COP 3223H — Honors Introduction to Programming with C April 13, 2019

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 `ivdim`), 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 `ivdim`), 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 you copied into the directory 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`. Be sure that the new file is made part of the project. (If you fail to do that when creating the file, you can 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`). If you add write or change any modules (.c and .h files), then turn those in as well so we know what all the code is that is run.

Problems

1. (40 points) [Programming] This problem is about palindromes. In English a *palindrome* is a text whose alphabetic characters are the same (ignoring case) forwards and backwards. An example is "Madam,

I'm Adam." Note that the case of characters and any punctuation and white space (blanks, newlines, etc.) are ignored. Thus "Aa" is a palindrome, but "ant" is not.

In C, write a program that will be named `palindrome` (or `palindrome.exe` on Windows) that prompts the user with "`line? `" on standard output, then reads a line of text of no more than 500 characters from standard input. If the line of text is a palindrome (by the above definition) then the program prints "`Yes, it is a palindrome.`" followed by a newline on standard output, otherwise it prints "`No, it is not a palindrome.`" on standard output. The program must exit with a success code, provided no errors occured during its run.

The following are examples of two-line dialogs with the program, with the input from stdin following "`line? `" on the first line of the dialog and all the other characters going to standard output, with a final newline at the end of the entire dialog.

```
line? a
Yes, it is a palindrome.

line? Able was I ere I saw Elba
Yes, it is a palindrome.

line? Never odd or even
Yes, it is a palindrome.

line? XY
No, it is not a palindrome.

line? Neither a borrower, nor a lender, be!
No, it is not a palindrome.
```

For testing, since this is a stand-alone program, you can test it in several ways.

The easiest way is to run the program several times, recording the dialog you have with it. This would involve testing your program on each of the inputs given in the files whose names are of the form "`test_palindrome*.in`" (with * replaced by a number) found in the `hw6-tests.zip` file. Hand in the recorded dialogs from all of those tests.

Alternatively, from the command line, you can compile your program and place it in a file (`palindrome.exe` on Windows and `palindrome` on a Mac or Linux), so that "`./palindrome`" will run your program. Then you would also compile the file `test_palindrome.c` as a program. Then run `test_palindrome` in a directory containing your palindrome program and all of the files named "`test_palindrome*.in`" and "`test_palindrome*.expected`" from the `hw6-tests.zip` file.

Remember to turn in your C source code file for your program, and the output of the tests (run using one of the ways described above). These should be submitted to webcourses as ASCII text files that you upload. If you write any additional modules (`.c` and `.h` files), then these should also be uploaded.

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.

```
/* 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`.

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

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

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

```
// $Id: test_ivdim.c,v 1.1 2019/03/31 00:48:12 leavens Exp leavens $
// Compile with
// gcc tap.c ivdim.c test_data_ivec.c test_ivdim.c -o test_ivdim
#include "tap.h"
#include "ivdim.h"

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

// testing for ivdim().
int main() {
    plan(11);
    int v[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    ok(ivdim(v) == 5, "ivdim(v) == 5");
    // using test data from ivec_test_data.c below
    init_test_data();
    ok(ivdim(v0) == 0, "ivdim(v0) == 0");
    ok(ivdim(v1) == 1, "ivdim(v1) == 1");
    ok(ivdim(v3223) == 4, "ivdim(v3223) == 4");
    ok(ivdim(v3123) == 4, "ivdim(v3123) == 4");
    ok(ivdim(v2123) == 4, "ivdim(v2123) == 4");
    ok(ivdim(v3323) == 4, "ivdim(v3123) == 4");
    ok(ivdim(v3224) == 4, "ivdim(v3224) == 4");
    ok(ivdim(v3223ext) == 4, "ivdim(v3224ext) == 4");
    ok(ivdim(vucf) == 3, "ivdim(vucf) == 3");
    int d100 = ivdim(vto100);
    ok(d100 == 100, "ivdim(vto100) == 100, result was %d", d100);
    return exit_status();
}
```

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

To run our tests, copy the files `tap.c`, `tap.h`, `ivdim.h`, `test_ivdim.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 `ivdim` in a file `ivdim.c`. Run the tests in `test_ivdim.c`.

Remember to turn in your C source code file `ivdim.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 ivsmult(const int s, int target[]);
   ```

   that takes an int `s`, and an ivec, `target`, and modifies `target`'s elements such that each of the elements before the first 0 in `target` becomes `s` 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 `ivsmult.h` (shown above).

   To run our tests, copy the files `tap.c`, `tap.h`, `ivsmult.h`, `test_ivsmult.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 `ivsmult` in a file `ivsmult.c`.

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

   Remember to turn in your C source code file `ivsmult.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. (20 points) [Programming] In C, write a function

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

   that takes two ivecs, `target` and `src`, with the same dimension (as measured by `ivdim`). The function `ivadd` modifies `target`'s elements so that at each index i (where $0 \leq$ `i` and `i` is strictly less than `ivdim(target)`), `target[i]` becomes `target[i]` plus `src[i]`. There are tests in Figure 5 on page 9 and a detailed specification is found in `ivadd.h`.

   To run our tests, copy the files `tap.c`, `tap.h`, `ivadd.h`, `test_ivadd.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 `ivadd` in a file `ivadd.c`.

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

   Remember to turn in your C source code file `ivadd.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. (25 points) [Programming] In C, write a function

   ```
   extern int ivcmp(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. Lexicographic order is the ordering used in a dictionary. So, `left` is considered to be strictly less than `right` if for the first index `j` that is smaller than the smallest index of either ivec that is 0 and at which `left[j] != right[j]`, the inequality `left[j] < right[j]` holds, or if all indexes that are legal in both they are equal but the dimension of `left` is less than that of `right`. See the file `ivcmp.h` for a detailed specification. There are tests in Figure 6 on page 10.

   To run our tests, copy the files `tap.c`, `tap.h`, `ivcmp.h`, `test_ivcmp.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 `ivcmp` in a file `ivcmp.c`.

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

   Remember to turn in your C source code file `ivcmp.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: test_ivsmult.c,v 1.2 2019/03/31 02:50:54 leavens Exp leavens $
// Compile with:
// gcc tap.c ivdim.c ivsmult.c test_data_ivec.c test_ivsmult.c -o test_ivsmult
#include <stdbool.h>
#include "tap.h"
#include "ivsmult.h"
// declarations of test data (see test_data_ivec.c) and array_equals used below
#include "test_data_ivec.h"

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

    int v0ans[] = {0};
    ivsmult(5021, v0);
    ok(array_equals(v0, v0ans, 1), "ivsmult of 5021 * v0");

    int v1ans[] = {45, 0};
    ivsmult(45, v1);
    ok(array_equals(v1, v1ans, 2), "ivsmult of 45 * v1");

    int v3223ans[] = {6, 4, 4, 6, 0};
    ivsmult(2, v3223);
    ok(array_equals(v3223, v3223ans, 5), "ivsmult of 2 * v3223");

    int v3123ans[] = {15, 5, 10, 15, 0};
    ivsmult(5, v3123);
    ok(array_equals(v3123, v3123ans, 5), "ivsmult of 5 * v3123");

    int v2123ans[] = {200, 100, 200, 300, 0};
    ivsmult(100, v2123);
    ok(array_equals(v2123, v2123ans, 5), "ivsmult of 100 * v2123");

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

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

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

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

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

```c
// $Id: test_ivadd.c,v 1.2 2019/03/31 02:50:54 leavens Exp leavens $
// Compile with:
// gcc tap.c ivdim.c ivadd.c test_data_ivec.c test_ivadd.c -o test_ivadd
#include <stdbool.h>
#include "tap.h"
#include "ivadd.h"
// declarations of test data (see test_data_ivec.c) and array_equals used below
#include "test_data_ivec.h"

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

    int v97869[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    int v13245[] = {1, 3, 2, 4, 5, 0};
    ivadd(v97869, v13245);
    int vans2[] = {10, 10, 10, 10, 14, 0, 5, 4, 0};
    ok(array_equals(v97869, vans2, 9), "v97869 = v97869 + v13245");
    // using test data from ivec_test_data.c below...
    init_test_data();

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

    int v3223ans[] = {6, 3, 4, 6, 0};
    ivadd(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};
    ivadd(v2123, v100);
    ok(array_equals(v2123, v3123ans, 5), "v2123 = v2123 + v100");

    int vfan[] = {3, 2, 6, 0};
    int vucfans[] = {11, 4, 9, 0};
    ivadd(vucf, vfan);
    ok(array_equals(vucf, vucfans, 4), "vucf += vfan");

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

    return exit_status();
}
```

Figure 5: Tests for `ivadd`, which use the test data in Figure 1 on page 4.

```
// $Id: test_ivcmp.c,v 1.5 2019/04/13 15:52:24 leavens Exp leavens $
// Compile with:
// gcc tap.c ivdim.c ivcmp.c test_data_ivec.c test_ivcmp.c -o test_ivcmp

#include "tap.h"
#include "ivcmp.h"

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

// testing for ivcmp().
int main() {
    plan(33);
    int v[] = {9, 7, 8, 6, 9, 0, 5, 4, 0};
    ok(ivcmp(v, v) == 0, "ivcmp(v, v) == 0");
    int w[] = {10, 8, 0};  // tests involving w and x are new
    ok(ivcmp(v, w) == -1, "ivcmp(v, w) == -1");
    ok(ivcmp(w, v) == +1, "ivcmp(w, v) == +1");
    int x[] = {10, 8, -9, 0};
    ok(ivcmp(w, x) == -1, "ivcmp(w, x) == -1");
    ok(ivcmp(x, w) == +1, "ivcmp(x, w) == +1");
    // using test data from ivec_test_data.c below
    init_test_data();
    ok(ivcmp(v0, v0) == 0, "ivcmp(v0, v0) == 0");
    ok(ivcmp(v0, v1) == -1, "ivcmp(v0, v1) == -1");
    ok(ivcmp(v1, v0) == +1, "ivcmp(v1, v0) == +1");
    ok(ivcmp(v1, v1) == 0, "ivcmp(v1, v1) == 0");
    ok(ivcmp(v1, v3223) == -1, "ivcmp(v1, v3223) == -1");
    ok(ivcmp(v3223, v1) == +1, "ivcmp(v3223, v1) == +1");
    ok(ivcmp(v3223, v3223) == 0, "ivcmp(v3223, v3223) == 0");
    ok(ivcmp(v3223, v3123) == +1, "ivcmp(v3223, v3123) == +1");
    ok(ivcmp(v3123, v3223) == -1, "ivcmp(v3123, v3223) == -1");
    ok(ivcmp(v3223, v2123) == +1, "ivcmp(v3223, v2123) == +1");
    ok(ivcmp(v2123, v3223) == -1, "ivcmp(v2123, v3223) == -1");
    ok(ivcmp(v3223, v3323) == -1, "ivcmp(v3223, v3323) == -1");
    ok(ivcmp(v3323, v3223) == +1, "ivcmp(v3323, v3223) == +1");
    ok(ivcmp(v3323, v3224) == +1, "ivcmp(v3323, v3224) == +1");
    ok(ivcmp(v3224, v3323) == -1, "ivcmp(v3224, v3323) == -1");
    ok(ivcmp(v3223, v3223ext) == 0, "ivcmp(v3223, v3223ext) == 0");
    ok(ivcmp(v3223ext, v3223) == 0, "ivcmp(v3223ext, v3223) == 0");
    ok(ivcmp(v0, vucf) == -1, "ivcmp(v0, vucf) == -1");
    ok(ivcmp(vucf, v0) == +1, "ivcmp(vucf, v0) == +1");
    ok(ivcmp(vto100, vto100) == 0, "ivcmp(vto100, vto100) == 0");
    int u2[] = {0, 1, 2, 0};
    ok(ivcmp(u2, vto100) == -1, "ivcmp(u2, vto100) == -1");
    ok(ivcmp(vto100, u2) == +1, "ivcmp(vto100, u2) == +1");
    int ce1[] = {3, -5, 2, 0};
    int ce2[] = {3, 3, 3, 5, 0};
    ok(ivcmp(ce1, ce2) == -1, "ivcmp(ce1, ce2) == -1");
    ok(ivcmp(ce2, ce1) == +1, "ivcmp(ce2, ce1) == +1");
    int ce3[] = {3, 2, 2, 0};
    int ce4[] = {3, 3, 2, 5, 0};
    ok(ivcmp(ce3, ce4) == -1, "ivcmp(ce3, ce4) == -1");
    ok(ivcmp(ce4, ce3) == +1, "ivcmp(ce4, ce3) == +1");
    int ce5[] = {3, 2, 2, 4, 9, 8, 7, 0};
    int ce6[] = {3, 3, 2, 4, 0};
    ok(ivcmp(ce5, ce6) == -1, "ivcmp(ce5, ce6) == -1");
    ok(ivcmp(ce6, ce5) == +1, "ivcmp(ce6, ce5) == +1");
    return exit_status();
}
```

Figure 6: Tests for `ivcmp`, which use the test data in Figure 1 on page 4.

6. (30 points; extra credit) [Design] Design a cross 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.

Interval Problems

Intervals, such as $[x, y]$ can be used to represent a set of contiguous real values. An interval might represent the result of an experimental measurement with limited precision. In this section you will implement several operations on intervals, which we represent pointers to two-element arrays of doubles, with the 0th element representing the lower bound of an interval, and the 1th element representing the interval's upper bound. This is describe in the header file `interval.h`, shown in Figure 7 on the following page.

The functions described in `interval.h` are implemented for you in the file `interval.c`, which is provided in this homework's testing file: `hw6-tests.zip`.

```
#include <string.h>
#include <stdbool.h>

// Intervals are represented by 2 element arrays
// with the 0th element the lower bound
// and the 1th element the upper bound.
typedef double *interval;

// requires: neither lb nor ub is NaN
// ensures: result is either NULL (if malloc failed)
//          or a pointer to a 2-element array iv such that
//          *iv == lb and *(iv+1) == ub.
extern interval iMake(double lb, double ub);

// This is a version of iMake intended for testing.
// requires: neither lb nor ub is NaN
// effect: call iMake(lb, ub), and if that returns non-NULL,
//         then that result is the result of this function.
//         However, if iMake(lb, ub) returns NULL,
//         then print an error message on stderr and abort the program.
extern interval callIMake(double lb, double ub);

// requires: iv != NULL
// ensures: result is the lower bound of iv (*iv).
extern double iLower(interval iv);

// requires: iv != NULL
// ensures: result is the upper bound of iv (*iv+1).
extern double iUpper(interval iv);

// requires: iv != NULL
// ensures: result is true iff iLower(iv) > upper(iv).
extern bool iEmpty(interval iv);

// requires: iv1 != NULL && iv2 != NULL
// ensures: result is true iff iLower(iv1) == iLower(iv2)
//          and iUpper(iv1) == iUpper(iv2)
extern bool iEqual(interval iv1, interval iv2);

// requires: iv1 != NULL && iv2 != NULL
// ensures: result is true iff iLower(iv1) ~~ iLower(iv2)
//          and iUpper(iv1) ~~ iUpper(iv2)
//          where ~~ denotes approximate equality to within epsilon
extern bool iApproxEqual(interval iv1, interval iv2, double eps);

// requires: iv != NULL
// ensures: result is either NULL
//          or a pointer to a string of the form "[lb, ub]"
//          where lb and ub are strings representing iLower(iv)
//          and iUpper(iv) respectively.
//          Formatting for both lb and ub uses the "%f" format.
extern const char*iString(interval iv);
```

Figure 7: The header file `interval.h`.

7. (20 points) [Programming] In C, write the `iMember` function described in the header file `iMember.h` shown below:

   ```
   #include "interval.h"

   // requires: x is not NaN and iv is not NULL
   // ensures: result is true iff x is
   //          between iLower(iv) and iUpper(iv), inclusive.
   extern bool iMember(double x, const interval iv);
   ```

   The `iMember` function takes a double, `x`, and an interval `iv`, and return true just when `x` is between the lower and upper bounds of the interval, inclusive. Note that you can assume that `x` is not NaN. There are tests in Figure 8.

```
// $Id: test_iMember.c,v 1.1 2019/04/13 18:27:59 leavens Exp leavens $
// Compile with:
// gcc tap.c interval.c iMember.c test_iMember.c -o test_iMember
#include <stdbool.h>
#include "tap.h"
#include "iMember.h"

// testing for iMember()
int main() {
    plan(16);
    interval iv12 = callIMake(1.0, 2.0);
    ok(iMember(1.0, iv12), "1.0 in iv12");
    ok(iMember(2.0, iv12), "2.0 in iv12");
    ok(iMember(1.4, iv12), "1.4 in iv12");
    ok(iMember(3.1415, iv12) == false, "3.1415 not in iv12");
    ok(iMember(955.324e10, iv12) == false, "955.324e10 not in iv12");
    interval bigI = callIMake(7.95, 3024.2947);
    ok(!iMember(5932.2, bigI), "5932.2 not in bigI");
    ok(iMember(1.99, iv12), "1.99 in iv12");
    ok(!iMember(0.99, iv12), "0.99 not in iv12");
    ok(!iMember(2.001, iv12), "2.001 not in iv12");
    interval near4 = callIMake(3.999, 4.001);
    ok(iMember(4.0, near4), "4.0 in near4");
    ok(iMember(3.999, near4), "3.999 in near4");
    ok(iMember(4.001, near4), "4.001 in near4");
    ok(iMember(3.99901, near4), "3.99901 in near4");
    ok(iMember(4.0009, near4), "4.0009 in near4");
    ok(!iMember(3.98, near4), "3.98 not in near4");
    ok(!iMember(4.002, near4), "4.002 not in near4");
    return exit_status();
}
```

Figure 8: Tests for `iMember`.

   To run our tests, copy the files `tap.c`, `tap.h`, `interval.h`, `interval.c`, `iMember.h`, and `test_iMember.c`, to your project (if you are using Code::Blocks). Then write your code for `iMember` in a file `iMember.c`.

   Remember to turn in your C source code file `iMember.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. (20 points) [Programming] In C, write a function `iAdd` as defined in `iAdd.h` below:

The function `iAdd` takes two intervals and returns their sum, which is the smallest interval containing the sum of any numbers in the two interval arguments. The function can return `NULL` when `malloc` returns `NULL`. There are tests in Figure 9 on the following page.

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

Remember to turn in your C source code file `iAdd.c` and the output of running our tests, along with any other files you create to solve the problem. Code and also the output of running our tests should be submitted to webcourses as ASCII text files that you upload.

9. (30 points; extra credit) [Design] Design a console program in C acts as a calculator for intervals. It prompts for and reads intervals, and then prompts for an operation, which it performs and prints the interval to stdout. 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.

Points

This homework's total points: 165. Total extra credit points: 60.

```
// $Id: test_iAdd.c,v 1.2 2019/04/01 02:08:35 leavens Exp leavens $
// Compile with:
// gcc tap.c interval.c iAdd.c test_iAdd.c -o test_iAdd
#include <stdlib.h>
#include <stdbool.h>
#include "tap.h"
#include "iAdd.h"

// requires: iv1 != NULL && iv2 != NULL;
// effect: call iAdd(iv1, iv2) and if that returns NULL, abort;
//         otherwise return the result of the addition.
static interval callIAdd(const interval iv1, const interval iv2) {
    interval ret = iAdd(iv1, iv2);
    if (ret == NULL) {
        BAIL_OUT("Call of iAdd(%s, %s) returned NULL", iv1, iv2);
        exit(EXIT_FAILURE);
    }
    return ret;
}

// requires: left != NULL && right != NULL && expected != NULL;
//           and eps is not NaN.
// effect: assert that left+right is approximately equal to expected,
//         within eps.
static void testAdd(interval left, interval right, interval expected, double eps) {
    interval res = callIAdd(left, right);
    ok(iApproxEqual(res, expected, eps),
       "%s+%s ~~ %s", iString(left), iString(right),iString(res));
}

// testing for iAdd()
int main() {
    // plan(13);
    interval zero = callIMake(0.0, 0.0);
    interval one = callIMake(0.995, 1.005);
    interval negone = callIMake(-1.005, -0.995);
    interval nearzero = callIMake(-0.01, 0.01);
    interval two = callIMake(1.6, 2.4);
    interval negtwo = callIMake(-2.4, -1.6);
    interval three = callIMake(2.595, 3.405);
    interval ten = callIMake(9.0, 11.0);
    interval thirteen = callIMake(11.595, 14.405);
    double eps = 0.001;
    testAdd(zero, one, one, eps);
    testAdd(one, negone, nearzero, eps);
    testAdd(two, zero, two, eps);
    testAdd(two, negtwo, zero, 0.85);
    testAdd(one, two, three, eps);
    testAdd(three, ten, thirteen, eps);
    return exit_status();
}
```

Figure 9: Tests for `iAdd`.