CSC 230 Project 4

Point-of-interest list

For this project, you're going to write a program that stores and manipulates a list of points of interest on a map (like parks, restaurants or museums), each with a location and a short description. You can add and remove points of interest from the list and generate reports of nearby points or points with descriptions containing a given word.

Your program will be called `attractions`. You should be able to run it as follows. Here, the user is entering commands to add several points of interest to the list, then asking for the list of places that are within 4.0 miles of the user's current location. After that, we remove a point from the list, enter a new current location for the user and then ask for points that contain the word "park" in their description.

```
$ ./attractions
1> add bell-tower 35.786107 -78.663513 Beautiful tower & an iconic Raleigh landmark.

2> add park-and-ride-3 35.854255 -78.792262 Place to park, with shuttle service to RDU.

3> add library 35.769246 -78.676410 Second main library of North Carolina State University.

4> add train-station 35.774942 -78.645720 Amtrak train station served by four daily passenger trains.

5> add waffle-house 35.777403 -78.676269 All-day breakfast, including signature waffles.

6> add nc-museum-of-art 35.810451 -78.703361 Collection spanning 5,000 years, outdoor monument park & amphitheater.

7> add pullen-park 35.779564 -78.663132 5th-oldest US amusement park, carousel, mini-train, paddle boats.

8> add crabtree-valley-mall 35.839736 -78.679632 Largest enclosed mall in the Triangle with over 220 stores.

9> nearby 4.0

library (0.3 miles)
  Second main library of North Carolina State University.
waffle-house (0.4 miles)
  All-day breakfast, including signature waffles.
pullen-park (0.8 miles)
  5th-oldest US amusement park, carousel, mini-train, paddle boats.
bell-tower (1.1 miles)
  Beautiful tower & an iconic Raleigh landmark.
train-station (1.6 miles)
  Amtrak train station served by four daily passenger trains.
nc-museum-of-art (3.1 miles)
  Collection spanning 5,000 years, outdoor monument park & amphitheater.
10> remove pullen-park

11> move 35.795529 -78.708268

12> match park

nc-museum-of-art (1.1 miles)
  Collection spanning 5,000 years, outdoor monument park & amphitheater.
park-and-ride-3 (6.2 miles)
  Place to park, with shuttle service to RDU.
13> quit
```

As with recent projects, you'll be developing this one using git for revision control. You should be able to just unpack the starter into the p4 directory of your cloned repo to get started. See the Getting Started section for instructions.

This project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 4

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow. For this

assignment, we're putting some constraints on the functions you'll need to define, the data structures you'll use and how you're going to organize your code into components. Still, you will have lots of opportunities to design parts of your solution and to create additional functions to simplify your implementation.

Requirements

This section says what your program is supposed to be able to do, and what it should do when something goes wrong.

Point-of-Interest List

The program maintains an arbitrarily long list of points of interest. We'll call each of these, *points*, and we'll call the whole list the *point list*. Each point has a unique string name, a global location given as latitude and longitude (both doubles) and a text description. The name will be a string of up to 20 non-whitespace characters.

Latitude values must be between -90.0 and 90.0 (inclusive), and longitude must be between -180.0 and 180.0 (inclusive). The description can be up to 1024 characters in length, and may contain spaces, but no other whitespace characters (e.g., no newlines or tabs).

Current Location

The program maintains a current location for the user, represented as latitude and longitude. By default, the current location should be latitude = 35.772325, longitude = -78.673581. This corresponds to the grass circle in the parking lot just north of EB 2. There's a command to let the user change the current location whenever they want to.

User Interaction

The program will repeatedly prompt the user for a command using the following prompt. There's a space after the angle bracket in this prompt (but you probably can't see it), and the number in the prompt should start at 1 and increment each time the prompt is printed. Putting a number in the prompt will help you track down errors in your program's output; since the prompts show up in the output, if your output isn't right, you can go back and see which command is causing your output disagrees with the expected output.

```
1>
```

At the prompt, the user can type any of 8 available commands, `add`, `remove`, `move`, `list`, `nearby`, `match`, `help`, or `quit`. These are described below. Each valid command starts with one of these keywords, possibly preceded by whitespace.

Each user command will be given as a single line of text. After reading a command, the program will print a newline before responding to the command. In the sample execution above, you can see this produces a blank line of output after each command the user enters. But, when output is sent to a file, it won't actually produce a blank line (since the output file will just get the program output, without any of the input entered by the user). Instead, we'll just get a newline at the end of each prompt. This should help to make the program output easier to read, even when we've sent it to a file.

Add command

The add command lets the user add new points of interest to the list. Like the following example, it's entered with the `add` keyword, followed by the unique name for the point of interest. This is followed by the point's location (latitude then longitude). After one or more spaces, all the remaining text up to the end-of-line is taken as the text description for the point.

```
add waffle-house 35.777403 -78.676269 All-day breakfast, including signature waffles.
```

On success, the new point is added to the point list. An add command would be invalid if its arguments didn't describe a valid point (e.g., if the name was too long or if the latitude or longitude values weren't in the right range). It would also be invalid to add a point with the same name as one that's already on the list.

Remove command

The remove command is for removing points from the list. It's entered as the keyword `remove` followed by whitespace, then the name of a point. For example, the following is a request to remove the point named "library" from the point list.

```
remove library
```

On success, the given point is removed from the list. A remove command would be considered invalid if its argument wasn't a valid name, or if it wasn't the name of a point currently on the point list.

Move command

The following is an example of the move command. It changes the user's current location to the given latitude and longitude.

```
move 35.785844 -78.667889
```

A move command would be invalid if its parameters wouldn't parse as values of type double, or if the given latitude and longitude values were out of range.

List command

The list command is entered as the word, "list". In response, the program should list all the points of interest currently on the point list. The output should be sorted by distance, with points closer to the user's current location listed first. If there's a tie in the distance to a point, you can report them in either order (although there shouldn't be any cases like this in the test inputs).

In the list, each point should be reported on two output lines. The first will give the name of the point, followed by a space and then the distance to the point in miles. Put the distance in parentheses, and round it to the nearest tenth of a mile. The next line should give the point's description, indented by two spaces. The two-space indentation makes it easier to see where each point starts in the list.

```
waffle-house (0.4 miles)
  All-day breakfast, including signature waffles.
```

If the point list is empty, the response to this command should be an empty list. Just print the newline character that's printed after each user command.

Nearby command

The nearby command lets the user list just the points that are near the user's current location. It's entered like the following example, with one real-valued (double) argument giving a distance in miles.

```
nearby 3.5
```

The output for the `nearby` command is just like the `list` command, but the output should only give points that are no farther than the given distance from the current location. The command would be invalid if the given argument was negative or if it couldn't be parsed as a double.

If there are no points that are sufficiently close, you should just print out an empty list (i.e., no output other than the the newline that's printed after every user command).

Match command

The match command lets the user list just the points with descriptions that contain a given word. It's entered like the following example, the word `match` followed by a word (`public` in this example). The given word must be a sequence of up to 20 lower-case letters.

```
match public
```

The output for the `match` command is like the `list` command, but the it should only give points that have the given word in their description. Matching the word should be case-insensitive. So, "`match public`" should match a description containing the word `public` or `Public` or even `PUBLIC`. However, it should only match whole words in the description. So, `public` shouldn't match words like `publication` or `republic` in the description.

A match command would be invalid if the given word was too long, or if it contained text other than lower-case letters. If there are no points that contain he given word, you should just print out an empty list.

Help command

When the user enters the `help` command, the program will respond with a report of the valid commands. It will print the following message to standard output, then prompt for another command:

```
add <name> <latitude> <longitude> <description>
remove <name>
move <latitude> <longitude>
list
nearby <distance>
match <word>
help
quit
```

Quit command and termination

The quit command doesn't take any arguments. It should terminate the program without prompting for any more commands. It's entered like the following:

```
quit
```

The program should also terminate successfully if it reaches the end-of-file on standard input while it's trying to the next command.

Valid and Invalid Commands

Commands and their arguments may be separated by any number of spaces. This should be easy to handle, since scanf() automatically skips whitespace when it's trying to match most conversion specifications. You can assume that commands will always be given on a single line. Although scanf() could easily parse commands that are split across multiple lines, we won't actually test your program on commands like this. That would make it trickier to discard invalid commands, and it would make it more difficult to use the number in the prompt to figure out which command had trouble.

If the user enters something other than a valid command, the program will print the following line to standard output, ignore (i.e., discard) the rest of the input line, and prompt for another command.

```
Invalid command
```

A command would be considered invalid if it it didn't start with one of the keywords or for the reasons noted in the previous sections.

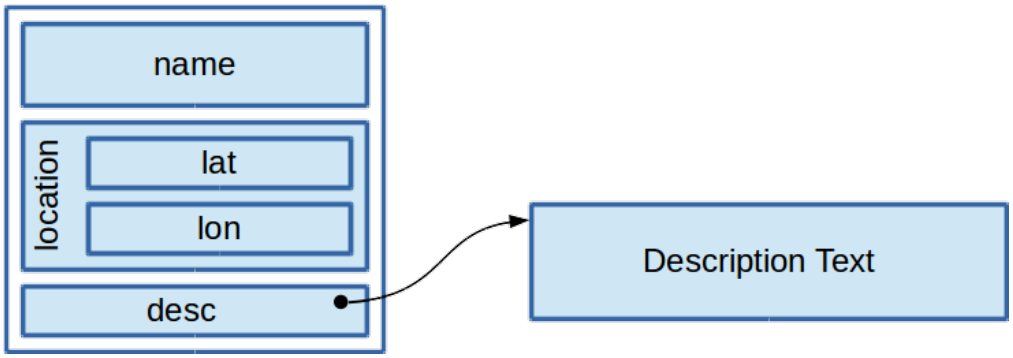
Design

Global Locations

We will be using a struct called `Coords` to store latitude and longitude values. This structure has already been implemented, in the starter.

Point List Representation

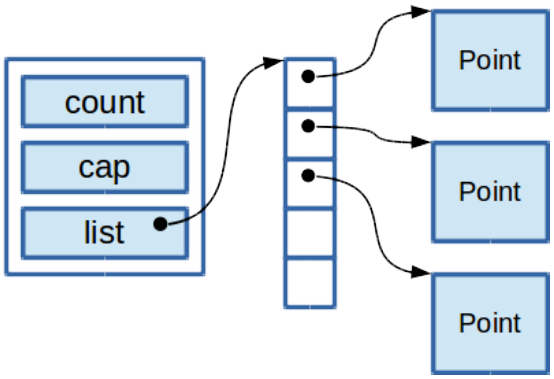
Each point of interest will be represented as an instance of the Point struct. This struct will contain an array of characters for storing the name, and an instance of the Coords struct, for storing the point's location. It will also contain a char pointer to a dynamically allocated char array holding the point's text description.



Representation for a Point

Storing the name right inside the struct makes it easier to work with, but it limits the length of the name. Storing the description elsewhere in memory is a little more complicated (we have to malloc() space separately for the description), but it gives us the flexibility of allocating differing amounts of space, depending on the length of the description. In your implementation, you should allocate the space for each point's description to be exactly long enough to hold the text of the description, plus the null terminator at the end.

The Pointlist is just a struct containing a resizable array of pointers to Point instances. It contains the fields you'd expect to support a resizable array, a pointer, `list`, to a dynamically allocated array of pointers to Point, an int field, `count`, for counting the number of Points currently on the list, and an int field, `cap`, for the current capacity of `list`.



Representation for the Pointlist

You'll need to write your own definitions of the Point and Pointlist structs. Use typedef to make the name, `Point` an alias for your Point struct and `Pointlist` an alias for your Pointlist struct. The functions described below assume that you these typedefs are available.

In defining your structs, remember:

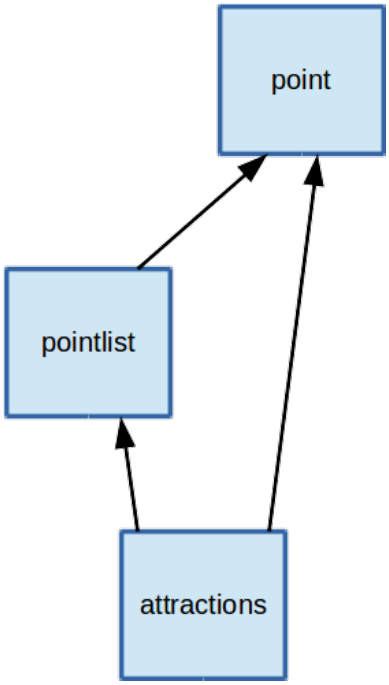
- In the Point struct, the name is stored inside the struct, as a char array field. In contrast, the description is accessed via a char pointer; it's stored elsewhere in memory.
- In the Point struct, the location is stored right inside the struct, as an instance of the Coords struct (not as a pointer to a struct stored elsewhere in memory).
- In the Pointlist struct, the array of pointers (to Point instances) has to be dynamically allocated elsewhere in memory. This will let us reallocate it as needed when more points are added. Since we're storing an array of pointer to `Point` instances, the field pointing to this array will need the type, pointer to pointer to `Point`. (Sorry. Right about now, I'm wishing I had called the first struct something other than `Point`)

Program Organization

Your source code will consist of three implementation files and two headers.

- `point.c` / `point.h`
  This component will define the Coords and Point structs, in the header, and functions for working with coordinates and individual points of interest in the implementation file. You don't have to try to enforce encapsulation. Other source files can directly examine the fields of a Point or Coords struct. You should just try to put the functions that work with Coords and Point in this component. There are some example functions described below.
- `pointlist.c` / `pointlist.h`
  This component will define the Pointlist struct, in the header, and functions for working with a Pointlist, in the implementation file. If you decide to create more functions to simplify your implementation, here's where to put them if they're specific to Pointlist.
- `attractions.c`
  This component will define the main() function and any other functions not provided by the other two components. It will be responsible for reading commands from the user and performing them (often by calling functions in the other components). This component will also include some functions that it passes to `listPoints()` (by address) to select which Points to report (see below).

The following figure shows the dependency among these components. This is the kind of thing you should think about when you're trying to design for code reuse and testability. In our project, the point component doesn't depend on any of the other components, so it should not use functions defined in pointlist or attractions. The pointlist component only depends on point, so it can use functions, types or constants defined in point, but it shouldn't depend on anything in attractions. The attractions component is our top-level component (although I know I put it at the bottom of the figure). I can use anything provided by the other two components.



Dependency structure for the components

Current Location

The attractions component will use a static global variable to keep up with the user's current location. Storing the location this way will make it easier to implement the `nearby` command, were we have to know how far away a Point is in order to decide whether to print it.

Expected Functions

As part of your implementation, you will define and use the following functions. You can define more if you want to. Just try to put them in a component that's suitable for whatever they do.

- `Point *parsePoint()`
  This function will create a new Point based on text read from standard input. It will be responsible for parsing all the arguments of the `add` command, everything after the word `add` itself. So, when it starts reading from standard input, it should expect to see a name, followed by latitude and longitude, then a text description that continues up to the end of the line. If successful, it returns a pointer to the new dynamically allocated Point. If the input doesn't contain valid arguments for the add command, it should return NULL. It's part of the point component.

- `void freePoint( Point *pt )`
  This function will free the memory for the given Point, including the Point struct itself and the memory for its description. It's part of the point component.

- `void reportPoint( Point const *pt, Coords const *ref )`
  This function should print out a description of the given point of interest, in the format required by the `list`, `nearby` and `match` commands. The Coords parameter gives the user's current location. The function can use this to report how far away the Point is. This is part of the point component.

- `double globalDistance( Coords const *c1, Coords const *c2 )`
  Given the addresses of two Coords, this function will return the distance between them in miles. This function is a lot like the one from exercise 13, except it uses two structs as parameters rather than four real values. You can adapt the version from the exercise to implement this function, or, if you want a little more fun, you can write it yourself. As in the exercise, you can assume the earth is a perfect sphere, with a radius of 3959.0 miles. This is part of the point component.

- `Pointlist *createPointlist()`
  This function will dynamically allocate an instance of Pointlist, and initialize its fields. The Pointlist should be initially empty, but with enough capacity to store a few points before it has to re-allocate the list of Points (really, pointers to Points) as a larger array. It's part of the pointlist component.

- `void freePointlist( Pointlist *ptlist )`
  This will free all the dynamically allocated memory used by a Pointlist, including the Pointlist object itself, its resizable array of pointers, and all the Point instances in the list. It will be part of the pointlist component.

- `bool addPoint( Pointlist *ptlist, Point *pt )`
  This function is part of the pointlist component. It will add the given Point to the given Pointlist list, resizing its internal array if necessary. It will return true if successful and false otherwise (say, if the given point has the same name as one that's already on the list).

- `bool removePoint( Pointlist *ptlist, char const *name )`
  This function is part of the pointlist component. It will remove the Point with the given name from the given Pointlist, returning true if successful or false if there's no Point with that name.

- `void listPoints( Pointlist *ptlist, Coords const *ref, bool (*test)( Point const *pt, void *data ), void *data )`
  This function is part of the pointlist component. It goes through the Points on the given Pointlist, printing selected ones. Its first parameter is the Pointlist it's supposed to print, and its second parameter is the user's current location. The user's location lets it sort the points by distance and report the distance along with each Point. As its third parameter, it takes a pointer to a test function. This is used to decide which Points to include in the output. By giving it different functions, we can get it to report all the points, or ones within a given distance from the user, or ones that include a given word in their description. The last parameter is a pointer to additional data that the test function can use to decide which points to print. The next section explains more about how this is supposed to work.

Selecting Points

The `listPoints()` function can be used to print any selected points on the point list. How does it know which points to print? Internally, it will call the provided test function for each Point in the list. If the test function returns true, listPoints() prints that Point; otherwise, it doesn't. This lets client code use a single interface to print any subset of the Points. The client code just needs to provide a pointer to a function listPoints() can use to decide what to print and what not to print. For example, to perform the `list` command, you can pass in a pointer to a test function that always returns true. To print points with a particular distance from the user's current location, you can pass in a pointer to a function that checks the distance to a given point and returns true if it's close enough. To do this, we'll need to use the `data` parameter to listPoints().

Notice that listPoints() takes a void pointer data parameter, and the test function also takes a void pointer data parameter. This parameter is a mechanism for providing extra information the test function needs in order to do its job, like the distance threshold for reporting points or a string to look for in a Point's description. When you call listPoints(), you can pass in a pointer to anything you want as the data parameter (even NULL, if you don't need this parameter). The listPoints() function will remember this parameter and will give this same pointer to the test function every time it calls it. This gives you a way to supply a pointer to anything your test function needs to answer the question, "Should we print this Point?". Inside each of your test functions, you will just need to convert the data value from a void pointer back to whatever it really points to before it can use it.

For example, to implement `nearby 3.5` command, our test function needs to know the distance threshold, 3.5. That's what the data parameter is for. You can put 3.5 in a variable, then pass that variable' address to the listPoints() function. You'll also give listPoints() the address of a test function that knows how to check the distance to a point against a threshold given in the data parameter. Each time listPoints() calls your test function, it will the test function the same pointer to the value 3.5. Inside your test, you can cast the data pointer back to a double pointer, then use it to decide if a Point is close enough to report.

You'll use the same technique to implement the match command, except for match, you'll use a pointer to the string you need to match, rather than a pointer to a double.

This is a common trick in C, using a void pointer to pass any information you need through general-purpose code and eventually back into code written for a specific purpose. When you take the operating systems class, you'll see a similar technique in the POSIX threads API, to pass arbitrary data to a new thread you're creating.

Sorting Points

Whenever you print the list of points of interest, you'll need to report them in sorted order, based on how far away they are from the user's current location. Since the user can change their location at any time (thus, changing the sorting order), we'll just re-sort the list right before we start reporting the points.

If you want, you can use the standard-library sort function, qsort() to sort your list of points. It will require some thinking, but, if you can get it right, it will probably shorten and speed up your implementation, compared to writing your own sort.

To use qsort() you'll need to think about a few things. As usual, you'll need to write your own comparison function, one that takes two (const) void pointers, but knows that they're really pointers to two elements of the array inside the Pointlist struct. So, it will need to cast these to pointers of the right type before it can start looking at the fields of the two points it's trying to compare. Remember that the comparison function gets pointers to two array elements (not the values of two array elements, **pointers** to the elements). So, since the array you're sorting is full of pointers to Point structs, your comparison function will get two pointers to array element that contain these pointers. You have to define your comparison so it takes two void pointer parameters, but, internally, your comparison function will know that these are really pointers to pointers to Point structs.

To do a comparison, you also need to know how far it is to the user's current location. You can either add a field to Point, to temporarily store the distance while you're doing the sort, or you can use a static global variable in pointlist.c, to temporarily hold a copy of the user's current location while you're doing the sort. Or, if you write your own sort instead of using qsort(), you will probably be able to

just use the ref parameter passed to listPoints().

Function Visibility

Any functions that are needed by a different component should be prototyped (and commented) in the header. Functions that don't need to be used by a different component should not be prototyped in the header, and should be marked static (given internal linkage), so they can't be used by another part of the program.

You'll need three test functions for passing to listPoints() (one to help implement `list`, one for `nearby` and one for `match`). These should be defined as static functions in the top-level, attractions component. These test functions will actually be called from inside the pointlist component, but pointlist doesn't really need to be able to see these functions during linking. Instead, it will be given pointers to these functions whenever listPoints() is called, and it will just call the test function using this pointer, without even knowing what function it's calling. This lets us maintin the dependency structure shown earlier; code in pointlist can use code from the top-level component without having to link with symbols defined in the top-level component.

Discarding Invalid Input

If the user enters an invalid command you'll need to discard the rest of the input line and report that it's an invalid command. This is like what you had to do in the previous project. If you want, you can get scanf() to discard this input for you, if you can figure out format string that tells it to skip the rest of the current input line. That's probably the easiest way. Or, you can just call getchar(), discarding its return value until you reach the end-of-line or you run out of input.

Build Automation

You get to implement your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build your program, compiling each source file to an object file and then linking the objects together into an executable. Since this program will need the constant for PI and the math library functions, you'll need to give the `-D_GNU_SOURCE` option when you compile and the `-lm` option when you link.

As usual, your Makefile should correctly describe the project's dependencies, so targets can be rebuilt selectively, based on what parts of the project have changed. It should also have a `clean` rule, to let the user easily delete any temporary files or target files that can be rebuilt the next time make is run (e.g., the object files, the executable and any temporary output files).

In addition to the "-Wall" and "-std=c99" options that we normally include when we're compiling, be sure to include the "-g" flag. This will be useful when you try to use `gdb` or `valgrind` to help debug the program.

Extra Credit

I'm planning to have an extra credit option for this assignment. I'm thinking it will involve implementing a line-breaking policy for printing long descriptions across multiple output lines, or maybe letting the user search for multiple words in point descriptions. Let me try out these options and see which one looks like a good choice for extra credit. Then, I'll fill in this part of the assignment.

Testing

The starter includes a test script, along with test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other test inputs we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build your program (using your Makefile) and see how it does against all the tests.

As you develop your program, you'll want to try it out with user input and see what it's doing on individual test cases. Until your Makefile is working, you should be able to execute the compiler directly with the following command (although this isn't as efficient as how your Makefile builds). As with exercise 13, the `-D_GNU_SOURCE` and `-lm` options let us use the math functions and the mathematical constant for PI.

```
$ gcc -g -Wall -std=c99 -D_GNU_SOURCE attractions.c pointlist.c point.c -o attractions -lm
```

To try out your program on one of the provided test cases, you can run it as follows. Here, we're saving the program's output to a file. After running the program, we check its exit status to make sure it ran successfully, and then we check the output file to make sure we got what was expected.

```
$ ./attractions < input-09.txt > output.txt
$ echo $?
0
```

```
$ diff output.txt expected-09.txt
```

Memory Error and Leaks

Your program is expected to free all of the dynamically allocated memory it allocates and close any files it opens. Although it's not part of an automated test, we encourage you to try out your executable with valgrind. We certainly will when we're grading your work. Any leaked memory, use of uninitialized memory, access to memory outside the range of an allocated block or leaked files will cost you some points. Valgrind can help you find these errors before we do.

The compile instructions above include the `-g` flag when building your program. This will help valgrind give more useful reports of where it sees errors. To get valgrind to check for memory errors, including leaks, you can run your program like the following. This example runs the program on input 15, an test input that uses the add and match commands. You can use similar commands to try your program on any input using valgrind.

```
$ valgrind --tool=memcheck --leak-check=full ./attractions < input-15.txt
-lots of valgrind output deleted-
```

Your program shouldn't need to open any files while it's running. It just uses standard input and standard output. You can still use valgrind to check for leaked files, if you want, but you'll probably just see standard input, standard output and standard error open when your program exits (which is OK).

Test Cases

We've prepared 20 tests for your program. They exercise the various commands your program is supposed to support, working from the easier ones to the more difficult tests and error cases. In developing your program, you may want to follow the order of these tests, adding the code to support the first test, then working toward the later (more complex) tests as you get the earlier ones working.

1. Runs just the quit command, to terminate the program immediately.
2. Runs the help command, the quits.
3. Runs the help command right before the end-of-file on the input, another way to end the program.
4. Contains an invalid command name then a quit.
5. Uses the add command to add one point to the list.
6. Tries to add two points with the same name, so the add of the duplicate name should be invalid.
7. Adds three points to the list, already ordered by distance (so this test will work even before you implement the sort), then uses list to print them out.
8. Adds some points that aren't ordered by distance, so this will let you test your sort.
9. Tests the remove command by adding several points, then removing some of them.
10. Includes an attempt to remove a point that doesn't exist.
11. Removes a point, then adds another one with the same name.
12. Uses the nearby command to list places within 1.5 miles.
13. Uses the move command to move over to Daniels hall, then lists nearby locations.
14. Includes an invalid move command, where longitude doesn't parse as a double. This shouldn't change the current location.
15. Test of the match command, with the word "park" which shows up in two descriptions and "old" which only shows up in one.
16. Example given at the start of the assignment, showing several different commands.
17. A test of the match command, with some invalid words to search for.
18. A large test, adding and removing a few hundred points of interest with randomly-generated descriptions. It also tests the match command near the end.
19. Adds a point with a description with the maximum possible length, and another that's too long.
20. Tries to add a point with a name that's the maximum length, one that's too long and points with latitude and longitude that are out of range.

Grading

The grade for your program will depend mostly on how well it functions. You also get to provide your own Makefile, and we'll expect your program to compile cleanly, to follow the style guide and to follow the expected design.

- Compiling cleanly on the common platform: **10 points**
- Working Makefile: **5 points**
- Behaves correctly on all tests: **80 points**
- Program follows the style guide: **20 points**
- Probably some extra credit, but I'm not sure what it's worth yet.
- Deductions
  - Up to **-60 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - Up to **-30 percent** for exhibiting file leaks, memory leaks or other memory errors.
  - **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the `p4` directory of your repo. You'll submit by committing files to your repo and pushing the changes back up to the NCSU github.

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't already done this, go back to the assignment for project 2 and follow the instructions for cloning your repo.

Unpack the starter into your cloned repo

Make sure you're in the `p4` directory in your cloned repo. You will need to copy and unpack the project 4 starter. We're providing this as a compressed tar archive, starter4.tgz. After you download this file, you can unpack its contents into your `p4` directory. As with previous assignments, remember there's a `.gitignore` file that needs to be there, even though this file won't show up (by default) in a directory listing.

As usual, if you are logged in on one of the common platform systems, you can save yourself a few steps by unpacking the starter directly from our official copy in AFS. Be sure you're in the `p4` directory of your repo and run:

```
$ tar xzvpf /afs/eos.ncsu.edu/courses/csc/csc230/common/www/proj/p4/starter4.tgz
```

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `attractions.c` : source file, created by you.
- `pointlist.c` : source file, created by you.
- `pointlist.h` : header file, extended by you.
- `point.c` : source file, created by you.
- `point.h` : header file, provided with the starter but extended significantly by you.
- `Makefile` : the project's Makefile, created by you.
- `input-*.txt` : test inputs given to the program on standard input, provided with the starter.
- `expected-*.txt` : expected output from the program, provided with the starter.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file provided with the starter, to tell git not to track temporary files specific to this project.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command. You should only need to add each file once. Afterward, you can get git to automatically commit changes to that file:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Remember, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ unset SSH_ASKPASS # if needed
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to the previous project. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 4. This job polls your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Explain, inspect, and implement programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.
- Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O.
- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.
- Describe and demonstrate how to avoid the implications of common programming errors that lead to security vulnerabilities, such as buffer overflows and injection attacks.