ECE 209
Fall 2020

Program 4: `cellphone`

Due Friday, November 13 @ 11:59pm

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 zero 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 design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program will model a cellular phone network. The system consists of a number of *towers* (base stations) and a number of *phones*. A phone communicates with a single tower -- the tower that is closest to the phone, within its transmission range. A tower can communicate with any phone that is associated with that tower, and it may communicate with a specified set of other towers. (See Figure 1.)

The user will have an interface that allows him/her to perform the following actions:

- List all towers.
- List all phones associated with a particular tower.
- Change the location of a phone.
- Place a call from one phone to another, displaying the connections (phone-to-tower and tower-to-tower) that are used to connect the phones.

You must implement this program using *multiple source files*. The main function (and some helper functions) will be provided for you in the **main.c** file. You will implement two additional files: **Phone.c** will implement phone-related functions, and **Tower.c** will implement tower-related functions. You will be given a header file (**PhoneTower.h)** and you must implement all of the functions declared in that header file.

Command-Line Interface

When the program runs, it will ask the user for two file names. The first file gives information about the towers in the system. The second file gives information about the phones in the system. The formats of these files is described below. (An alternate way of running the program without typing the file names is described in Appendix C.)

The main function is provided for you, which reads from the files and also implements the user interface. So you don't have to write that code for this assignment.

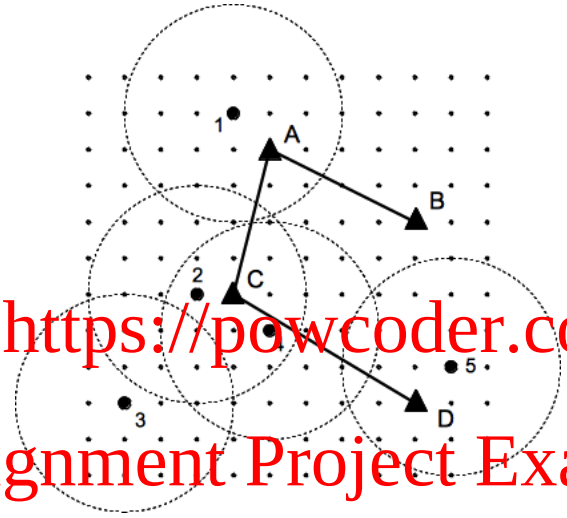

Figure 1: Example cellular phone system.
Triangles are towers, and solid lines represent connections between towers.
Dots are phones, and a dashed circle is the transmission range of that phone.

Program Description

Figure 1 shows an example cell phone system. The components are laid out on a grid, where each point has an x-y coordinate, where x and y are both integers. Point (0,0) is the bottom left corner of the grid; there are no negative coordinates. Example: The point labeled "3" is at point (1, 2).

The triangles (A, B, C, and D) are towers. Towers that are connected by a line can communicate with each other. For example, A can communicate with B and C, but not with D (because there is no connection between A and D). In a real cellular network, towers are connected via land lines or directional microwave links.

The closed circles (1, 2, 3, 4, and 5) are phones. Each phone has a transmission range of three units in all directions, as shown by the dashed circle around each.

A phone is associated with (at most) one tower, the one that is closest and within range. For example, phones 2 and 4 are associated with tower C; phone 5 is associated with tower D. Phone 3 does not have a tower within its range, so it is not associated with any tower.

Making a Call

A call between two phones is relayed via one or more towers. The originating phone communicates with its associated tower. If the receiving phone is also associated with the same tower, then the call will be completed by that single tower. If not, however, the tower must relay the call to one of its connected towers. That tower can also relay the call, etc., until the call reaches the tower associated with the receiving phone.

The list of towers that must cooperate to complete a call is called the *route* of that call.

For example, consider a call from phone 2 to phone 4. They are both associated with tower C, so no additional towers are needed. The route is: phone 2 / tower C / phone 4.

Consider a call from phone 1 to phone 5. Phone 1 is associated with tower A, but phone 5 is not. Tower A contacts its first neighbor (B) and asks it to find phone 5. Since phone 5 is not associated with B, and B has no neighbors, it tells A that it cannot find a route. A then asks its next neighbor, C. Since tower C

is not associated with phone 5, it asks its neighbor D. Tower D responds affirmatively, so C replies to A with the list of towers in the partial route: tower C / tower D. Tower A then prepends itself to the list of towers, to create the final result: phone 1 / tower A / tower C / tower D / phone 5.

In other words, to complete a call from phone 1 to phone 5, the phone contacts tower A, which relays the call to tower C, which relays the call to tower D, which completes the call to phone 5.

If no neighbor is able to find a route, tower A responds that it is unable to complete the call.

The algorithm for finding a route is given in a later section. It is important that you follow the algorithm exactly, so that all programs will find the same route.

The "null tower"

We don't want to keep a global list of all phones, but we also don't want to lose track of a phone in the system. Therefore, we will maintain a fictitious "null tower," that is used to keep track of phones that are not associated with any tower.

The null tower has the string "None" as its identifier. It is not connected to any other tower, so it may not be used to route any calls. Any phone that is not close enough to any tower (e.g., phone 3 in Figure 1) is associated with the null tower.

Note: Do not confuse the null tower with a NULL pointer (a pointer whose value is zero).

Other Details

The towers and phones in the system are specified by the configuration files provided by the user. No other phone or tower will be created during the execution of the program.

The number of phones is not known ahead of time. There is no upper bound on the number of phones that may be in the system.

The number of towers is not known ahead of time. There is no upper bound on the number of towers that may be in the system.

There is no upper bound on the x or y coordinates. In other words, the location grid extends infinitely in both the positive-x and positive-y directions. (In practice, this will be limited by the range of an integer, but you need not be concerned about this limit in your implementation. If you try to add too much to a coordinate, it will become negative, and will therefore be considered out of bounds.)

You must use linked lists in your implementation. It's the most logical data structure to use, but it is also required because the intent of this assignment is to give you experience using linked lists. If you do not use linked lists, major points will be deducted. (This does not mean that you can't use an array for something where it makes sense, but you must make significant use of lists.)

Implementation: Data Types

The header file **PhoneTower.h** defines four data types for this program, as follows:

```
typedef struct phone * Phone;          // a phone instance
typedef struct phoneNode * PhoneNode;  // node in a list of phones

typedef struct tower * Tower;          // a tower instance
typedef struct towerNode * TowerNode;  // node in a list of towers
```

We define the Phone and Tower types in terms of pointers, so that you have complete freedom in how you implement the data types. As we will discuss in class, this is known as an Abstract Data Type (ADT).

Typically, we would have separate header files for the Phone and Tower data types. However, the functions defined for these types have a circular relationship -- a Phone needs to return its associated Tower, and a Tower needs to return a list of its Phones. So it makes sense to include both types in a single header. Both implementation files (Phone.c and Tower.c) will need to include PhoneTower.h.

Implementation: Phone and PhoneNode

The **PhoneTower.h** header file specifies the following functions related to phones and lists of phones. You must implement these functions in a file named **Phone.c**. The **Phone.c** file must also define `struct phone`. The details of this struct os known only within **Phone.c** -- the details are not needed by any other file in the program.


```
Phone createPhone(const char *id, int x, int y);
```


This function creates a new `struct phone` and returns a pointer to it. The caller provides an identifier string (max. 12 characters) and the location (x and y). You may assume that the identifier is unique; there will be no other phone in the system with the same identifier. If either x or y is negative, the phone is not created and NULL is returned.

Note that the id argument is a `const char *`. This means that the `createPhone` function will not modify the caller's string. The function must make a copy of the string to store with the Phone instance; it may not simply use the pointer that is given by the caller. (The caller may choose to change its string after the call, and we don't want that change to modify the phone's id.)

Part of creating a phone is finding its associated tower. Use the phone's location and the list of towers (see below) to find the closest tower within range. The range for all phones is 3. The distance between point (x1,y1) and (x2,y2) is $\sqrt{(x1-x2)^2+(y1-y2)^2}$. Use the standard library's `sqrt` function to compute the square root. (You must include **math.h** to use that function.) If two towers are exactly the same distance from the phone, choose the tower whose name is alphabetically first (using `strcmp`). If no tower is within range, then the phone is associated with the null tower.


```
const char * phoneID(Phone p);
```

Returns the phone's identifier string. (String is returned as a pointer; the caller can look at this string, but may not use the pointer to change it.)

```
void phoneLocation(Phone p, int *xaddr, int *yaddr);
```

Returns the phone's location by storing the x- and y-coordinates to the memory locations pointed to by xaddr and yaddr, respectively.

```
Tower phoneTower(Phone p);
```

Returns the tower associated with the phone. The return value must not be NULL, but it may be the null tower (see above).

```
Phone findPhone(const char * id);
```

Returns the phone that has the specified ID string. If there is no such phone, the function returns NULL.

The `struct phoneNode` type is used to build a linked list of phones. Each node has a Phone (pointer to `struct phone`) and a next pointer. This is a "public" type and users of the PhoneNode type have direct access to these struct fields. Note the use of Phone (a pointer) as the data field. This means that multiple lists can refer to the same set of `struct phone` values, allocated on the heap (using **`createPhone`**).

Since the head of a list is a pointer to a node, the type PhoneNode is used to represent a list of phones.

Implementation: Tower and TowerNode

The **PhoneTower.h** header file specifies the following functions related to towers and lists of towers. You must implement these functions in a file named **Tower.c**. The **Tower.c** file must also define `struct tower`. The details of this struct is known only within **Tower.c** -- the details are not needed by any other file in the program.


```
Tower createTower(const char *id, int x, int y);
```


This function creates a new tower and returns a pointer to it. The caller provides an identifier string (max. 12 characters) and the location (x and y). You may assume that the identifier is unique; there will be no other tower in the system with the same identifier. If either x or y is negative, the tower is not created and NULL is returned.

Note that the id argument is a `const char *`. This means that the `createTower` function will not modify the caller's string. The function must make a copy of the string to store with the phone instance; it may not simply use the pointer that is given by the caller. (The caller may choose to change its string after the call, and we don't want that change to modify the phone's id.)

When created, a tower is initialized to have no neighbors and no associated phones. (The main function will always create towers before it creates phones.)

```
const char * towerID(Tower t);
```

Returns the tower's identifier string. (String is returned as a pointer; the caller can look at this string, but may not use the pointer to change it.)

```
void towerLocation(Tower t, int *xaddr, int *yaddr);
```

Returns the tower's location by storing the x- and y-coordinates to the memory locations pointed to by xaddr and yaddr, respectively.

```
void addConnection(Tower a, Tower b);
```

Creates a link between two tower instances. In effect, this adds b to a's list of neighbors, and adds a to b's list of neighbors. If a connection between the two towers already exists, this call is ignored. (Hint: Keeping the neighbor lists in alphabetical order will simplify other functions.)

```
void addPhone(Tower t, Phone p);
```

Adds this phone to the tower's list of associated phones. If the phone is already in the list, this call is ignored. (Hint: Building the list in alphabetical order will simplify other functions.)

```
PhoneNode towerPhones(Tower t);
```

Returns a list of phones associated with the tower. If there are no associated phones, the return value is NULL. The list will not be changed or deallocated by the caller. (Hint: If the tower keeps a list of phones in the proper order, it can simply return a pointer to this list.)


```
Tower nullTower();
```


Creates the null tower, if necessary, and returns a pointer to it. (If the null tower has already been created, the function returns a pointer to that instance.) (Hint: You can store a pointer to the null tower in a global variable. Initialize that variable to NULL, and use `createTower` to create the null tower when it is needed.)


```
Tower findTower(const char *id);
```


Returns a pointer to the tower instance with the specified identifier. If no such tower exists, the function returns NULL.

The `struct towerNode` type is used to build a linked list of towers. Each node has a Tower (pointer to `struct tower`) and a next pointer. This is a "public" type and users of the TowerNode type have direct access to these struct fields. Note the use of Tower (a pointer) as the data field. This means that multiple lists can refer to the same set of `struct tower` values allocated on the heap (using **`createTower`**).

Since the head of a list is a pointer to a node, the type TowerNode is used to represent a list of towers.

```
TowerNode allTowers();
```

Returns a list of all towers, except the null tower, sorted in alphabetical order. This list must NOT include the null tower. The return value may be NULL if there are no towers in the system. The caller must not deallocate any node in the list. (Hint: This list can be stored in a global variable.)

```
TowerNode towerNeighbors(Tower t);
```

Returns the head of a list of towers that are connected to the tower. The list must be sorted in alphabetical order. The caller must not deallocate any node in this list. (Hint: If the tower struct keeps a list of neighbors in the proper order, it can simply return a pointer to this list.)

```
PhoneNode towerPhones(Tower t);
```

Returns the head of a list of phones associated with the tower, sorted in alphabetical order. If there are no phones associated with the tower, the return value is NULL. The caller will not change or

deallocate this list. (Hint: If the tower struct keeps a list of phones in the proper order, it can simply return a pointer to this list.)

```
TowerNode routeCall(Phone start, Phone end);
```

Returns the head of a list of towers that can be used to route a call between two phones. If the call cannot be routed, the function returns NULL. The caller must not deallocate any node in this list.

This function MUST use the depth-first search algorithm described in the next section.

Implementation: Routing a Call

This section describes the algorithm used in the `routeCall` function discussed above. This algorithm only works if there are no cycles in the tower connections. (Using the example in Figure 1, a cycle would occur if B were connected to D.)

Inside the **Tower.c** file, we need to define a new function. For this discussion, I'll call it `routeInternal`, but you can name it whatever you like.


```
TowerNode routeInternal(Tower t, Phone p, Tower prev);
```

The third argument is necessary because the caller (*prev*) is a neighbor of *t*. But we don't want *t* to ask *prev* to find a route, because *prev* is already asking *t* to find a route! This would result in an infinite recursion.

A tower's neighbors are ordered alphabetically by ID. The `routeInternal` implementation uses the following algorithm to find a route from starting tower T to a receiving phone P, as requested by tower R:

1. If P is associated with T, return a list with a single node: T.
2. Else, let N be the T's first neighbor excluding R. If there are no neighbors other than R, return NULL (the empty list).

3. Let L be the list returned by routeInternal(N, P, T).
4. If L is not empty, prepend T to L and return the new list.
5. Else, set N to the next neighbor in T's list of neighbors (excluding R), and go back to Step 3. If there are no other neighbors, return NULL.

This algorithm is known as a "depth-first" search, because it checks all possible routes from the first neighbor before looking at the second neighbor.

To start the search, `routeCall` must do the following:

1. Get the tower associated with the starting phone. We'll call this S.
2. Call routeInternal on tower S and the ending phone P. Since there is no prior tower, the third argument should be NULL. This will be recognized as a special case, meaning that no neighbor is excluded from the search: routeInternal(S, P, NULL)
3. Return the list returned in step 2. If this list is empty, it means the call cannot be routed.

As noted above, the algorithm (as written) only works if there are no cycles in the connections among towers. We will not be detecting cycles, so we'll just have to trust that the topology of connections is cycle-free. Dealing with cycles is an interesting problem that you can think about later.

Implementation: Global List of Towers

Note that there are functions that ask you to return and/or search through a list of towers. You'll need to use global variables in **Tower.c** to accomplish this, because the list needs to exist beyond the lifetime of any single function call.

We can keep the name of these variables "private" to Tower.c by adding the `static` keyword before the type in the declaration. For example, to declare a file-private global integer variable named foo:

```
static int foo;
```


A static global is the same as any other global, but its scope is limited to the file in which it is declared. This is not absolutely necessary, but it's an easy way to avoid name conflicts among globals that do not need to be visible throughout the entire program. NOTE: This declaration must be in the implementation file (Tower.c), not the header file (Tower.h). Any function in Tower.c is allowed to use the variable, but functions in other files are not.

The `static` keyword can also be used for *functions* that are only needed in a single implementation file. Again, this limits the scope of that function definition to that file, avoiding name conflicts with other code in other files.

You may not access any global variable used to store the list of towers in ***Phone.c****. Use the allTowers function. If you do access the global variable, points will be deducted from your grade. You also may not keep a separate global list of towers in* ***Phone.c****.*

You may choose to keep a global list of phones in **Phone.c**, to simplify the `findPhone` function. But it's not required -- you can use the list of towers to look for the phone.

Implementation: Null Tower

There should also be a global variable for the null tower. This tower is used only to keep track of phones that are not close enough to any tower. The `nullTower` function, not the global variable, is used to access the null tower. This allows the **Tower.c** implementation to carefully control the null tower, making sure there is only one, and that the information for that tower is not overwritten by any other file.

The global variable should be declared as `static`, to keep other files from seeing it.

You may choose to allocate the null tower's `struct tower` value statically, as a global variable. Or you may choose for the global variable to be a pointer (`Tower`). In that case, initialize the variable as NULL, and allocate the value on the heap whenever the `nullTower` function is called for the first time. In either scenario, the `nullTower` function must return a pointer (`Tower`). and the function must always return the same pointer. There is, and will always be, only one null tower.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class.
- Work incrementally. Get one piece of the program to function correctly, and then move on to the next piece. This write-compile-test approach is a very good way to develop code.
- As soon as you get each part of the program working, submit it for testing. First, you'll have something submitted that should get you some points. Second, if you accidentally overwrite your source code file, or if your laptop disk crashes, we'll have a copy of what you submitted, and you can ask us for it instead of starting over from scratch. Third, you will be running the

completed parts of the code against the zyLab tests, and you can catch and correct errors early, rather than implementing the whole program with the same error in multiple places.

- 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 the "-Wall -Werror" flags to treat warnings as errors.
- Use the CLion debugger (or some other source-level debugger) to step through the program if the program behavior is not correct.
- 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 message and attach your code. Be specific about the problem you're having, and provide enough information for the person helping you to reproduce the problem.

Administrative Info

Updates or clarifications on the Message Board:

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

What to turn in:

- Two source files – they must be named **Phone.c** and **Tower.c**. Submit to the Program 4 assignment in the zyBook (13.4).

Grading criteria:

10 points: Proper coding style, comments, and headers. No unnecessary global variables. No goto. (See the Programs web page for more information.) These points are not awarded if only trivial code is submitted.

15 points: createTower, towerID, towerLocation, nullTower. All four of these functions must work properly -- all or nothing.

5 points: allTowers.

5 points: findTower.

10 points: addConnection, towerNeighbors. Both of these functions must work properly -- all or nothing.

15 points: createPhone, phoneID, phoneLocation. All three of these functions must work properly -- all or nothing. (For these points, phoneTower may return the null tower.)

5 points: phoneTower. (Requires createPhone to set tower properly.)

10 points: findPhone.

10 points: addPhone, removePhone, towerPhones. All three of these functions must work properly -- all or nothing.

5 points: movePhone.

10 points: routeCall.

NOTE: We will call your functions directly during testing, rather than using the user interface provided in main.c. Make sure that each function follows the spec exactly, and does not rely on anything in main.c.

NOTE: We will test your Phone functions using our Tower implementation, and vice versa. Don't rely on anything in your particular implementation of Phone to implement Tower, or vice versa.

Appendix A: Setting up the CLion Project

1. Create a new project for a C executable with the C11 standard.
2. Download the **main.c** file (overwriting the main.c created by your project).
3. Download **PhoneTower.h** into the same directory.
4. Download the **.txt** files into the same directory.
5. For each additional file (Tower.c, Phone.c), do the following:
    a. Right-click on the project name.
    b. Select New > C/C++ Source File.
    c. In the menu, set the file name (e.g., Phone) and select the .c file. Check the box next to "add to targets". Then click OK.

If you don't add to the target, then you will need to edit the CMakeLists.txt file to include the .c files in the list of files to be compiled for the target executable:

```
add_executable(XXX main.c Tower.c Phone.c)
```


The XXX can be anything, but it is generally the name of your project.

You will need to set the Working Directory, so that the program can find the .txt files.

Also, set the C compiler flags to be consistent with zyBook:


```
set(CMAKE_C_FLAGS "-Wall -Werror")
```


Appendix B: User Interface

You are not responsible for implementing the user interface, but you should know what it does, in order to run the program.

The main function will read the information from the configuration files (see below for format), and will call createTower, createPhone, addConnection, etc., to set up the system.

The program will then prompt the user for a command. The available commands are:

- **towers**: Prints information about all towers in the system. (No arguments.)
- **phones**: Prints information about all the phones associated with a particular tower, given as an argument by the user. Example: `phones A`
- **call**: Prints the route used to place a call between two phones. Takes two arguments: the ID of the originating phone, and the ID of the receiving phone. Example: `call 555-1234 555-4321`
- **move**: Moves an existing phone (specified by ID) to a new location. Takes three arguments: the ID of the phone, the new x-coordinate, and the new y-coordinate. Example: `move 555-1234 8 10`
- **quit**: Stops the program.

The main function will call your functions to carry out these commands.

File Formats

The tower file has two kinds of information: towers and connections.

A tower is specified by the following line:

```
tower <ID> <location-x> <location-y>
```

A connection is specified by the following line:

```
conn <tower1_ID> <tower2_ID>
```


The towers mentioned in a connection statement must have been created by a previous tower statement.

The phone file has ID and location information:

```
<ID> <location-x> <location-y>
```


There will be sample configuration files provided for you, but feel free to create your own for testing.

Appendix C: Running the Program

There are two modes for running the program.

First, if you just run the program as is, it will prompt you for two file names. The first file contains tower information, and the second file contains phone information. The program then prints a simple prompt character "?" and waits for you to type a command.

To save you some typing, there is a second mode for running the program, which uses command-line arguments. There are arguments that are provided by the user when the program is invoked, rather than entered by the user interactively. In Linux, these arguments would be typed on the same line as the program name -- hence, the term "command-line arguments."

In CLion, there is no shell and no command line. But you can specify command-line arguments through the Edit Configurations menu, which is the same place you set the Working Directory. Just enter the two file names in the **Program Arguments** box.

When the program runs, if it sees arguments, it will use them without prompting for the file names. This allows you to run the program on the same set of input files over and over, without having to retype the file names every time. (You're welcome.)

When you're ready to try with different files, just go back to Edit Configurations and change the Program Arguments. (Or delete them, if you prefer to type the file names.)