ECE 209
Fall 2023

Program 2: `map`

Due Tuesday, Dec 5 @ 11:45pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of 0 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to write this program on your own. **You may use GitHub Copilot** to assist with code development, but you are responsible for the code that you submit. Evidence of copying code from any source will be investigated as a potential academic integrity violation.

This program will implement a simplified map program, with streets that are restricted to a square grid: either west-to-east or north-to-south. Map info is read from a file. Your job is to maintain the collection of streets and to answer queries about the map.

The learning objectives of the program are:

- Create and manipulate linked lists.
- Create a multi-file C program.
- Implement an abstract data type.

Problem Description

This assignment is an extremely simplified version of a mapping program, such as Google Maps. The map area is described by a two-dimensional grid, with each point specified by an integer pair: (east, north). The coordinate (0,0) is the bottom left corner of the map, and the unit of distance is a "block." The first coordinate (east) is the number of blocks to the right of the left edge. The second coordinate (north) is the number of blocks above the bottom edge. (Basically, "east" is the x-coordinate and "north" is the y-coordinate.)

For this assignment, there is no restriction on the size of the map area. Each point coordinate must be a positive integer, but there is no maximum limit.

A street is a straight line, which must be either horizontal (west-to-east) or vertical (north-to-south). It is specified by a name (string), the starting point, and the ending point. For a horizontal street, the starting point is to the left -- the western-most point. For a vertical street, the starting point is the top one -- the northern-most point.

A map is a collection of streets. Each map has a name (up to 15 characters), and the street data for a map named "foo" is in a text file named "foo.txt". The text file contains a line of information for each street, consisting of the name (up to 15 characters), the starting point, and the ending point.

Figure 1 shows an example map. Starting points are shown as open circles. Dark circles show points that are intersections of two streets.

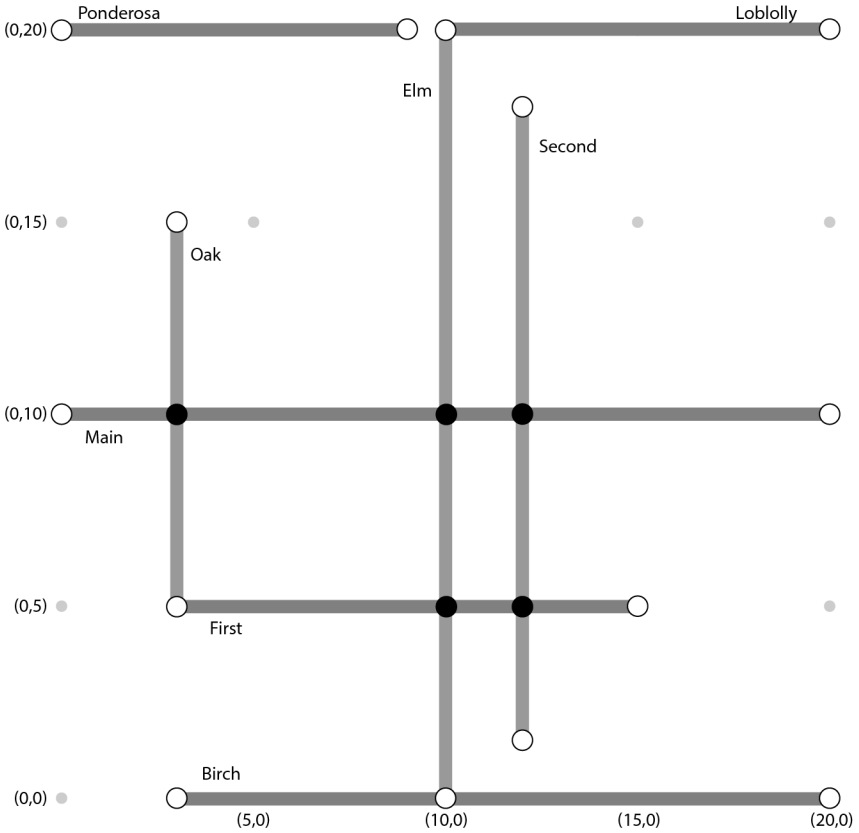

Figure 1. An example map, corresponding to the data in **map1.txt**.

Your job is to implement several functions that create and manipulate data structures that represent streets, points, and maps. In particular, you will need to create a linked list of streets. The programming interface is described in the next section.

Program Specification

For this program, we have three source files and two header files. I will provide **main.c**, **Street.h**, and **Map.h**. You will create **Street.c** and **Map.c**. See the Appendix for information about how to use these files in CLion.

Details: Street

The **Street.h** header file contains definitions of several struct types. The first defines a point as a value with two integer components:

```
struct point {
    int east;   /* x-coordinate of map point */
    int north;  /* y-coordinate of map point */
};
typedef struct point Point;
```

Another struct describes a street. It has three components: a name (string), starting point (struct point) and ending point (struct point).

```
struct street {
    char name[16];  /* street name is up to 15 characters */
    struct point start;
    struct point end;
};
typedef struct street Street;
```

Note that we have also used typedef to give the name Point to a struct point and the name Street to a struct street. You can use either in your code -- they are interchangeable.

By putting these struct definitions in a header file, we make them available to any source file that might want to write code using these structs. Each source file that wants to do so can use #include to bring in these definitions without making a copy of the struct definition.

We also include declarations for three functions related to streets. These functions will be defined in **Street.c**, but we put the declarations in a header file so that other source files can call these functions. (Remember, once the compiler sees a function declaration, it can generate code for the required function call, even without seeing the complete function definition.)

```
Street * newStreet(const char *name, const Point * start,
                   const Point *end);
```

This function is used to create a new instance of a street. We give the information needed (name, start, end), and the function will allocate a new struct (using **malloc**), fill in the components, and return a pointer to the new struct. See the **Street.h** file comments for more details.

```
int streetLength(const Street *s);
int streetDirection(const Street *s);
```

These two functions return useful information about a street. Given a pointer to a struct street, they return the length of the street or the direction. See the **street.h** file for more details.

Details: StreetList and StreetNode

Also in **Street.h**, we define structs used to create and manage a list of streets. The first is used as the node of the list. The data field is a pointer to a Street, not a struct value. This allows us to create multiple lists that refer to the same street data without copying the streets.

```
struct streetNode {
   Street * street;  /* data -- pointer to street data */
   struct streetNode * next;  /* pointer to next item in list */
};
typedef struct streetNode StreetNode;
```

Another struct is defined to hold the head and tail pointers for a list.

```
struct streetList {
   StreetNode * head;
   StreetNode * tail;
};
typedef struct streetList StreetList;
```

We do not declare any list-related functions here. You are encouraged to declare and implement these in Map.c, in order to implement the functions required for the Map data structure.

Details: Map

The second header file, **Map.h**, provides types and function declarations that define actions on a map. First, note that this header has `#include "Street.h"`. This is because any source file that wants to use the Map type will also use the Street and Point types. (The `#ifndef` and `#define` statements in the header files will prevent a header from from being included twice by the the same source code file.)

Next, the **Map** type is defined using typedef. Note that it is a struct pointer, but there's no definition of struct map! This is intentional. Map is what's known as an *Abstract Data Type (ADT)*. Other source files can use it, but none of them will know the details of the components of the struct. The struct will be defined in **Map.c**, which is the only source file that can access the individual components (fields) of the struct.

You are responsible for defining struct map in your **Map.c** file. You have complete freedom in how you want to represent the required data. Do you want a single list of all streets? Separate lists of west-east and north-south streets? Do you want to use arrays instead of lists? (You won't know how many streets you'll have when the map is created, so you'd have to figure out how to deal with this.) There are many options. As long as the map functions have the correct behavior, you are free to implement this any way you like.

Finally are the function declarations for map-related functions. See Map.h for details.

```
Map createMap(const char *name);
const char *mapName(Map m);
int mapAddStreet(Map m, Street * s);
StreetList * mapAllStreets(Map m);
StreetList * mapNSStreets(Map m);
SteetList * mapWEStreets(Map m);
Street * mapFindStreet(Map m, const char *name);
StreetList * mapCrossStreets(Map m, const Street *s);
```

Several of these functions will return a linked list of streets. Note that a *new list* must be created each time. (Both the list struct and all of the nodes must be newly allocated.) The caller (who gets the list) is allowed to delete the list nodes, so you ***must not*** simply copy the head of a list that is already part of your map.

Note that there are ordering requirements for the lists that are returned. See **Map.h** for details.

Details: main

I am providing the **main.c** file for you, but this section describes what the **main** function does. It provides a user interface for opening a map file and performing query operations on the map. These queries will call all of the functions that you are required to create, so it's a reasonable way to test your code. (Make sure to create some new map files, and don't just rely on the one that I provide.)

As with Program 3, you can specify a file name as an argument to the program. If you do, it will read the map file when the program begins. If not, you will have to use the **m** command (described below) to read a map file. Either way is fine.

After the file (if specified) is read, the program will print the following list of commands.

```
m = read map file
a = print all streets
p = print named street
c = print cross streets
```

```
h = help
q = quit
```

Most of these commands take arguments. You specify the one-letter command and any required arguments, preferably on the same line. See the Appendix for an example using each of these commands. If you use the m command to open a new map file, the old map file is deleted and the new map is used for subsequent commands.

Suggested Development Plan

To implement this program, I suggest the following plan.

1. Create **Street.c** and **Map.c** with dummy implementations of all functions. Just return something, so that the entire program can be compiled. For functions that return a list, return NULL.
2. Implement the street functions in **Street.c**. You can write a separate main file that tests these functions, but they are simple enough that you can probably just implement them first and test them by running the map functions.
3. In **Map.c**, define your struct map. Think about how you want to represent the collection of streets. You can change this later, but it's useful to give this some thought before you jump right in. (Design before coding!)
4. Implement **createMap**, **mapName**, and **mapAddStreet**. This, along with the street functions, will be enough to read in a map file. Now you can use the provided main function, either by specifying a file name for the program, or by using the **m** command.
5. The **a** command prints all streets, and also prints separate lists of west-east and north-south streets. Implement **mapAllStreets** first, and test using the **a** command. Then implement **mapNSStreets** and **mapWEStreets**, and test using the **a** command.
6. Implement **mapFindStreet** and test using the **p** command.
7. Implement **mapFindCrossing** and test using the **c** command.

Hints and Suggestions

- Don't overcomplicate the program.
- Work incrementally. See the suggested development plan.
- You must include **stdio.h** for **printf/scanf** and **stdlib.h** for **malloc**. You must include **string.h** to use standard string functions.
- Your **Street.c** file must include **Street.h**. Don't copy **Street.h** into your source file. Just use `#include`. (If you think it's not working, ask for help. It's important that you learn how this works, so don't try to hack it.)
- Your **Map.c** file must include **Map.h**. You don't have to include **Street.h**, since it is already included by **Map.h**, but it won't hurt if you include both. Don't copy **Street.h** or **Map.h** into your source file. And especially don't include **Street.c**. If you think you need this to get things to compile, then you don't understand something and you should ask for help.

- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger to step through the program if the program behavior is not correct. If you are using CLion on your own computer, the debugger is integrated with the editor and compiler, so there's no excuse for not using it.
- For general questions or clarifications, use Piazza, so that other students can see your question and the answer. For code-specific questions, post a private question and attach your code as separate files. We will not debug your code by staring at it; make it easier for us to download and run your code.

Administrative Info

Updates or clarifications on Piazza:

Any corrections or clarifications to this program spec will be posted on Piazza. It is important that you read these postings, so that your program will match the updated specification.

What to turn in:

- Source files -- this time there are two files: **Street.c** and **Map.c**. You must submit both, and they must have the correct names. Submit to ZyBook 15.2 for automatic testing and grading.

Grading criteria:

10 points: Compiles with *no warnings and no errors*. Pay attention to any warnings that are shown when you submit to ZyBook. They may be different than warnings that you see in CLion. You will not get these points by simply submitting the template files. You must make a reasonable attempt to complete the assignment.

10 points: Proper coding style, comments, and headers. No unnecessary global variables. No goto. (See the Programming Assignments section on Moodle for more information.) For this assignment, there is no reason to use any global variable.

20 points: **Street.c** functions
- 8 points -- newStreet
- 6 points -- streetLength
- 6 points -- streetDirection

60 points: **Map.c** functions
- 5 points -- createMap
- 5 points -- mapName
- 5 points -- mapAddStreet
- 15 points -- mapAllStreets
- 5 points -- mapNSStreets
- 5 points -- mapWEStreets
- 10 points -- mapFindStreet
- 10 points -- mapCrossStreets

Appendix: Setting up the CLion Project

1. Create a new project. I suggest you call it "map" and use "map" as the directory name, but that's really up to you. The source code files have to have the proper names, but I couldn't care less about your local project name. As for all of our other projects, it's a C Executable using the C17 standard.
2. Edit the CMakeLists.txt file. Add files to the executable dependency list. These files don't exist yet, and CLion will complain, but we're going to fix that.

```
add_executable(map main.c Street.c Map.c Street.h Map.h)
```

3. Download a ZIP file from the ZyBook assignment, and copy all of the template files into the project.
4. Edit Configuration to change the Working Directory and download the example map file (**map1.txt**) into the project directory. You can also set the program argument to **map1.txt** to have this file opened each time you run.

Appendix: Example Run

Here are the sorted outputs for each of the test files. Output is shown in multiple columns to save space, but of course it will not be printed this way on your console. This run has **map1.txt** as a program argument.

```
Commands:
m = read map file
a = print all streets
p = print named street
c = print cross streets
h = help
q = quit

MAP: map1

Command: a
All streets in map1
Birch: (3,0)->(20,0) [WE, 17 blocks]
Elm: (10,20)->(10,0) [NS, 20 blocks]
First: (3,5)->(15,5) [WE, 12 blocks]
Loblolly: (10,20)->(20,20) [WE, 10 blocks]
Main: (0,10)->(20,10) [WE, 20 blocks]
Oak: (3,15)->(3,5) [NS, 10 blocks]
Ponderosa: (0,20)->(9,20) [WE, 9 blocks]
Second: (12,18)->(12,2) [NS, 16 blocks]

West-to-east streets:
Birch: (3,0)->(20,0) [WE, 17 blocks]
First: (3,5)->(15,5) [WE, 12 blocks]
Loblolly: (10,20)->(20,20) [WE, 10 blocks]
Main: (0,10)->(20,10) [WE, 20 blocks]
Ponderosa: (0,20)->(9,20) [WE, 9 blocks]

North-to-south streets:
Elm: (10,20)->(10,0) [NS, 20 blocks]
Oak: (3,15)->(3,5) [NS, 10 blocks]
```

```
Second: (12,18)->(12,2) [NS, 16 blocks]

Command: p Main
Main: (0,10)->(20,10) [WE, 20 blocks]

Command: c Main
Streets that intersect Main:
Oak: (3,15)->(3,5) [NS, 10 blocks]
Elm: (10,20)->(10,0) [NS, 20 blocks]
Second: (12,18)->(12,2) [NS, 16 blocks]

Command: q
```