CS 211 Homework 1

Winter 2022

Code Due:	January 13, 2022, 11:59 PM, Central Time
Self-Eval Due:	January 16, 2022, 11:59 PM, Central Time
Partners:	No; must be completed by yourself

Purpose

The goal of this assignment is to get you programming in C, including simple I/O, separate compilation, and testing. Note that this assignment is done individually, not with a partner.

Preliminaries

Login to the server of your choice and *cd* to the directory where you keep your CS 211 work. Then unarchive the starter code, and change into the project directory:

```
% cd cs211
% tar -kxvf ~cs211/hw/hw01.tgz
⋮
% cd hw01
```

You can check that you have correctly downloaded and configured everything by building and running the tests:

```
% make
cc -c -o src/overlapped.o src/overlapped.c  -g -O1 -std=...
cc -c -o src/circle.o src/circle.c  -g -O1 -std=c11 -ped...
cc -o overlapped test/overlapped.o src/circle.o -lm -fsa...
cc -c -o test/test_circle.o test/test_circle.c  -g -O1 -...
cc -o test_circle test/test_circle.o src/circle.o -lm -f...
./test_circle

All 3 checks passed.
%
```

The build and tests should complete successfully. This doesn't mean that the code is correct, but rather that the tests are inadequate.

This homework assignment must be completed on Linux by logging into a Linux workstation. Each time you login to work on CS 211, you should run *211* to ensure your environment is setup correctly. (If you get an error saying that 211.h doesn't exist, that probably means you missed the step in Lab 1 where you needed to run ~cs211/setup211.)

Contents

Orientation

In this project, you will write:

- a tiny computational geometry library (src/circle.h and src/circle.c),

- a tiny client program that uses it (src/overlapped.c), and
- some tests for the library (test/test_circle.c).

Type definitions and function signatures for the library are provided for you in src/circle.h; since the grading tests expect to interface with your code via this header file, **you must not modify src/circle.h in any way.** All of your code will be written in the three .c files.

Make *targets*

The project also provides a Makefile with several targets:

target	**description**
test	builds everything & runs the tests * &
all	builds everything, runs nothing &
test_circle	builds the unit tests
overlapped	builds the *overlapped* program
clean	removes all build products &

* default & phony

Target `test` is the default, which means you can run it by typing `make` alone, with no target name.

Specifications

The project comprises two functional components, which are specified in the next two subsections.

The circle *library*

The *circle* library defines one `struct` type and three functions, as follows:

- The `circle` structure type represents a circle positioned on a Euclidean plane in terms its center (x and y coordinates) and its radius.
- Function `valid_circle(struct circle c)` returns a `bool` indicating whether circle `c` is *valid*. A circle is valid if and only if its radius is positive.
- Function `read_circle(void)` parses a `struct circle` from the standard input and returns it. It should expect the values of the three fields in order: `x`, `y`, `radius`.

  Exceptional cases: The returned circle must be fully initialized even if *scanf*() fails due to bad or end of input. If the input ends or is malformed, *read_circle*() returns a circle with center $(0.0, 0.0)$ and radius -1.0.

- Function `overlapped_circles(struct circle, struct circle)` returns a `bool` indicating whether the two given circles overlap. Circles are considered to overlap only if they contain some area in common, not if they are merely tangent to each other.

The overlapped *client program*

The *overlapped* client program reads a first (“target”) circle. If there is an error in reading the target circle, the program terminates with an exit code of 1 to indicate an error.

Then the program reads as many subsequent (“candidate”) circles as are provided by the user; for each valid circle read after the target circle, it prints `"overlapped\n"` if the candidate circle overlaps the target, or `"not overlapped\n"` if not. If the program reads an invalid candidate circle, then it terminates with an exit code of 0 to indicate success, printing nothing.

The program does not print anything else.

It’s a bug if your output differs from the specification.

Here are two examples of running overlapped:

```
% ./overlapped
0 0 5
0 2 1
overlapped
0 10 1
not overlapped
2020 211 -1
%
```

```
% ./overlapped
1 0 1
0 1 0.4
not overlapped
0 1 0.41
not overlapped
0 1 0.414
not overlapped
0 1 0.415
overlapped
1 -1 0.415
overlapped
-2020 -211 -2
%
```

Reading documentation effectively can depend on understanding typesetting conventions. In the transcripts on the left, the **bold** text is what the user types, and the `highlighted` text is what the computer responds with. Your actual prompt will probably differ from %, which is a convention for printing Unix shell prompts in documentation.

Reference

`CHECK()` *forms for unit testing*

Unlike many newer programming languages, C does not provide any built-in testing mechanism. Instead, we test C code using a library, often written in C itself.

In CS 211 we will use a library called *lib211*, which includes a basic testing framework. To access *lib211*’s definitions, you need to `#include <211.h>` from whichever files you want to use them in, so we have written that line in test/test_circle.c for you already.

The C standard library provides a *macro* `assert(`⟨*expression*⟩`)`, which aborts your program if ⟨*expression*⟩ is false. Assertions are not intended for testing, but as a fail-safe mechanism for stopping your program when a bug is detected.

The *lib211* library provides several forms that do various kinds of checks, but in this homework, we need only one: the *CHECK*(3) macro. *CHECK*() takes one argument, which it evaluates to a `bool`. If the resulting value is is `true` then the check passes *silently,* but if it is `false` then it *CHECK*() prints a message showing you the line number of the failed check.

For details on *CHECK*(3) and related forms, see `man CHECK`.

For example, here is a test case with one passing and two failing checks:

```
void test_less_than(void)
{
    CHECK( 2 < 3 );    // passes silently
    CHECK( 3 < 3 );    // fails noisily
    CHECK( 4 < 3 );    // fails noisily also
}
```

When all tests have finished, *lib211*'s testing framework prints information about the total number of successful and unsuccessful checks.

Hints

Definition of overlap for circles

Two circles overlap if the distance between their centers is less than the sum of their radii.

You don't need *sqrt*() here because this statement is equivalent: Two circles overlap if the square of the sum of their radii exceeds the square of the distance between their centers. You can still use *sqrt*() if you really want though.

Definitely don't use *pow*() for squaring numbers though. Just multiply a number by itself to square it as that's both easier to write and much faster at runtime.

Strategy for the `read_circle` function

First define a **`struct`** `circle` variable, without initializer, to hold the function's result. Then, try to initialize its three fields using the *scanf*(3) function. If *scanf*() is unable to convert all three `double`s as indicated by its result value, then initialize the **`struct`** `circle` to the invalid state `{0.0, 0.0, -1.0}` instead (per the specification above). Then, whether or not the input succeeded, return the **`struct`** `circle`.

Algorithm for the overlapped *program*

Here is an algorithm you can use in src/overlapped.c:

1. Define a `struct circle` variable to hold the target circle, and initialize it to the result of calling *read_circle*().
2. If the target circle is invalid according to *valid_circle*(), exit with an error code of 1.
3. Repeat indefinitely:
    (a) Define a `struct circle` variable to hold the candidate circle, and initialize it to the result of calling *read_circle*().
    (b) If the candidate circle is invalid according to *valid_circle*(), exit with an error code of 0.
    (c) Use *overlapped_circles*() in the condition of an `if–else` statement to check whether the target circle overlaps the candidate circle and print the correct message in either case.

    To get an infinite loop that repeats some statements, use a `while` loop with a condition of *true*:

```
while (true) {
    // Statements to repeat go here.
}
```

From `main`, exiting can be accomplished by `return`ing the desired error code, but to exit directly from any other function one must call the *exit*(3) function.

(Note that the "3" in *exit*(3) is not the argument you should pass, but the section of the Unix manual system where documentation for the *exit* function is found. To see why this matters, compare the result of running `man exit` with the result of running `man 3 exit`.)

Deliverables & evaluation

For this homework you must:

1. Implement the specification for the *circle* library from the previous section in src/circle.c.
2. Implement the specification for the *overlapped* client program from the previous section in src/overlapped.c.
3. Add more test cases for the `overlapped_circles` function provided by the *circle* library in test/test_circle.c.

    In particular, file test/test_circle.c already contains two test cases, `test_tangent` and `test_not_overlapped`, both of which are called from `main`. Your job is to add two more test cases, demonstrating that:

    - `overlapped_circles` returns `true` given different but overlapping circles, and
    - `overlapped_circles` returns `true` given the same circle for both arguments.

Grading will be based on:

- the correctness of your implementations with respect to the specifications,
- the presence of the two required test cases, and
- adherence to the CS 211 Style Manual.

In particular, pay careful attention to case and spacing, and note that extra output beyond what is specified is a bug, not a feature.

Submission

Homework submission and grading will use Gradescope. You must include any files that you create or change. For this homework, that will include src/circle.c, src/overlapped.c, and test/test_circle.c. (You must not modify Makefile or src/circle.h.)

You can submit as many times as you want. Check the Gradescope website to see the results of tests run on your code.

Per the syllabus, if you engaged in arms-length collaboration on this assignment, you must cite your sources. You may write citations either in comments on the relevant code, or in a file named README.txt that you submit along with your code. See the syllabus for definitions and other details.

Submit using the command-line tool `submit211`. You can run the command with the `--help` flag to see more details. The tool will ask you to log in with your Gradescope credentials, so make sure you've created an account!

To submit the necessary files for this homework, you will run something that looks like:

```
% submit211 submit --hw hw01 src/circle.c src/overlapped.c test/test_circle.c
```

Remember that those are relative paths to the files you want to submit. So make sure to change them to make sense for whatever directory you are running the command from. You can also add any additional files you want to upload, like README.txt, to the end of the command.