Add files...", then follow the dialog to add each of the files `test_f.c`, `testing_io.h`, `testing_io.c`, `tap.c`, and `tap.h` to the project. After this is done remove the dummy file `main.c` in the project, which can be done by using the "Project" menu, selecting the item "Remove files...", and then selecting the file `main.c`.

Writing Your Code

Then you will need to write the file `f.c` in the project. To do this, use the "File" menu in Code::Blocks, select the "New" item, and then from the submenu select the item "File..." Create the new file in the same project directory, `cop3223h/test_f`, with the file name `f.c`. Apparently creating the file in the project directory is not enough to have Code::Blocks understand that the file is part of the project. So you also need to use the "Project" menu and the "Add file..." item to make sure that the new file is included in the project. You can then use Code::Blocks to write the code for function `f` in the file `f.c`.

Running Our Tests

To run our tests, you can then build and run the project. If you encounter errors where Code::Blocks (or the system loader) says it cannot find a file, use the "Project" menu and the "Add file..." item to make sure that all files are included in the project.

Capturing Test Output Assuming you have everything built, you can run the tests from Code::Blocks directly, then copy and past the test output into a `.txt` file (using an editor).
However, a more automated way to capture the test output is to use the command line.
On Windows, run the `cmd.exe` program (which you can start by typing "cmd" into Cortana and selecting `cmd.exe`) and change to the directory (by a command like `cd cop3223h/test_f/`). If you made a "Debug" version of the testing program under Code::Blocks, which is the default for Code::Blocks, then change to the `bin/Debug` directory (with a command like `cd bin/Debug`). On Windows, then you would execute the following from the command line prompt

```
test_f.exe >test_f.txt
```

which will put the testing output into the file `test_f.txt`.
On a Mac, use the Terminal program instead of `cmd.exe` in the directions above. After changing to the right directory, execute the following from the terminal prompt:

```
./test_f >test_f.txt
```

Handing in Files

You can then upload the output `test_f.txt` file and your source file `f.c` to webcourses.

What to turn in

For problems that ask you to write a file of C functions, upload your code as an ASCII file with suffix `.c`, and also upload the output of running our tests (as an ASCII file with suffix `.txt`).

Problems

The problems in this homework are geared towards building a contacts application, which is specified in problem 4.

1. (50 points) [Programming] In this problem you will implement functions to deal with phone numbers (for phones in the USA). Your task is to implement the functions specified in Figure 1 on the following page and Figure 2 on page 5.

   There are tests in `test_us_phone.c` (which is provided in the `hw7-tests.zip` file), see Figure 3 on page 6 and Figure 4 on page 7.

   Hints: To implement `make_us_phone`, you will need to use `malloc`. You can write a statement such as

   ```
   us_phone p = (us_phone)malloc(sizeof(us_phone_t));
   ```

   to create a `us_phone_t` object on the heap and put a pointer to it in `p`. When this statement executes successfully, it produces a state like that pictured in Figure 5 on page 8. You will also need to use `malloc` to implement the functions `area_code`, `digits`, and `us_phone_to_string`, as they each return strings; don't create dangling pointers! You may also find `sprintf` (as specified in `string.h`) useful.

   When you read in strings from stdin, you can use `scanf` (declared in `stdio.h`), but be sure that the buffer you allocate for putting strings into has enough room to hold the null character at the end. That is, declare or allocate arrays that have one more char element than the number of chars you are expecting.

   You may want to use the `isdigit()` macro from the built-in `ctype.h` to test whether a char is a digit.

   You can convert a char `c` that represents a digit to its int equivalent by subtracting the char for `0` from it, as in `c - '0'`. Conversely, to convert an int `i` that is between 0 and 9 to the corresponding char code, use `'0'+ i`. It would be sensible to define little functions that do these tasks. (In general, don't hesitate to define helping functions.)

   To run our tests, copy the files `testing_io.h`, `testing_io.c`, `tap.c`, `tap.h`, `us_phone.h`, `test_us_phone.c`, `test_us_phone1.in`, and `test_us_phone1.expected` into your project/directory (and add them to your project if you are using Code::Blocks). Then write your code for the `us_phone` module in a file `us_phone.c`. Run the tests in `test_us_phone.c`.

   Remember to turn in your C source code file `us_phone.c` and the output of running our tests. Your code and also the output of running our tests should be submitted to webcourses as ASCII text files that you upload.

```c
// $Id: us_phone.h,v 1.2 2017/04/09 20:56:40 leavens Exp leavens $
#ifndef US_PHONE_H
#define US_PHONE_H 1
#include <stdlib.h>
#include <stdbool.h>

#define AREA_LEN 3
#define EXCHANGE_LEN 3
#define SUB_LEN 4

typedef struct {
    int area[AREA_LEN];
    int exchange[EXCHANGE_LEN];
    int subscriber[SUB_LEN];
} us_phone_t;

// Make "us_phone" a synonym for a us_phone_t pointer type.
typedef us_phone_t *us_phone;

// requires: area, exchange, and sub are all null-terminated
//           strings that are allocated, and the length of area is AREA_LEN
//           and the length of exchange is EXCHANGE_LEN, and the length
//           of sub is SUB_LEN.
// modifies: stdout
// effect: creates a fresh us_phone_t object allocated on the heap,
//         initializes its fields from the corresponding arguments,
//         and returns a pointer to the new object
//         (or NULL if the new object cannot be allocated or
//         if some of the characters in area, exchange, or sub are not digits)
//         When malloc fails the message "malloc failed!" is printed on stdout.
//         When some charactr is not a digit, a message of the form
//         "'C' is not a decimal digit" is printed on stdout, where C is the
//         non-digit character.
extern us_phone make_us_phone(const char *area, const char *exchange,
                              const char *sub);

// requires: stdin and stdout are open; area_prompt and digits_prompt are
//           both allocated, null-terminated strings.
// modifies: stdin, stdout
// effect: prompts with area_prompt on stdout, then reads the area code.
//         Then prompts with digits_prompt on stdout and reads the phone number.
//         The phone number is expected to be in the format NNN-NNNN,
//         with a hyphen between the exchange (NNN) and the subscriber number
//         (NNNN), where each N is a decimal digit.
//         Then allocates a us_phone_t struct on the heap and initializes it
//         with the data read from stdin, and returns a pointer to that
//         newly allocated struct (or NULL if there was an error or
//         if some of the characters that should be digits are not).
//         When malloc fails the message "malloc failed!" is printed on stdout.
//         When some charactr is not a digit, a message of the form
//         "'C' is not a decimal digit" is printed on stdout, where C is the
//         non-digit character.
extern us_phone read_us_phone(const char *area_prompt,
                              const char *digits_prompt);
```

Figure 1: Part 1 (of 2) of the header file `us_phone.h` for phone numbers, defining the struct type `us_phone_t` and the pointer type `us_phone`, and specifying operations you are to implement.

```
// requires: p is allocated (and not NULL)
// ensures: result is a null-terminated string representing the area code of p
extern char *area_code(us_phone p);

// requires: p is allocated (and not NULL)
// ensures: result is a string of the form "Exchange-Subscriber" where
//          Exchange is a string representing p's exchange and Subscriber
//          is a string representing p's subscriber number.
extern char *digits(us_phone p);

// requires: p is allocated (and not NULL)
// ensures: result is a string of the form:
//              "(Areacode)Exchange-Subscriber"
//          where Areacod is a string representing the area code of p,
//          Exchange is a string representing p's exchange and Subscriber
//          is a string representing p's subscriber number.
extern char *us_phone_to_string(us_phone p);

// requires: p1 and p2 are both allocated (and not NULL)
// ensures: result is true just when each field of p1 is equal to
//          the corresponding field of p2.
extern bool equal_us_phone(us_phone p1, us_phone p2);
#endif
```

Figure 2: Part 2 (of 2) of the header file `us_phone.h` for phone numbers, specifying operations you are to implement.

```
// $Id: test_us_phone.c,v 1.3 2017/04/10 00:29:35 leavens Exp leavens $
// Compile with:
// gcc tap.c testing_io.c us_phone.c test_us_phone.c -o test_us_phone

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tap.h"
#include "testing_io.h"
#include "us_phone.h"

static int test_read_us_phone() {
    for (int i = 0; i < 7; i++) {
        us_phone p = read_us_phone("area code (digits only): ",
                                   "number (in the form 555-1212): ");
        if (p == NULL) {
            continue;
        }
        printf("read phone %d is: \"%s\"\n", i, us_phone_to_string(p));
    }
    return EXIT_SUCCESS;
}

int main() {
    plan(19);
    us_phone pTV = make_us_phone("913", "555", "1212");
    ok(pTV != NULL);
    if (pTV != NULL) {
        ok(pTV->area[0] == 9 && pTV->area[1] == 1 && pTV->area[2] == 3,
           "pTV->area[0] == 9 && pTV->area[1] == 1 && pTV->area[2] == 3");
        ok(pTV->exchange[0] == 5 && pTV->exchange[1] == 5 && pTV->exchange[2] == 5,
           "pTV->exchange[0] == 5 && pTV->exchange[1] == 5 && pTV->exchange[2] == 5");
        ok(pTV->subscriber[0] == 1 && pTV->subscriber[1] == 2
           && pTV->subscriber[2] == 1 && pTV->subscriber[3] == 2,
           "pTV->subscriber[0] == 4 && pTV->subscriber[1] == 7\n && pTV->subscriber[2] == 5 && pTV->subscriber[3] == 2")
    }

    us_phone pGL = make_us_phone("407", "823", "4758");
    ok(pGL != NULL);
    if (pGL != NULL) {
        ok(pGL->area[0] == 4 && pGL->area[1] == 0 && pGL->area[2] == 7,
           "pGL->area[0] == 4 && pGL->area[1] == 0 && pGL->area[2] == 7");
        ok(pGL->exchange[0] == 8 && pGL->exchange[1] == 2 && pGL->exchange[2] == 3,
           "pGL->exchange[0] == 8 && pGL->exchange[1] == 2 && pGL->exchange[2] == 3");
        ok(pGL->subscriber[0] == 4 && pGL->subscriber[1] == 7
           && pGL->subscriber[2] == 5 && pGL->subscriber[3] == 8,
           "pGL->subscriber[0] == 4 && pGL->subscriber[1] == 7\n && pGL->subscriber[2] == 5 && pGL->subscriber[3] == 8")
    }
```

Figure 3: Part 1 (of 2) of the tests for the module us_phone.

```
    if (pTV != NULL && pGL != NULL) {
        ok(strcmp(area_code(pTV), "913") == 0,
           "strcmp(area_code(pTV), \"913\") == 0, area_code(pTV) == \"%s\"",
           area_code(pTV));
        ok(strcmp(area_code(pGL), "407") == 0,
           "strcmp(area_code(pGL), \"407\") == 0, area_code(pGL) == \"%s\"",
           area_code(pGL));

        ok(strcmp(digits(pTV), "555-1212") == 0,
           "strcmp(digits(pTV), \"555-1212\") == 0, digits(pTV) == \"%s\"",
           digits(pTV));
        ok(strcmp(digits(pGL), "823-4758") == 0,
           "strcmp(digits(pGL), \"823-4758\") == 0, digits(pGL) == \"%s\"",
           digits(pGL));

        ok(strcmp(us_phone_to_string(pTV), "(913)555-1212") == 0,
           "us_phone_to_string(pTV) == \"%s\"", us_phone_to_string(pTV));
        ok(strcmp(us_phone_to_string(pGL), "(407)823-4758") == 0,
           "us_phone_to_string(pGL) == \"%s\"", us_phone_to_string(pGL));

        ok(equal_us_phone(pTV, pTV), "equal_us_phone(pTV, pTV)");
        ok(equal_us_phone(pGL, pGL), "equal_us_phone(pGL, pGL)");
        ok(!equal_us_phone(pTV, pGL), "!equal_us_phone(pTV, pGL)");
        ok(!equal_us_phone(pGL, pTV), "!equal_us_phone(pGL, pTV)");
    }

    testproc(test_read_us_phone, "test_us_phone1");


    return exit_status();
}
```

Figure 4: Part 2 (of 2) of the tests for the module us_phone.

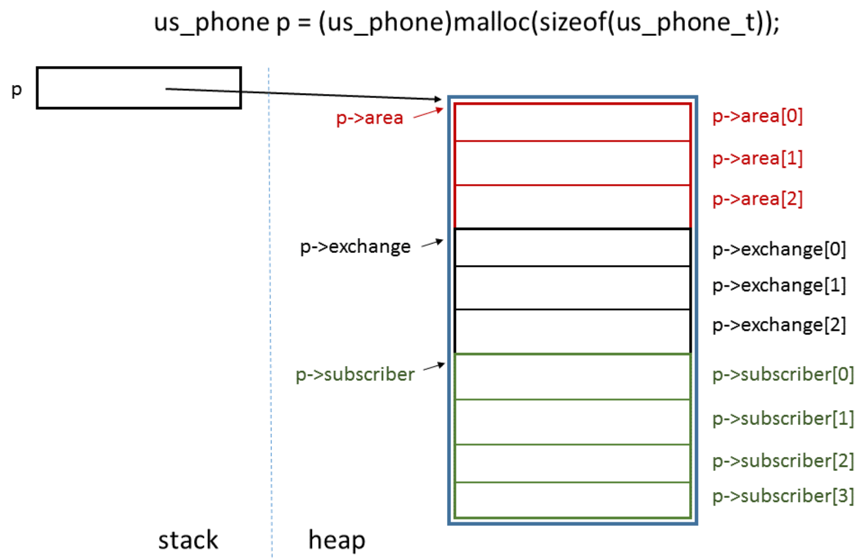

Figure 5: Picture of the stack (on left) and heap (on right) after executing the code shown at the top.

2. (50 points) [Programming] In this problem you will implement functions to deal with people's names. Your task is to implement the functions specified in Figure 6 on the next page.

   There are tests in `test_person.c` (which is provided in the `hw7-tests.zip` file), see Figure 7 on page 11.

   Hints: In this problem the struct type `person_t` contains pointers to two strings; the arrays holding the chars in those strings must be allocated using `malloc` so that the code does not create dangling pointers. Make sure that when you allocate arrays for strings that you leave enough room for the null character at the end of the string.

   When reading names, use the macro `NAME_READING_FORMAT`, which is defined in `person.h`, as the format for `scanf`.

   To run our tests, copy the files `testing_io.h`, `testing_io.c`, `tap.c`, `tap.h`, `person.h`, `test_person.c`, `test_person1.in`, and `test_person1.expected` into your directory (and add them to your project if you are using Code::Blocks). Then write your code for the `person` module in a file `person.c`. Run the tests in `test_person.c`.

   Remember to turn in your C source code file `person.c` and the output of running our tests. Your code and also the output of running our tests should be submitted to webcourses as ASCII text files that you upload.

```c
// $Id: person.h,v 1.2 2017/04/10 00:29:35 leavens Exp leavens $
#ifndef PERSON_H
#define PERSON_H 1
#include <stdlib.h>
#include <stdbool.h>
#define MAX_NAME_LENGTH 50
// The following is for use in scanf calls
#define NAME_READING_FORMAT "%50s"

typedef struct person_s {
    const char *first_name;
    const char *last_name;
} person_t;

// Make "person" a synonym for a person_t pointer type.
typedef person_t *person;

// requires: first_name and last_name are both null-terminated
//           strings that are allocated on the heap.
// effect: creates a fresh person_t object allocated on the heap,
//         initializes its fields with the corresponding arguments,
//         and returns a pointer to the new object
//         (or NULL if the new object cannot be allocated)
//         When malloc fails the message "malloc failed!" is printed on stdout.
extern person make_person(const char *first_name, const char *last_name);

// requires: stdin and stdout are open; fn_prompt, ln_prompt are
//           all allocated, null-terminated strings.
// modifies: stdin, stdout
// effect: prompts with fn_prompt on stdout, then reads the first name.
//         Then prompts with ln_prompt on stdout and reads the last name.
//         Names are assumed to be no more than MAX_NAME_LENGTH chars long.
//         Then allocates a person_t struct on the heap and initializes it
//         with the data read from stdin, and returns a pointer to that
//         newly allocated struct (or NULL if there was an error).
//         When malloc fails the message "malloc failed!" is printed on stdout.
extern person read_person(const char *fn_prompt, const char* ln_prompt);

// requires: p is allocated (and not NULL)
// ensures: result is the first name of p
extern const char *first_name(person p);

// requires: p is allocated (and not NULL)
// ensures: result is the last name of p
extern const char *last_name(person p);

// requires: p is allocated (and not NULL)
// ensures: result is a string of the form:
//   "Firstname Lastname"
// where Firstname is the result of calling first_name() on p,
// and Lastname is the result of calling last_name() on p.
extern char *person_to_string(person p);

// requires: p1 and p2 are both allocated (and not NULL)
// ensures: result is true just when each field of p1 is equal to
// the corresponding field of p2 (i.e., when they have the same
// first names and last names).
extern bool equal_person(person p1, person p2);
#endif
```

Figure 6: Header file `person.h`, defining the struct type `person_t` and the pointer type `person`, and specifying operations you are to implement.

```
// $Id: test_person.c,v 1.2 2017/04/10 00:29:35 leavens Exp leavens $
// Compile with:
// gcc tap.c testing_io.c person.c test_person.c -o test_person

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tap.h"
#include "testing_io.h"
#include "person.h"

static int test_read_person() {
    for (int i = 0; i < 7; i++) {
        person p = read_person("first name? ",
                               "last name? ");
        if (p == NULL) {
            continue;
        }
        printf("read person %d is: \"%s\"\n", i, person_to_string(p));
    }
    return EXIT_SUCCESS;
}

int main() {
    plan(13);
    person ada = make_person("Ada", "Lovelace");
    ok(ada != NULL, "made ada successfully");
    if (ada != NULL) {
        ok(strcmp(ada->first_name, "Ada") == 0,
           "strcmp(ada->first_name, \"Ada\") == 0");
        ok(strcmp(ada->last_name, "Lovelace") == 0,
           "strcmp(ada->last_name, \"Lovelace\") == 0");
        ok(strcmp(person_to_string(ada), "Ada Lovelace") == 0,
           "strcmp(person_to_string(ada), \"Ada Lovelace\") == 0");
    }

    person alan = make_person("Alan", "Turing");
    ok(alan != NULL, "made alan successfullly");
    if (alan != NULL) {
        ok(strcmp(alan->first_name, "Alan") == 0,
           "strcmp(alan->first_name, \"Alan\") == 0");
        ok(strcmp(alan->last_name, "Turing") == 0,
           "strcmp(alan->last_name, \"Turing\") == 0");
        ok(strcmp(person_to_string(alan), "Alan Turing") == 0,
           "strcmp(person_to_string(alan), \"Alan Turing\") == 0");
    }

    if (ada != NULL && alan != NULL) {
        ok(equal_person(ada, ada), "equal_us_phone(ada, ada)");
        ok(equal_person(alan, alan), "equal_us_phone(alan, alan)");
        ok(!equal_person(ada, alan), "!equal_us_phone(ada, alan)");
        ok(!equal_person(alan, ada), "!equal_us_phone(alan, ada)");
    }

    testproc(test_read_person, "test_person1");

    return exit_status();
}
```

Figure 7: Tests for the module person.

3. (25 points) [Programming] In this problem you will implement functions to deal with contact structs that combine a person's name and a phone number. Your task is to implement the functions specified in Figure 8.

```
// $Id: contact.h,v 1.3 2017/04/10 04:37:39 leavens Exp leavens $
#ifndef CONTACT_H
#define CONTACT_H 1
#include <stdlib.h>
#include "person.h"
#include "us_phone.h"

typedef struct contact_pair {
    person who;
    us_phone num;
} contact_t;

typedef contact_t * contact;

// requires: num_contacts < MAX_CONTACT_SIZE;
// modifies: stdout, stdin
// effect: From stdin, read a person (with prompts "first name? " and
//         "last name? ")  and a US phone number (with prompts "area code? ",
//         and "number (in form 555-1212)? "), the prompts all going to stdout.
//         Then create a new contact with that person and phone number,
//         and return it. Errors cause a message to be printed on stdout
//         (as specified in the modules person and us_phone)
//         and NULL to be returned.
extern contact read_contact();

// requires: contact is allocated (and not NULL)
// ensures: result is a string of the form:
//            "name: Firstname Lastname phone: (AAA)EEE-SSSS"
//          where Firstname is the first_name of c->who,
//                Lastname is the last_name of c->who,
//                and AAA is the area_code of c->num
//                and EEE-SSSS is the digits of c->num.
extern char *contact_to_string(contact c);

#endif
```

Figure 8: Header file `contact.h`, defining the struct type `contact_t` and the pointer type `contact`, and specifying operations you are to implement.

There are tests in `test_contact.c` (which is provided in the `hw7-tests.zip` file), see Figure 9 on the following page.

The input file is shown in Figure 10 on page 14.

The expected output is shown in Figure 11 on page 15.

Hints: Be sure to use `malloc` to avoid creating dangling pointers in both functions.

Use the exact prompt strings specified in Figure 8 when reading a contact.

To run our tests, copy the files `testing_io.h`, `testing_io.c`, `tap.c`, `tap.h`, `contact.h`, `test_contact.c`, `test_contact1.in`, and `test_contact1.expected` into your directory (and add them to your project if you are using Code::Blocks). You should also copy (and add) the code for the

```
// $Id: test_contact.c,v 1.1 2017/04/10 00:29:35 leavens Exp $
// Compile with:
// gcc tap.c testing_io.c person.c us_phone.c contact.c test_contact.c -o test_contact
#include <stdlib.h>
#include "contact.h"
#include "tap.h"
#include "testing_io.h"

#define NUM_TO_READ 10

int test_read_contact() {
    contact test_contacts[NUM_TO_READ];
    int next = 0;
    for (int i = 0; i < 10; i++) {
        test_contacts[next] = read_contact();
        if (test_contacts[next] != NULL) {
            printf("input %d is %s\n", i, contact_to_string(test_contacts[i]));
            next++;
        }
    }
    return EXIT_SUCCESS;
}

int main() {
    plan(1);
    testproc(test_read_contact, "test_contact1");
    return exit_status();
}
```

Figure 9: Tests for the module `contact`, which use the input file `test_contact1.in` and have expected output in `test_contact1.expected`.

us_phone and `person` modules to this directory (or Code::Blocks project). Then write your code for the `contact` module in a file `contact.c`. Run the tests in `test_contact.c`.

Remember to turn in your C source code file `contact.c` and the output of running our tests. Your code and also the output of running our tests should be submitted to webcourses as ASCII text files that you upload.

```
Red
Cross
800
733-2767
American
HeartAssociation
800
242-8721
American
LungAssociation
800
586-4872
American
CancerSociety
800
227-2345
Oxfam
America
800
776-9326
Ada
Lovelace
101
555-1212
Zaphod
Beeblebrox
321
555-1234
Donald
Trump
202
456-1111
Bill
Nelson
202
224-5274
Marco
Rubio
202
224-3041
```

Figure 10: Input file for `test_contact.c`.

```
first name? last name? area code? number (in form 555-1212)? input 0 is name: Red Cross phone: (800)733-2767
first name? last name? area code? number (in form 555-1212)? input 1 is name: American HeartAssociation phone: (800)242-
first name? last name? area code? number (in form 555-1212)? input 2 is name: American LungAssociation phone: (800)586-4
first name? last name? area code? number (in form 555-1212)? input 3 is name: American CancerSociety phone: (800)227-234
first name? last name? area code? number (in form 555-1212)? input 4 is name: Oxfam America phone: (800)776-9326
first name? last name? area code? number (in form 555-1212)? input 5 is name: Ada Lovelace phone: (101)555-1212
first name? last name? area code? number (in form 555-1212)? input 6 is name: Zaphod Beeblebrox phone: (321)555-1234
first name? last name? area code? number (in form 555-1212)? input 7 is name: Donald Trump phone: (202)456-1111
first name? last name? area code? number (in form 555-1212)? input 8 is name: Bill Nelson phone: (202)224-5274
first name? last name? area code? number (in form 555-1212)? input 9 is name: Marco Rubio phone: (202)224-3041
```

Figure 11: Expected output file for `test_contact.c`.

4. (50 points) [Programming] In this problem you will build on the earlier problems to implement an application that manages a database of contacts. The database itself can be implemented as an array of contact structs. Your task is to implement the functions specified in Figure 12 on the following page.

   There are tests in `test_contacts_app.c` (which is provided in the `hw7-tests.zip` file), see Figure 13 on page 18.

   Hints: You should use lots of helping functions. Think abstractly about what has to be done first, before writing the code.

   You may find it useful to use a **`switch`** statement or two.

   If you use **`static`** variables to communicate between parts of your code, be sure to reinitialize them when `contacts_app()` is called.

   It may be helpful to use a main function that simply calls `contacts_app()` so that you can test the code interactively before running our tests.

   To run our tests, copy the files `testing_io.h`, `testing_io.c`, `tap.c`, `tap.h`, `contacts_app.h`, `test_contacts_app.c`, `test_contacts_app1.in`, `test_contacts_app2.in`, `test_contacts_app3.in`, and `test_contacts_app1.expected`, `test_contacts_app2.expected`, `test_contacts_app3.expected` into your directory (and add them to your project if you are using Code::Blocks). You should also copy (and add) the code for the `us_phone`, `person`, and `contact` modules to this directory (or Code::Blocks project). Then write your code for the `contacts_app` module in a file `contacts_app.c`. Run the tests in `test_contacts_app.c`.

   Remember to turn in your C source code file `contacts_app.c` and the output of running our tests. Your code and also the output of running our tests should be submitted to webcourses as ASCII text files that you upload.

```c
// $Id: contacts_app.h,v 1.2 2017/04/10 04:37:39 leavens Exp leavens $
#ifndef CONTACTS_APP_H
#define CONTACTS_APP_H 1
#include <stdlib.h>
#include "contact.h"

#define MAX_CONTACT_SIZE 100

// requires: stdin and stdout are open
// modifies: stdin and stdout
// effect: The program has 2 phases after intialization. First it
//         reads contact information from the user, then it answers
//         queries.
//         The reading phase starts by outputting
//           "To start, you can input up to 100 contacts"
//         on stdout. Then the user is asked "Input a contact? [Y/n] ";
//         when the user responds with "n", "N", "No", or "no",
//         the reading phase ends. When the user responds with "Y", "y",
//         "yes", or "Yes", then a contact is read from stdin
//         (with prompts on stdout as specified in contacts.h)
//         and the reading phase continues. The reading phase can also end
//         if 100 contacts are entered.
//         The query phase starts by outputting
//           "Now you can query the database of contacts"
//         on stdout. The user is prompted to enter a command number using
//         the following prompt on stdout:
//           "Choose a command by typing in its number (and a return)"
//           "0 = quit this program,  1 = find contact info. by first name"
//           "2 = find contact info. by last name,  3 = print all contacts"
//           "Command number? "
//         If the user enters 0, then the query phase and the program finishes.
//         If the user enters 1, then the user is prompted for the first name
//         (with "First name to search for? ", on stdout),
//         and the name is read from stdin;
//         then all contacts in the database with that first name are printed
//         (using the format in contact_to_string) to stdout,
//         and the query process repeats.
//         If the user enters 2, then the user is prompted for the last name
//         (with "Last name to search for? ", on stdout),
//         and the name is read from stdin;
//         then all contacts in the database with that last name are printed
//         (using the format in contact_to_string) to stdout,
//         and the query process repeats.
//         If the user enters 3, then each contact in the database is printed,
//         (using the format in contact_to_string) to stdout. The order of
//         printing is the order in which the contacts were entered.
//         Then the query process repeats.
extern int contacts_app();

#endif
```

Figure 12: Header file `contacts_app.h`, specifying the functions you are to implement.

```c
// $Id: test_contacts_app.c,v 1.1 2017/04/10 00:29:35 leavens Exp $
// Compile with:
// gcc tap.c testing_io.c person.c us_phone.c contact.c contacts_app.c test_contacts_app.c -o test_contacts_app
#include "contacts_app.h"
#include "tap.h"
#include "testing_io.h"

int main() {
    plan(3);
    testproc(contacts_app, "test_contacts_app1");
    testproc(contacts_app, "test_contacts_app2");
    testproc(contacts_app, "test_contacts_app3");
    return exit_status();
}
```

Figure 13: Tests for the module `contacts_app`. These tests use the inputs given in `test_contacts_app1.in`, `test_contacts_app2.in`, and `test_contacts_app3.in`, as well as the expected outputs in `test_contacts_app1.expected`, `test_contacts_app2.expected`, and `test_contacts_app3.expected`. These files are found in `hw7-tests.zip`.

```
To start, you can input up to 100 contacts
Input a contact? [Y/n] y
first name? Ada
last name? Lovelace
area code? 321
number (in form 555-1212)? 123-4567
Input a contact? [Y/n] y
first name? Zaphod
last name? Beeblebrox
area code? 407
number (in form 555-1212)? 995-0001
Input a contact? [Y/n] y
first name? Ada
last name? Augusta
area code? 995
number (in form 555-1212)? 000-0000
Input a contact? [Y/n] n
Now you can query the database of contacts
Choose a command by typing in its number (and a return)
0 = quit this program,  1 = find contact info. by first name
2 = find contact info. by last name,  3 = print all contacts
Command number? 3
name: Ada Lovelace phone: (321)123-4567
name: Zaphod Beeblebrox phone: (407)995-0001
name: Ada Augusta phone: (995)000-0000
Choose a command by typing in its number (and a return)
0 = quit this program,  1 = find contact info. by first name
2 = find contact info. by last name,  3 = print all contacts
Command number? 1
First name to search for? Ada
name: Ada Lovelace phone: (321)123-4567
name: Ada Augusta phone: (995)000-0000
Choose a command by typing in its number (and a return)
0 = quit this program,  1 = find contact info. by first name
2 = find contact info. by last name,  3 = print all contacts
Command number? 2
Last name to search for? Lovelace
name: Ada Lovelace phone: (321)123-4567
Choose a command by typing in its number (and a return)
0 = quit this program,  1 = find contact info. by first name
2 = find contact info. by last name,  3 = print all contacts
Command number? 0
```

Figure 14: Sample dialog with `contacts_app`. Many prompts run from the beginning of a line to the space after the question mark. The names and numbers are user inputs.

5. (30 points; extra credit) [Design] Design and test an additional query for the contacts app, that allows the user to query for full names, instead of just a first or last name. Your submission will be judged on the following basis:

   - How sensible the design is (10 points).
   - The code, in particular how correct and clear it is (10 points)
   - The tests, how thorough they are (10 points). Be sure to cover some corner cases.

6. (50 points; extra credit) [Design] Instead of separating the construction of the database and the querying into 2 phases of the contacts app, redesign it so that:

   - contacts can be added at any time,
   - contacts can be read from a file, and
   - contacts can be saved to a file.

   Your submission will be judged on the following basis:

   - How sensible the design and specifications are (10 points).
   - The code, in particular how correct and clear it is (10 points)
   - The tests, how thorough they are (10 points). Be sure to cover some corner cases.

Points

This homework's total points: 175. Total extra credit points: 80.