# 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 the function you should be writing (such as `fl_temp_judge`), 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.
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 `fl_temp_judge`), 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 the project tests for `f` (i.e., the file `test_f.c`) and the files `tap.c` and `tap.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`.

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`.
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.
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 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, you can accomplish the same thing by executing from the terminal prompt:

```
./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] In sunny Florida, long term residents develop a semi-tropical sense of what temperatures mean. Your task in this problem is to write a function, in C, that captures that judgment.[1] That is, you are to write a function

   ```
   char * fl_temp_judge(double temp);
   ```

   that takes a temperature, `temp`, which is a **`double`** representing the temperature in degrees Farenheit, and produces a C string (which has type **`char *`**, because a character array used as a return value in C really returns the address of the array). This function should return the string `"cold"` if `temp` is less than 55.0, it should return `"cool"` if `temp` is at least 55.0 and less than 70.0, `"warm"` if `temp` is at least 70.0 and less than 85.0, and `"hot"` if `temp` is 85.0 or more.[2] Tests are shown in Figure 1 on the following page.

   Remember to turn in your C source code file `fl_temp_judge.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,

   ```
   int divisors(int n, int divs[], int sz);
   ```

   that takes a positive integer `n`, an array of ints, `divs`, and in int `sz`, such that `divs` has at least `sz` elements, where `n` is at least as large as the number of divisors of `n`. The result of a call

[1] The temperature ranges in this problem were given to Dr. Leavens by his wife.

[2] So that you don't have to worry about the strange properties of the double called "not a number" (NaN), you should assume that `temp` is such that `isnan(temp)` is false. The C math library `math.h` defines `isnan`.

```c
// $Id: test_fl_temp_judge.c,v 1.2 2017/03/06 04:14:06 leavens Exp leavens $
// compile with:
// gcc tap.c fl_temp_judge.c test_fl_temp_judge.c -o test_fl_temp_judge

#include <stdlib.h>
#include <stdio.h>
#include "tap.h"
extern
char * fl_temp_judge(double temp);

// Testing for fl_temp_judge().
int main() {
    plan(19);
    is(fl_temp_judge(30.0), "cold");
    is(fl_temp_judge(54.0), "cold");
    is(fl_temp_judge(54.99), "cold");
    is(fl_temp_judge(55.0), "cool");
    is(fl_temp_judge(56.0), "cool");
    is(fl_temp_judge(60.1259), "cool");
    is(fl_temp_judge(69.999), "cool");
    is(fl_temp_judge(70.0), "warm");
    is(fl_temp_judge(70.125), "warm");
    is(fl_temp_judge(70.125), "warm");
    is(fl_temp_judge(72.0), "warm");
    is(fl_temp_judge(79.0), "warm");
    is(fl_temp_judge(84.5), "warm");
    is(fl_temp_judge(84.999), "warm");
    is(fl_temp_judge(85.0), "hot");
    is(fl_temp_judge(85.01), "hot");
    is(fl_temp_judge(95.0), "hot");
    is(fl_temp_judge(125.0), "hot");
    is(fl_temp_judge(-20.0), "cold");
    return exit_status();
}
```

Figure 1: Tests for the function `fl_temp_judge`. 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.

`divisors(n, divs, sz)` is the number of (positive) divisors of `n` such that for each $i$ such that $0 \leq i$ and $i$ is strictly less than the result, `divs`[$i$] is positive, and `divs`[$i$] evenly divides `n`. Furthermore, `divs` is sorted in strictly increasing order (for the elements up to the result). (For convenience, this specification for `divisors` is stated in the file `divisors.h`.) Tests for `divisors` appear in Figure 2 on page 5, which includes some output that may be helpful for debugging.

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

3. (20 points) [Programming] The number 6 is considered a "perfect number" because it is the sum of its proper divisors. A *proper divisor* is a positive divisor that is strictly less than the number itself; for example, the proper divisors of 6 are 1, 2, and 3, which sum to 6. 28 is also a perfect number.

   Your task in this problem is to write a function

   ```c
   bool is_perfect(int n);
   ```

   that takes a positive int, `n`, and returns true just when `n` is a perfect number (i.e, when the sum of its proper divisors is `n`). Tests are shown in Figure 3 on page 6.

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

## Points

This homework's total points: 50.

```c
// $Id: test_divisors.c,v 1.2 2017/03/06 04:22:01 leavens Exp leavens $
// compile with:
// gcc tap.c divisors.c test_divisors.c -o test_divisors

#include <stdlib.h>
#include <stdio.h>
#include "tap.h"
#include "divisors.h"
#define SIZE 150

// Requires: 0 < n and divs has at least count elements
// Effect: print to stdout the count divisors of n contained in divs
void print_divisors(int n, int divs[], int count) {
    printf("divisors of %d are: ", n);
    for (int i = 0; i < count; i++) {
        printf("%d", divs[i]);
        if (i < count-1) {
            printf(", ");
        }
    }
    printf("\n");
}

/* Check the divsors of n computed by divisors()
   and return the number of divisors of n. */
int check_divisors(int n) {
    int divs[SIZE];
    int count = divisors(n, divs, SIZE);
    print_divisors(n, divs, count);
    // 1 and n must both divide n
    ok(0 < count && divs[0] == 1 && divs[count-1] == n);
    // each element in divs must be a divisor
    for (int i = 0; i < count; i++) {
        ok(0 < divs[i] && divs[i] <= n && n % divs[i] == 0);
    }
    return count;
}

// Testing for divisors().
int main() {
    ok(check_divisors(1) == 1);
    ok(check_divisors(2) == 2);
    ok(check_divisors(3) == 2);
    ok(check_divisors(4) == 3);
    ok(check_divisors(6) == 4);
    ok(check_divisors(7) == 2);
    check_divisors(12);
    check_divisors(28);
    check_divisors(42);
    check_divisors(50);
    check_divisors(60);
    check_divisors(72);
    check_divisors(100);
    check_divisors(128);
    check_divisors(360);
    check_divisors(496);
    check_divisors(500);
    check_divisors(840);
    return exit_status();
}
```

Figure 2: Tests for the function `divisors`.

```c
// $Id: test_is_perfect.c,v 1.2 2017/03/06 04:16:20 leavens Exp leavens $
// compile with:
// gcc tap.c divisors.c is_perfect.c test_is_perfect.c -o test_is_perfect

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

// Testing for is_perfect()
int main() {
    ok(!is_perfect(4));
    ok(is_perfect(6));
    ok(!is_perfect(12));
    ok(!is_perfect(27));
    ok(is_perfect(28));
    ok(!is_perfect(29));
    ok(is_perfect(496));
    ok(is_perfect(8128));
    return exit_status();
}
```

Figure 3: Tests for the function `is_perfect`.