COP 3223H — Honors Introduction to Programming with C March 7, 2019

Homework 5: Conditionals and Loops 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 `hw5-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 function you should be writing (such as `which_sort`), 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 as 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 than using Eustis.
If you need help installing Code::Blocks on your own machine, see the course staff or the running C page.
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 `which_sort`), 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`.
You should unzip the contents of our hw5-tests.zip file into a directory you own, say `cop3223h/hw5testing`.
You will need to add tests for `f` (i.e., the file `test_f.c`) and the files `tap.c` and `tap.h` to the project (`test_f`) for each problem. 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`.

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`. That file must also be added to the project. If you are not prompted to add the file when you first create it, then 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`.
To run our tests, you can then build and run the project, which will run the main() function in the testing file (`test_f.c`). 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.
When everything is built, the easiest way to capture the test output is to use the command line. Run the `cmd.exe` program (on Windows, which you can start by typing “cmd” into Cortana and selecting `cmd.exe`) or the Terminal (on a Mac) 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 on Windows, then you would execute the following from the command line prompt:

```
bin/Debug/test_f.exe >test_f.txt
```

which will put the testing output into the file `test_f.txt`. On a Mac, you can accomplish the same thing by executing from the terminal prompt:

```
bin/Debug/test_f >test_f.txt
```

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

1. (10 points) [Programming] Write a function

   ```
   extern char * which_sort(int a, int b, int c);
   ```

   that takes 3 **`int`** arguments, `a`, `b`, and `c`, and returns a string describing their relationship. The result is `"all equal"` if `a`, `b`, and `c` are all equal; result is `"strictly increasing"` if `a` is strictly less than `b` and `b` is strictly less than `c`; result is `"increasing"` if `a` is no greater than `b` and `b` is no greater than `c`, but they are not strictly increasing; and result is `"strictly decreasing"` if `a` is strictly greater than `b` and `b` is strictly greater than `c`; and result is `"decreasing"` if `a` is not less than `b` and `b` is not less than `c`, but they are not strictly decreasing; and result is `"unrelated"` if none of the other relationships above hold.

   Tests are shown in Figure 1 on the following page.

   Remember to turn in your C source code file `which_sort.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_which_sort.c,v 1.2 2019/03/06 22:33:00 leavens Exp $
// compile with:
// gcc -g -Wall -pedantic tap.c which_sort.c test_which_sort.c -o test_which_sort

#include <stdlib.h>
#include <stdio.h>
#include "tap.h"
#include "test_which_sort.h"

// Testing for which_sort().
int main() {
    plan(13);
    is(which_sort(-3, -3, -3), "all equal");
    is(which_sort(-3, -3, -2), "increasing");
    is(which_sort(3, 4, 5), "strictly increasing");
    is(which_sort(3, 4, 4), "increasing");
    is(which_sort(100, 500, 200), "unrelated");
    is(which_sort(300, 200, 100), "strictly decreasing");
    is(which_sort(300, 300, 299), "decreasing");
    is(which_sort(300, 299, 299), "decreasing");
    is(which_sort(0, 0, 0), "all equal");
    is(which_sort(90, 80, 70), "strictly decreasing");
    is(which_sort(90, 80, 80), "decreasing");
    is(which_sort(72, 36, 95), "unrelated");
    is(which_sort(-2, -3, -4), "strictly decreasing");
    return exit_status();
}
```

Figure 1: Tests for the function `which_sort`. Note that **`is()`** is part of libtap (file `tap.h`) that compares two strings for equality. The **`plan`** and **`exit_status`** functions are also part of libtap.

2. (15 points) [Programming] In C, write a function,

   ```
   extern int below20int(const char word[]);
   ```

   that takes a null-terminated string, `word`, and returns the number that `word` represents, if `word` is the English name for a number between 0 and 19 (inclusive), which is all in lowercase letters; if `word` is does not represent such a name for a number, then it returns -1. See Figure 2 for tests. We also provide the file `below20int.h` in `hw5-tests.zip`.

```
// $Id: test_below20int.c,v 1.4 2019/03/06 23:56:56 leavens Exp $
// compile with:
// gcc -g -Wall -pedantic tap.c below20int.c test_below20int.c -o test_below20int

#include <stdlib.h>
#include <stdio.h>
#include "tap.h"
#include "below20int.h"

// Testing for below20int().
int main() {
    plan(22);
    ok(below20int("zero") == 0, "test for zero");
    ok(below20int("one") == 1, "test for one");
    ok(below20int("two") == 2, "test for two");
    ok(below20int("three") == 3, "test for three");
    ok(below20int("four") == 4, "test for four");
    ok(below20int("five") == 5, "test for five");
    ok(below20int("six") == 6, "test for six");
    ok(below20int("seven") == 7, "test for seven");
    ok(below20int("eight") == 8, "test for eight");
    ok(below20int("nine") == 9, "test for nine");
    ok(below20int("ten") == 10, "test for ten");
    ok(below20int("eleven") == 11, "test for eleven");
    ok(below20int("twelve") == 12, "test for twelve");
    ok(below20int("thirteen") == 13, "test for thirteen");
    ok(below20int("fourteen") == 14, "test for fourteen");
    ok(below20int("fifteen") == 15, "test for fifteen");
    ok(below20int("sixteen") == 16, "test for sixteen");
    ok(below20int("seventeen") == 17, "test for seventeen");
    ok(below20int("eighteen") == 18, "test for eighteen");
    ok(below20int("nineteen") == 19, "test for nineteen");
    ok(below20int("uno") == -1, "test for uno");
    ok(below20int("diez") == -1, "test for diez");
    return exit_status();
}
```

Figure 2: Tests for the function `below20int`.

Remember to turn in your C source code file `below20int.c` and the output of running our tests.

3. (20 points) [Programming] In C, write procedure eng2num that takes no arguments; it prompts on standard output (stdout) with "`number word? `", and then reads a word (followed by a newline) from standard input (stdin), which should be the English name for a natural number strictly less than 20. The procedure should output to stdout either the printed form of the number corresponding to that word, or the string "error: not a number less than 20" if the word is not the lower case name of a number less than 20 in English.

   The following are examples of 2 lines each (separated by a blank line), where the rest of the (first) line following the prompt "`number word? `" is typed by the user, and the second line is the procedure's response.

   ```
   number word? eighteen
   18

   number word? six
   6

   number word? duex
   error: not a number less than 20

   number word? 8
   error: not a number less than 20

   number word? seven
   7
   ```

   You must use `below20int` in your solution in an essential way.

   Tests are provided in the file `test_eng2num.c`, which uses the file `testing_io.c`, the header file `testing_io.h`, and files named `test_eng2num*.in` and `test_eng2num*.expected`, where * represents some digits. We also provide the header file `eng2num.h`. All these files are found in `hw5-tests.zip`.

   Remember to turn in your C source code file `eng2num.c` and the output of running our tests.

4. (20 points) [Programming] In C, write a function,

   ```
   extern double average(const double nums[], int sz);
   ```

   that takes an array of doubles, `nums`, and a positive integer `sz`, such that each integer i such that $0 \leq i <$ `sz` is a legal index into `nums` and `nums[`i`]` is not `NaN`, and returns the average of the numbers in `nums`. Note that `NaN` is a special double called "Not a Number," that results from numerical errors (such as dividing a number by 0); since `NaN` has various odd properties (such as any comparison involving it returns false, including `NaN == NaN`), we don't want you to have to consider it. Tests for `average` appear in Figure 3 on the following page, which includes some output that may be helpful for debugging. Note that `approx` is a function we provide (see Figure 4 on the next page and Figure 5 on page 7) in `hw5-tests.zip`; it returns true if the absolute value of the difference between its first two arguments is strictly less than the third argument. Another function we provide in `hw5-tests.zip` for testing is `init_array`, and the rule functions used in testing; see Figure 6 on page 7 and Figure 7 on page 8. We also provide the file `average.h` in `hw5-tests.zip`.

   Remember to turn in your C source code file `average.c` and the output of running our tests.

```
// $Id: test_average.c,v 1.3 2019/03/06 22:23:34 leavens Exp $
// compile with:
// gcc -g -Wall -pedantic tap.c approx.c init_array.c average.c test_average.c -o test_average
#include <stdio.h>
#include <stdlib.h>
#include "tap.h"
#include "approx.h"
#include "init_array.h"
#include "average.h"

// Testing for average()
int main() {
    double data[10];
    double small = 0.00001;
    data[0] = 0.0;
    data[1] = 1.0;
    ok(approx(average(data, 2), 0.5, small), "average of 0.0 and 1.0");

    // the same test as above, but written using init_array and the rule castToDouble
    init_array(data, castToDouble, 2);
    ok(approx(average(data, 2), 0.5, small), "average of 0.0 and 1.0 with rule");

    init_array(data, square, 7);
    // the following shows some code that can be useful for debugging, if you uncomment the printf
    double temp = average(data, 7);
    printf("average of squares is %lf\n", temp);
    ok(approx(temp, 13.0, small), "average of the squares from 0 to 6 (inclusive)");
    init_array(data, plus10, 10);
    ok(approx(average(data, 10), 14.5, small), "average of i+10 for i from 0 to 9 (inclusive)");
    init_array(data, fifty, 9);
    ok(approx(average(data, 9), 50.0, small), "average of 50.0 entries");
    init_array(data, eToPower, 10);
    ok(approx(average(data, 10), 1281.830805, small), "average of e**i for i from 0 to 9 (inclusive)");
    init_array(data, logePlus10, 10);
    temp = average(data, 10);
    printf("average of logePlus10 is %lf\n", temp);
    ok(approx(temp, 2.653806, small), "average of logePlus10(i) for i from 0 to 9 (inclusive)");
    return exit_status();
}
```

Figure 3: Tests for the function average.

```
// $Id: approx.h,v 1.1 2019/03/06 21:07:13 leavens Exp leavens $

#include <stdbool.h>

// Requires: neither argument is NaN
// Ensures: result is true just when the absolute value of actual - expected is less than epsilon.
extern bool approx(double actual, double expected, double epsilon);
```

Figure 4: The header file `approx.h` from `hw5-tests.zip`.

```
// $Id: approx.c,v 1.1 2019/03/06 21:07:13 leavens Exp leavens $

#include <stdbool.h>
// math.h declares the fabs function
#include <math.h>
#include "approx.h"

// Requires: neither argument is NaN
// Ensures: result is true just when the absolute value of actual - expected is less than epsilon.
bool approx(double actual, double expected, double epsilon) {
    return fabs(actual - expected) < epsilon;
}
```

Figure 5: The implementation file `approx.c` from `hw5-tests.zip`.

```
// $Id: init_array.h,v 1.1 2019/03/06 22:23:34 leavens Exp leavens $

// The following is the type of functions mapping ints to doubles,
// which is used to create test data. Think of such a function
// as describing (by a rule) the ith element of an array of doubles.
typedef double (*RuleFun)(int i);

// Requires: for all 0 <= i and i < sz :: i is a legal index into data
// and rule is defined on i and rule has no effects when applied to i.
// Effect: for all 0 <= i and i < sz :: data[i] == rule(i)
extern void init_array(double data[], RuleFun rule, int sz);

// Rule function examples that are in init_array.c are declared below.

// Ensures: result is (double) i
extern double castToDouble(int i);
// Ensures: result is (double) i*i
extern double square(int i);
// Ensures: result is i+10.0
extern double plus10(int i);
// Ensures: result is 50.0
extern double fifty(int i);
// Ensures: result is e**i;
extern double eToPower(int i);
// Ensures: result is log(i+10.0);
extern double logePlus10(int i);
```

Figure 6: The header file `init_array.h` from `hw5-tests.zip`.

```c
// $Id: init_array.c,v 1.1 2019/03/06 22:23:34 leavens Exp leavens $

#include <math.h>
#include "init_array.h"

// Requires: for all 0 <= i and i < sz :: i is a legal index into data
// and rule is defined on i and rule has no effects when applied to i.
// Effect: for all 0 <= i and i < sz :: data[i] == rule(i)
void init_array(double data[], RuleFun rule, int sz) {
    for (int i = 0; i < sz; i++) {
        data[i] = rule(i);
    }
}

// Some rule functions follow.
double castToDouble(int i) { return (double) i; }
double square(int i) { return (double) i*i; }
double plus10(int i) { return i+10.0; }
double fifty(int i) { return 50.0; }
double eToPower(int i) { return exp(i); }
double logePlus10(int i) { return log(i+10.0); }
```

Figure 7: The implementation file `init_array.c` from `hw5-tests.zip`.

5. (30 points) [Programming] In C, write a function,

   ```
   extern int below100int(const char *words[], int sz);
   ```

   that takes an array of null-terminated strings, `words`, that together form the English name for a number between 0 and 99 (inclusive), all in lowercase letters, and an int `sz` (between 1 and 2, inclusive), which gives the number of strings in `words` and returns the number that the words in `words` (put together with a space between them if `sz` is 2) represents. See Figure 8 and Figure 9 on the following page for tests. We also provide the file `below100int.h` in `hw5-tests.zip`.

   Your code should use `below20int` as a helping function.

```
// $Id: test_below100int.c,v 1.3 2019/03/06 22:47:44 leavens Exp leavens $
// compile with:
// gcc -g -Wall -pedantic tap.c below20int.c below100int.c test_below100int.c -o test_below100int

#include <stdlib.h>
#include "tap.h"
#include "below100int.h"

// Testing for below100int().
int main() {
    plan(26);
    const char *words[2];
    words[0] = "zero";
    words[1] = NULL;
    ok(below100int(words, 1) == 0, "test for zero");
    words[0] = "one";
    ok(below100int(words, 1) == 1, "test for one");
    words[0] = "two";
    ok(below100int(words, 1) == 2, "test for two");
    words[0] = "seven";
    ok(below100int(words, 1) == 7, "test for seven");
    words[0] = "nine";
    ok(below100int(words, 1) == 9, "test for nine");
    words[0] = "ten";
    ok(below100int(words, 1) == 10, "test for ten");
    words[0] = "eleven";
    ok(below100int(words, 1) == 11, "test for eleven");
    words[0] = "twelve";
    ok(below100int(words, 1) == 12, "test for twelve");
    words[0] = "thirteen";
    ok(below100int(words, 1) == 13, "test for thirteen");
    words[0] = "fourteen";
    ok(below100int(words, 1) == 14, "test for fourteen");
```

Figure 8: First part of the tests for the function `below100int`.

Remember to turn in your C source code file `below100int.c` and the output of running our tests.

Points

This homework's total points: 95.

```
    words[0] = "fifteen";
    ok(below100int(words, 1) == 15, "test for fifteen");
    words[0] = "nineteen";
    ok(below100int(words, 1) == 19, "test for nineteen");
    words[0] = "twenty";
    ok(below100int(words, 1) == 20, "test for twenty");
    words[1] = "one";
    ok(below100int(words, 2) == 21, "test for twenty one");
    words[1] = "three";
    ok(below100int(words, 2) == 23, "test for %s %s", words[0], words[1]);
    words[0] = "thirty";
    words[1] = "three";
    ok(below100int(words, 2) == 33, "test for %s %s", words[0], words[1]);
    words[0] = "fourty";
    words[1] = "nine";
    ok(below100int(words, 2) == 49, "test for %s %s", words[0], words[1]);
    words[0] = "sixty";
    words[1] = "five";
    ok(below100int(words, 2) == 65, "test for %s %s", words[0], words[1]);
    words[0] = "sixty";
    words[1] = "seven";
    ok(below100int(words, 2) == 67, "test for %s %s", words[0], words[1]);
    words[0] = "fifty";
    words[1] = "two";
    ok(below100int(words, 2) == 52, "test for %s %s", words[0], words[1]);
    words[0] = "seventy";
    words[1] = "six";
    ok(below100int(words, 2) == 76, "test for %s %s", words[0], words[1]);
    words[0] = "seventy";
    words[1] = "three";
    ok(below100int(words, 2) == 73, "test for %s %s", words[0], words[1]);
    words[0] = "eighty";
    words[1] = "eight";
    ok(below100int(words, 2) == 88, "test for %s %s", words[0], words[1]);
    words[0] = "ninety";
    words[1] = "nine";
    ok(below100int(words, 2) == 99, "test for %s %s", words[0], words[1]);
    words[0] = "ninety";
    words[1] = NULL;
    ok(below100int(words, 1) == 90, "test for %s", words[0]);
    words[0] = "fourty";
    ok(below100int(words, 1) == 40, "test for %s", words[0]);
    return exit_status();
}
```

Figure 9: Second part of the tests for the function `below100int`.