Homework 6: Loops 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 `hw6-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 the function you should be writing (such as `ivec_len`), and also some other files, including `tap.h` and `tap.c`. Continuing to use `f` to stand for the name of the function you should be writing, place the code for `f` in a file named `f.c`, with the function itself named `f` (actually, it should be named with whatever `f` stands for). 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 function 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 function `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 `ivec_len`), 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 hw6-tests.zip file into a directory you own, say `cop3223h/hw6testing`. You will need to add the project tests for `f` (i.e., the file `test_f.c`) and the files `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`, `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 C function, 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

Integer Vector (ivec) Problems

Vectors are useful in various kinds of mathematics, the physical sciences, and in computer graphics. In this section you will implement several operations on ivecs, our name for int vectors. For purposes of this

problem an ivec is an array of **int** values, which contains a zero (0); the zero is used to mark the end of the section of the array used for the vector. Everything in the array after the zero (i.e., at higher indexes) is not considered part of the vector.
In this problem you will implement a few operations on vectors. Note that some of these manipulate the vector in place by assigning to the elements of the array in which it lives (before the element containing zero).
The tests for these ivec problems share some data, which is found in `test_data_ivec.c` (with header file `test_data_ivec.h`), as shown in Figure 1 and Figure 2 on the next page. These files are provided in `hw6-tests.zip`. For each ivec problem you will need to copy both `test_data_ivec.c` and `test_data_ivec.h` to your project/directory.

```
// $Id: test_data_ivec.c,v 1.5 2017/03/20 13:15:58 leavens Exp $
#include <stdio.h>
#include "test_data_ivec.h"
// data for testing ivecs
int v0[1];    /* {0} */
int v1[2];    /* {1, 0} */
int v3223[5]; /* {3, 2, 2, 3, 0} */
int v3123[5]; /* {3, 1, 2, 3, 0} */
int v2123[5]; /* {2, 1, 2, 3, 0} */
int v3323[5]; /* {3, 3, 2, 3, 0} */
int v3224[5]; /* {3, 2, 2, 4, 0} */
int v3221[5]; /* {3, 2, 2, 1, 0} */
int v3223ext[9]; /* {3, 2, 2, 3, 0, 5, 7, 0, 99} */
int vucf[9];     /* {8, 2, 3, 0, 4, 7, 5, 8, 0} */
int vto100[VTO100_SIZE];  /* {1, 2, ..., 99, 100, 0} */

// assigns: all of the elements of all of the arrays above
// effect: (re)initialize the test data arrays
void init_test_data() {
    v0[0] = 0;
    v1[0] = 1; v1[1] = 0;
    v3223[0] = 3; v3223[1] = 2; v3223[2] = 2; v3223[3] = 3; v3223[4] = 0;
    v3123[0] = 3; v3123[1] = 1; v3123[2] = 2; v3123[3] = 3; v3123[4] = 0;
    v2123[0] = 2; v2123[1] = 1; v2123[2] = 2; v2123[3] = 3; v2123[4] = 0;
    v3323[0] = 3; v3323[1] = 3; v3323[2] = 2; v3323[3] = 3; v3323[4] = 0;
    v3224[0] = 3; v3224[1] = 2; v3224[2] = 2; v3224[3] = 4; v3224[4] = 0;
    v3221[0] = 3; v3221[1] = 2; v3221[2] = 2; v3221[3] = 1; v3221[4] = 0;
    v3223ext[0] = 3; v3223ext[1] = 2; v3223ext[2] = 2; v3223ext[3] = 3;
    v3223ext[4] = 0; v3223ext[5] = 5; v3223ext[6] = 7; v3223ext[7] = 0; v3223ext[8] = 99;
    vucf[0] = 8; vucf[1] = 2; vucf[2] = 3; vucf[3] = 0;
    vucf[4] = 4; vucf[5] = 7; vucf[6] = 5; vucf[7] = 8; vucf[8] = 0;
    for (int i = 0; i < VTO100_SIZE; i++) {
        vto100[i] = i+1;
    }
    vto100[VTO100_SIZE-1] = 0;
}
```

Figure 1: Part 1 (of 2) of the Test data for ivecs, in the file `test_data_ivec.c`. There is also a header file `test_data_ivec.h` (not shown) that declares these arrays.

```c
/* requires: value and expected have at least sz elements
   effect: When the result is false, prints debugging information
           about the two arrays on stdout. */
static void print_debug_info(const int value[], const int expected[], int sz) {
    for (int j = 0; j < sz; j++) {
        if (value[j] == expected[j]) {
            printf("value[%d] == %d == expected[%d]\n", j, value[j], j);
        } else {
            printf("value[%d] == %d but expected[%d] == %d\n",
                   j, value[j], j, expected[j]);
        }
    }
}

/* requires: value and expected have at least sz elements
   ensures: result is true just when the first sz elements of value
            are equal to the first sz elements of expected, i.e.,
            the result is true just when
               for all i such that 0 <= i < sz, value[i] == expected[i].
   effect: When the result is false, prints the two arrays on stdout. */
bool array_equals(const int value[], const int expected[], int sz) {
    for (int i = 0; i < sz; i++) {
        if (value[i] != expected[i]) {
            print_debug_info(value, expected, sz);
            return false;
        }
    }
    return true;
}
```

Figure 2: Part 2 (of 2) of the Test data for ivecs, in the file `test_data_ivec.c`.

1. (20 points) [Programming] In C, write a function

   ```
   extern int ivec_len(const int ivec[]);
   ```

   that takes an ivec (i.e., an array of ints that is guaranteed to contain a 0 element), and returns its length, which is the number of non-zero elements up to the first element containing 0. There are tests in `test_ivec_len.c` (which is provided in the `hw6-tests.zip` file), see Figure 3.

```
// $Id: test_ivec_len.c,v 1.5 2017/03/21 03:12:06 leavens Exp leavens $
// Compile with
// gcc tap.c ivec_len.c test_data_ivec.c test_ivec_len.c -o test_ivec_len
#include "tap.h"
#include "ivec_len.h"

// declarations of test data (see test_data_ivec.c) used below
#include "test_data_ivec.h"

// testing for ivec_len().
int main() {
    plan(11);
    int v[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    ok(ivec_len(v) == 5, "ivec_len(v) == 5");
    // using test data from ivec_test_data.c below
    init_test_data();
    ok(ivec_len(v0) == 0, "ivec_len(v0) == 0");
    ok(ivec_len(v1) == 1, "ivec_len(v1) == 1");
    ok(ivec_len(v3223) == 4, "ivec_len(v3223) == 4");
    ok(ivec_len(v3123) == 4, "ivec_len(v3123) == 4");
    ok(ivec_len(v2123) == 4, "ivec_len(v2123) == 4");
    ok(ivec_len(v3323) == 4, "ivec_len(v3123) == 4");
    ok(ivec_len(v3224) == 4, "ivec_len(v3224) == 4");
    ok(ivec_len(v3223ext) == 4, "ivec_len(v3224ext) == 4");
    ok(ivec_len(vucf) == 3, "ivec_len(vucf) == 3");
    int l100 = ivec_len(vto100);
    ok(l100 == 100, "ivec_len(vto100) == 100, result was %d", l100);
    return exit_status();
}
```

Figure 3: Tests for `ivec_len`, which use the test data in Figure 1 on page 3.

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

   Remember to turn in your C source code file `ivec_len.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.

2. (20 points) [Programming] In C, write a function

   ```
   extern void ivec_scalar_mult(int target[], const int x);
   ```

   that takes an ivec, `target` and an int `x`, and modifies `target`'s elements such that each of the elements before the first 0 in `target` becomes `x` times that element (and the elements past the first 0 are unchanged). There are tests in Figure 4 on the following page and a detailed specification in `ivec_scalar_mult.h`.

```c
// $Id: test_ivec_scalar_mult.c,v 1.6 2017/03/20 13:15:58 leavens Exp $
// Compile with:
// gcc tap.c ivec_len.c ivec_scalar_mult.c test_data_ivec.c test_ivec_scalar_mult.c -o test_ivec_scalar_mult
#include <stdbool.h>
#include "tap.h"
#include "ivec_scalar_mult.h"
// declarations of test data (see test_data_ivec.c) and array_equals used below
#include "test_data_ivec.h"

// testing for ivec_scalar_mult().
int main() {
    plan(10);
    int v[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    ivec_scalar_mult(v, 10);
    int vans[] = {90, 70, 80, 60, 90, 0, 5, 4, 0};
    ok(array_equals(v, vans, 9), "scalar_mult of v is vans");
    // using test data from ivec_test_data.c below...
    init_test_data();

    int v0ans[] = {0};
    ivec_scalar_mult(v0, 934);
    ok(array_equals(v0, v0ans, 1), "scalar mult of v0 by 934");

    int v1ans[] = {55, 0};
    ivec_scalar_mult(v1, 55);
    ok(array_equals(v1, v1ans, 2), "scalar mult of v1 by 55");

    int v3223ans[] = {6, 4, 4, 6, 0};
    ivec_scalar_mult(v3223, 2);
    ok(array_equals(v3223, v3223ans, 5), "scalar mult of v3223 by 2");

    int v3123ans[] = {9, 3, 6, 9, 0};
    ivec_scalar_mult(v3123, 3);
    ok(array_equals(v3123, v3123ans, 5), "scalar mult of v3123 by 2");

    int v2123ans[] = {200, 100, 200, 300, 0};
    ivec_scalar_mult(v2123, 100);
    ok(array_equals(v2123, v2123ans, 5), "scalar mult of v2123 by 100");

    int v3223extans[] = {150, 100, 100, 150, 0, 5, 7, 0, 99};
    ivec_scalar_mult(v3223ext, 50);
    ok(array_equals(v3223ext, v3223extans, 9), "scalar mult of v3223ext by 50");

    int v3223extans2[] = {300, 200, 200, 300, 0, 5, 7, 0, 99};
    ivec_scalar_mult(v3223ext, 2);
    ok(array_equals(v3223ext, v3223extans2, 9), "scalar mult of v3223ext by 2");

    int vucfans[] = {48, 12, 18, 0, 4, 7, 5, 8, 0};
    ivec_scalar_mult(vucf, 6);
    ok(array_equals(vucf, vucfans, 9), "scalar mult of vucf by 6");

    int zeros[VTO100_SIZE]; // all zeros
    for (int i = 0; i < VTO100_SIZE; i++) {
        zeros[i] = 0;
    }
    ivec_scalar_mult(vto100, 0);
    ok(array_equals(vto100, zeros, VTO100_SIZE), "scalar mult of vto100 by 0");
    return exit_status();
}
```

Figure 4: Tests for `ivec_scalar_mult`, which use the test data in Figure 1 on page 3.

To run our tests, copy the files `tap.c`, `tap.h`, `ivec_scalar_mult.h`, `test_ivec_scalar_mult.c`, `test_data_ivec.c`, and `test_data_ivec.h` into your project/directory and add them to your project (if you are using Code::Blocks). Then write your code for `ivec_scalar_mult` in a file `ivec_scalar_mult.c`.

Hint: if you want to use your solution for `ivec_len` in your code, then also copy `ivec_len.c` and `ivec_len.h` to your project/directory and add them to your project.

Remember to turn in your C source code file `ivec_scalar_mult.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.

3. (20 points) [Programming] In C, write a function

```
extern void ivec_add(int target[], const int src[]);
```

that takes two ivecs, `target` and `src`, such that the length of `src` is at least as great as that of `target` (as measured by `ivec_len`). The function `ivec_add` modifies `target`'s elements so that at each index i (where $0 \leq$ `i` and `i` is strictly less than `ivec_len(target)`), i.e., `target[i]` `target[i]` becomes `target[i]` plus `src[i]`. The assumption about the length of `src` means that `src[i]` is always defined for such an index `i`. There are tests in Figure 5 on the following page and a detailed specification is found in `ivec_add.h`.

To run our tests, copy the files `tap.c`, `tap.h`, `ivec_add.h`, `test_ivec_add.c`, `test_data_ivec.c`, and `test_data_ivec.h` into your project/directory and add them to your project (if you are using Code::Blocks). Then write your code for `ivec_add` in a file `ivec_add.c`.

Hint: if you want to use your solution for `ivec_len` in your code, then also copy `ivec_len.c` and `ivec_len.h` to your project/directory and add them to your project.

Remember to turn in your C source code file `ivec_add.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.

4. (25 points) [Programming] In C, write a function

```
extern int ivec_cmp(const int left[], const int right[]);
```

that takes two ivecs, `left` and `right`, and does a lexicographical comparison, returning -1 if `left` is strictly less than `right`, returning 0 if they are equal, and returning +1 if `left` is strictly greater than `right`. Note that these comparisons only involve the elements up to the first 0 in either ivec's array. See the file `ivec_cmp.h` for a detailed specification. There are tests in Figure 6 on page 9.

To run our tests, copy the files `tap.c`, `tap.h`, `ivec_cmp.h`, `test_ivec_cmp.c`, `test_data_ivec.c`, and `test_data_ivec.h` into your project/directory and add them to your project (if you are using Code::Blocks). Then write your code for `ivec_cmp` in a file `ivec_cmp.c`.

Hint: C does not have a built in `min` function.

Remember to turn in your C source code file `ivec_cmp.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.

5. (30 points; extra credit) [Design] Design a dot product operation on ivecs. Write a header file with a specification, implement it, and write unit tests, following our example. Hand in your header file, implementation file, testing file, and the output of your tests. Your submission will be judged on the following basis:

   - How sensible the design of the operation is, including the specification in the header file (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.

```
// $Id: test_ivec_add.c,v 1.8 2017/04/01 13:23:00 leavens Exp leavens $
// Compile with:
// gcc tap.c ivec_len.c ivec_add.c test_data_ivec.c test_ivec_add.c -o test_ivec_add
#include "tap.h"
#include "ivec_add.h"

// declarations of test data (see test_data_ivec.c) used below
#include "test_data_ivec.h"

// testing for ivec_add().
int main() {
    plan(7);
    int v[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    int c[] = {3, 5, 7, 4, 1, 6, 0, 0, 7};
    ivec_add(v, c);
    int vans[] = {12, 12, 15, 10, 10, 0, 5, 4, 0};
    ok(array_equals(v, vans, 9), "v = v + c");

    // using test data from ivec_test_data.c below...
    init_test_data();

    int vempty[] = {0};
    int v0a[] = {0};
    ivec_add(vempty, vempty);
    ok(array_equals(vempty, v0a, 1), "vempty = vempty + vempty");

    int v0ans[] = {0};
    ivec_add(v0, v1);
    ok(array_equals(v0, v0ans, 1), "v0 = v0 + v1");

    int v3223ans[] = {6, 3, 4, 6, 0};
    ivec_add(v3223, v3123);
    ok(array_equals(v3223, v3223ans, 5), "v3223 = v3223 + v3123");

    int v100[] = {100, 100, 100, 100, 0};
    int v3123ans[] = {102, 101, 102, 103, 0};
    ivec_add(v2123, v100);
    ok(array_equals(v2123, v3123ans, 5), "v2123 = v2123 + v100");

    int vucfans[] = {11, 4, 5, 0, 4, 7, 5, 8, 0};
    ivec_add(vucf, v3223ext);
    ok(array_equals(vucf, vucfans, 9), "vucf += v3223ext");

    int vto100by2[VTO100_SIZE];
    for (int i = 0; i < VTO100_SIZE-1; i++) {
        vto100by2[i] = 2*(i+1);
    }
    vto100by2[VTO100_SIZE-1] = 0;
    ivec_add(vto100, vto100);
    ok(array_equals(vto100, vto100by2, VTO100_SIZE), "vto100 += vto100");
    return exit_status();
}
```

Figure 5: Tests for `ivec_add`, which use the test data in Figure 1 on page 3.

```
// $Id: test_ivec_cmp.c,v 1.6 2017/04/01 19:22:59 leavens Exp leavens $
// Compile with:
// gcc tap.c ivec_len.c ivec_cmp.c test_data_ivec.c test_ivec_cmp.c -o test_ivec_cmp

#include "tap.h"
#include "ivec_cmp.h"

// declarations of test data (see test_data_ivec.c) used below
#include "test_data_ivec.h"

// testing for ivec_cmp().
int main() {
    plan(27);
    int v[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    ok(ivec_cmp(v, v) == 0, "ivec_cmp(v, v) == 0");
    int w[] = {10, 8, 0};  // tests involving w and x are new
    ok(ivec_cmp(v, w) == -1, "ivec_cmp(v, w) == +1");
    ok(ivec_cmp(w, v) == +1, "ivec_cmp(v, w) == +1");
    int x[] = {10, 8, -9, 0};
    ok(ivec_cmp(w, x) == -1, "ivec_cmp(w, x) == -1");
    ok(ivec_cmp(x, w) == +1, "ivec_cmp(x, w) == +1");
    // using test data from ivec_test_data.c below
    init_test_data();
    ok(ivec_cmp(v0, v0) == 0, "ivec_cmp(v0) == 0");
    ok(ivec_cmp(v0, v1) == -1, "ivec_cmp(v0, v1) == -1");
    ok(ivec_cmp(v1, v0) == +1, "ivec_cmp(v1, v0) == +1");
    ok(ivec_cmp(v1, v1) == 0, "ivec_cmp(v1, v1) == 0");
    ok(ivec_cmp(v1, v3223) == -1, "ivec_cmp(v1, v3223) == -1");
    ok(ivec_cmp(v3223, v1) == +1, "ivec_cmp(v3223, v1) == +1");
    ok(ivec_cmp(v3223, v3223) == 0, "ivec_cmp(v3223, v3223) == 0");
    ok(ivec_cmp(v3223, v3123) == +1, "ivec_cmp(v3223, v3123) == +1");
    ok(ivec_cmp(v3123, v3223) == -1, "ivec_cmp(v3123, v3223) == -1");
    ok(ivec_cmp(v3223, v2123) == +1, "ivec_cmp(v3223, v2123) == +1");
    ok(ivec_cmp(v2123, v3223) == -1, "ivec_cmp(v2123, v3223) == -1");
    ok(ivec_cmp(v3223, v3323) == -1, "ivec_cmp(v3223, v3323) == -1");
    ok(ivec_cmp(v3323, v3223) == +1, "ivec_cmp(v3323, v3223) == +1");
    ok(ivec_cmp(v3323, v3224) == +1, "ivec_cmp(v3323, v3224) == +1");
    ok(ivec_cmp(v3224, v3323) == -1, "ivec_cmp(v3224, v3323) == -1");
    ok(ivec_cmp(v3223, v3223ext) == 0, "ivec_cmp(v3223, v3223ext) == 0");
    ok(ivec_cmp(v3223ext, v3223) == 0, "ivec_cmp(v3223ext, v3223) == 0");
    ok(ivec_cmp(v0, vucf) == -1, "ivec_cmp(v0, vucf) == -1");
    ok(ivec_cmp(vucf, v0) == +1, "ivec_cmp(vucf, v0) == +1");
    ok(ivec_cmp(vto100, vto100) == 0, "ivec_cmp(vto100, vto100) == 0");
    int u2[] = {0, 1, 2, 0};
    ok(ivec_cmp(u2, vto100) == -1, "ivec_cmp(u2, vto100) == -1");
    ok(ivec_cmp(vto100, u2) == +1, "ivec_cmp(vto100, u2) == +1");
    return exit_status();
}
```

Figure 6: Tests for `ivec_cmp`, which use the test data in Figure 1 on page 3.

String Set Problems

Sets are a useful concept in many areas of Mathematics and science, particularly in Computer Science. For example, a database relation is essentially a set of records. In this section you will implement several operations on string sets, which we represent as arrays of strings. Since strings are represented in C as arrays of characters containing a null character, we will work with formal parameters such as the following

```
const char *const strs[]
```

which says that `strs` is an array, each of whose elements is a **`char *`**, i.e., a string. Both the string pointers and the characters in the string are declared to be **`const`**, meaning that they cannot be changed through assignment to `strs[i]` (for some index `i`) or through assignment to `*(strs[i])` (or by an equivalent double subscript).

The elements of such an array represent the elements of the set in the sense that each string that is an element of the array is an element of the set.

The tests for the string set problems share some data, which is found in `test_data_string_sets.c` (with header file `test_data_string_sets.h`), as shown in Figure 7. These files are provided in `hw6-tests.zip`. For each string set problem you will need to copy both `test_data_string_sets.c` and `test_data_string_sets.h` to your project/directory.

```
// $Id: test_data_string_sets.c,v 1.1 2017/03/20 02:41:50 leavens Exp $

#include <string.h>
#include "test_data_string_sets.h"
// data for testing string sets

const char *const s1[] = {"hello"};
const char *const s5[] = {"The", "abbey", "was", "clammy", "today"};
const char *const Ada_Lovelace_quote[LLQ_LENGTH] = {
    // quoted in Wikipedia, original from an article in 1842
    "[The", "Analytical", "Engine]", "might", "act", "upon", "other", "things",
    "besides", "number,", "were", "objects", "found", "whose", "mutual",
    "fundamental", "relations", "could", "be", "expressed", "by", "those",
    "of", "the", "abstract", "science", "of", "operations,", "and", "which",
    "should", "be", "also", "susceptible", "of", "adaptations", "to", "the",
    "action", "of", "the", "operating", "notation", "and", "mechanism", "of",
    "the", "engine...", "Supposing,", "for", "instance,", "that", "the",
    "fundamental", "relations", "of", "pitched", "sounds", "in", "the",
    "science", "of", "harmony", "and", "of", "musical", "composition",
    "were", "susceptible", "of", "such", "expression", "and", "adaptations,",
    "the", "engine", "might", "compose", "elaborate", "and", "scientific",
    "pieces", "of", "music", "of", "any", "degree", "of", "complexity",
    "or", "extent"};
```

Figure 7: Test data for string sets, in the file `test_data_string_sets.c`. There is also a header file `test_data_string_sets.h` (not shown) that declares these arrays.

---

6. (20 points) [Programming] In C, write a function

   ```
   extern bool is_member(const char *sought, const char * const strs[],
                         const int strs_sz);
   ```

   that takes a C string `sought`, an array of C strings `strs`, and an int `strs_sz`, which gives the size of `strs`. The function `is_member` returns true just when the string `sought` is equal to a string in `strs`; that is if there is some index `i` such that `0 <= i` and `i < strs_sz`, such that `sought` equals `strs[i]`. See the file `is_member.h` for a detailed specification. There are tests in Figure 8.

   Note that two strings are *equal* if and only if they have the same length and the same characters in the same order; this is the comparison given by `strcmp`.

```
// $Id: test_is_member.c,v 1.3 2017/04/02 19:01:08 leavens Exp leavens $
// Compile with:
// gcc tap.c is_member.c test_data_string_sets.c test_is_member.c -o test_is_member
#include <stdbool.h>
#include "tap.h"
#include "test_data_string_sets.h"

extern bool is_member(const char *sought, const char * const strs[],
                      const int strs_sz);

// testing for is_member()
int main() {
    // for some reason, not calling plan() makes the tests effective
    // against implementations of is_member that don't return anything...
    // plan(11);
    ok(!is_member("foo", s1, 1), "!is_member(\"foo\", s1, 1)");
    ok(is_member("hello", s1, 1) == true, "is_member(\"hello\", s1, 1)");
    ok(is_member("frank", s1, 1) == false, "!is_member(\"frank\", s1, 1)");
    ok(is_member("", s1, 1) == false, "!is_member(\"\", s1, 1)");
    ok(is_member("abbey", s5, 5) == true, "is_member(\"abbey\", s5, 5)");
    ok(is_member("today", s5, 5) == true, "is_member(\"today\", s5, 5)");
    ok(is_member("Today", s5, 5) == false, "!is_member(\"Today\", s5, 5)");
    ok(is_member("Python", s5, 5) == false, "!is_member(\"Python\", s5, 5)");
    ok(is_member("relations", Ada_Lovelace_quote, LLQ_LENGTH) == true,
       "is_member(\"relations\", Ada_Lovelace_quote, LLQ_LENGTH)");
    ok(is_member("the", Ada_Lovelace_quote, LLQ_LENGTH) == true,
       "is_member(\"the\", Ada_Lovelace_quote, LLQ_LENGTH)");
    ok(is_member("UCF", Ada_Lovelace_quote, LLQ_LENGTH) == false,
       "!is_member(\"UCF\", Ada_Lovelace_quote, LLQ_LENGTH)");
    return exit_status();
}
```

Figure 8: Tests for `is_member`, which use the test data in Figure 7 on the preceding page.

To run our tests, copy the files `tap.c`, `tap.h`, `is_member.h`, `test_is_member.c`, `test_data_string_sets.c`, and `test_data_string_sets.h` into your project/directory and add them to your project (if you are using Code::Blocks). Then write your code for `is_member` in a file `is_member.c`.

Hint: to compare strings, use `strcmp` from the C library, using the declarations in the built-in header file `string.h`. Don't use `==` to compare C strings, as that will only compare their addresses, not their contents, which is not what is meant by "equals" for strings.

Remember to turn in your C source code file `is_member.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.

7. (20 points) [Programming] When representing a set, one can save space by eliminating duplicate entries. This problem is to write a function to check if the representation of a string set does indeed have no duplicates.

   In C, write a function

   ```
   extern bool no_dups(const char * const strs[], const int strs_sz);
   ```

   that takes an array of C strings `strs`, and an int `strs_sz`, which gives the size of `strs`. The function no_dups returns true just when there are not two distinct elements of `strs` that are equal to each other; that is there are no duplicates in `strs` if there is no pair of indexes `i` and `j` such that `0 <= i`, `i < j`, and `j < strs_sz`, such that `strs[i]` equals `strs[j]`. See the file `no_dups.h` for a detailed specification. There are tests in Figure 9.

```
// $Id: test_no_dups.c,v 1.4 2017/04/02 22:10:03 leavens Exp leavens $
// Compile with:
// gcc tap.c test_data_string_sets.c is_member.c no_dups.c test_no_dups.c -o test_no_dups
#include <stdbool.h>
#include "tap.h"
#include "test_data_string_sets.h"
#include "no_dups.h"

// testing for is_no_dups()
int main() {
    // plan(5);
    const char *const sset0[1] = {""};
    const char *const sset[] = {"Now", "is", "the", "time"};
    const char *const sset2[] = {"Now", "is", "the", "time", "yes:", "Now"};
    // create an array sset3 that doesn't have just literals in it...
    char str1[4]; char str2[4];
    str1[0] = 'N'; str1[1] = 'o'; str1[2] = 'w'; str1[3] = '\0';
    str2[0] = 'N'; str2[1] = 'o'; str2[2] = 'w'; str2[3] = '\0';
    const char *const sset3[3] = {str1, "Spacer", str2};
    const char *const sset4[3] = {"Spacer", str1, str2};
    ok(no_dups(sset0, 0), "no_dups(sset0, 0)");
    ok(no_dups(sset0, 1), "no_dups(sset0, 1)");
    ok(no_dups(sset, 4), "no_dups(sset, 4)");
    ok(!no_dups(sset2, 6), "!no_dups(sset2, 6)");
    ok(!no_dups(sset3, 3), "!no_dups(sset3, 3)");
    ok(!no_dups(sset4, 3), "!no_dups(sset4, 3)");
    // the following use test data from test_data_string_sets.c
    ok(no_dups(s1, 1), "no_dups(s1, 1)");
    ok(no_dups(s5, 5), "no_dups(s5, 5)");
    ok(!no_dups(Ada_Lovelace_quote, LLQ_LENGTH), "!no_dups(Ada_Lovelace_quote, LLQ_LENGTH)");
    return exit_status();
}
```

Figure 9: Tests for no_dups, which use the test data in Figure 7 on page 10.

To run our tests, copy the files `tap.c`, `tap.h`, `no_dups.h`, `test_no_dups.c`, `test_data_string_sets.c`, and `test_data_string_sets.h` into your project/directory and add them to your project (if you are using Code::Blocks). Then write your code for no_dups in a file `no_dups.c`.

Hint: you may want to use your solution to the `is_member` problem to help solve this problem. If you do that, you will need to copy `is_member.c` and `is_member.h` into the project for this problem and add them into this project. You would also need to include `is_member.h` into `no_dups.c` to have a declaration for `is_member` available to the compiler when it compiles `no_dups.c`.

Remember to turn in your C source code file `no_dups.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.

8. (30 points; extra credit) [Design] Design a console program in C that prompts the user for some strings that are used to form a string set, and then prompts for and answers queries from the user about whether various strings are in the set. Write your design in the form of a problem to be solved, with examples, and then solve it in C and test it. Record the problem in a `.txt` file. 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.

   Hint: if you want to record tests as we have done in class, you may want to use our files `testing_io.c` and `testing_io.h`, which available from the course code examples page.

9. (30 points; extra credit) [Design] Design string set operations to perform unions and intersections of string sets. Record your design in header files (i.e., in `.h` files), with specifications. Then implement your specifications in C code and test them using tests that you write yourself. 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: 125. Total extra credit points: 90.