EECS 281: Project 1 - Treasure Hunt

Introduction

It's a Pirate's Life for me, for me, it's a Pirate's Life for me!

Nothing beats a hunt for buried treasure. The only problem? There's no map! But a wily captain and hardy first mate have teamed up for a treasure hunt. They've conscripted you to write a program to lead their hunt for riches in and around a chain of islands. The Captain will sail the ship over the water, and the First Mate will lead the search party on land.

A Chain of Islands

The chain of islands will be specified in a text file, and can be represented as a 2D-grid size N x N, with N >= 2. The text file will either be a map or a list of grid coordinates and terrains. The following ascii characters will be used to describe terrain.

- .: Water
- o: Land
- #: Impassable (land or water)
- @: Starting location (always water)
- $: Treasure location (**always land**)

The grid is a complete representation of all relevant features in the chain but it does not include a path from the start to the treasure. Using the algorithm described below and input from the command line and a file, you will try to find and describe the path, if one exists. In either case, the details of the hunt must be displayed, or you will be forced to walk the plank!

Example Map

```
# Spec Map
# Two small islands with treasure in the NW corner and start in the SE corner.
M
5
o$...
oo...
#..o.
..oo.
....@
```

Routing Scheme

The Captain and the First Mate don't always see eye-to-eye, and will sometimes favor different methods of exploration. Each one has a default method for searching, and this can be overridden (more on this later).

The Captain will travel from island to island with their own log and future plans, moving only over water locations (`.`). As soon as land is discovered while sailing, a search party is put ashore before any other discovery is attempted. Once put ashore to lead a search party, the First Mate will search all available land with a separate log and plans before returning to the ship, moving only over land locations (`o`). Any water (inland or open ocean) encountered by the search party is completely ignored. Neither the Captain nor the First Mate will ever accidentally overlook treasure (`$`), and both are unable to move over impassable terrain (`#`).

All investigation happens in squares adjacent to the current location, whether on land or water. New locations can be discovered by looking one square in each of the cardinal directions: North, East, South, and West. Discovery will never happen diagonally or at a distance greater than one. Each square can only be discovered once; get these pirates stuck in an infinite loop, and you will most surely be keelhauled.

To satisfy the moods of your "employers", you develop two routing schemes, one that uses a stack and one that uses a queue to store newly discovered locations. The Captain and First Mate can choose from either of these container types. Stack-based routing tends to continue investigation along a recent heading, eventually returning to locations that were passed in order to cover the hunt area, while queue-based routing covers the hunt area in ever-increasing radii.

Investigation and Discovery

Never use the term visited, or think in terms of locations visited.

While trying to find treasure and create a map, the important terms are **investigation** and **discovered**.

Investigation is the act of examining squares adjacent the "current" location. An adjacent square can only be discovered if it is the proper terrain type (water while sailing, land while searching) and has not already been discovered.

A newly discovered square should be marked as discovered immediately, to prevent infinite loops while investigating. This discovered location should then be added to the proper container, so that investigation can continue. If the location added to the container is the treasure location, it is important to stop investigating immediately.

When a location is removed from a container, it becomes the current location, and investigation begins again. Since a location should only be added to a continer once (discovered), it can only become the current location once (investigation).

The Hunt Order

When discovering new locations use the Hunt Order to determine the order of investigation. By default the hunt order is North->East->South->West. Any expedition can choose another hunt order at the command line, but it must include each of the cardinal directions exactly once.

The Hunt Algorithm

The treasure hunt will be one large hunt (over water) with zero or more smaller hunts (over land). The Captain always begins the hunt from the start location (`@`), which is a water location. The First Mate begins a search party subhunt from a land location that was discovered by the Captain. Use the following hunt algorithms with the container (stack or queue) and initial location that suits the hunter in charge.

The Captain's Hunting

1. Add the Starting Location (`@` on the map) to the sail container.
2. If the sail container is empty, the hunt has ended (jump to Step 5). If not, set the "sail location" to the "next" available location in the sail container (where next is front for queue, top for stack) and remove it from the sail container.
3. From the sail location, add any adjacent water locations that are not impassable and have not already been discovered to the sail container. Discover new locations as dictated by the Hunt Order. The Captain will only add water locations to the sail container.
    - If the Captain discovers land, the First Mate will be immediately put ashore (before the captain examines other adjacent locations) to start a search party at that location using a separate container; jump to Step 1 of the First Mate's Hunting. When the First Mate finishes hunting, if the treasure was found, jump to Step 5. If the treasure was not found, the Captain should continue investigating any remaining locations around the sail location.
4. Repeat from Step 2.
5. Report the outcome of the hunt (see Output Format).

First Mate's Hunting

The First Mate performs their hunting in a method very similar to the Captain, only on land.

1. Add the Starting Location (wherever the search party was put ashore) to the search container.
2. If the search container is empty, this land hunt has ended (jump to Step 5). If not, set the "search location" to the "next" available location in the search container (where next is front for queue, top for stack) and remove it from the search container.
3. From the search location, add any adjacent land locations that are not impassable and have not already been discovered to the search container. Discover new locations as dictated by the Hunt Order. The First Mate will only add land locations to the search container. If the First Mate encounters water it will be ignored regardless of whether it has already been discovered or not.
    - If any location added to the search container is the treasure (`$`), end this subhunt, because the existence of a path has been found; jump to Step 5.
4. Repeat from Step 2.
5. Report the outcome of the land hunt to the Captain (pick up where the Captain left off in The Captain's Hunting Step 3).

Command Line Interfaces (CLI)

The solution you generate will run from the command line, in the spirit of the Unix tradition.

Silence is Golden

Most programs run with as little output as possible. Often, programs that are able to successfully execute will print nothing at all. Programs that generate output will usually display the smallest possible output that registers success or failure. By default, this solution will generate a single line of output.

Modifying Behavior

Programs often offer *options* that can change how they work at runtime. These are specified at the command line in the form of "short options" which are a single character following a single hyphen (eg. -o), or "long options" which use full words following a double hyphen (eg. --a-long-option). Multiple short options can be combined (eg. -al is equivalent to -a -l). Both short and long options can also accept *arguments*, with some options prohibiting, requiring, or optionally allowing arguments.

Q: Why both short and long options?

A1: There are only 52 different single letter options, but an infinite number of long options can be created.

A2: Two programs can implement the same functionality using different options (eg. -R or -r for --recursive), but full words are more easily remembered.

A3: Short options are quick and easy to type at the command line, and when programs are used in scripts, long options make scripts more readable.

Like many programs, the solution to this problem will implement both short and long options. Some options will be implemented with no arguments, and others will be implemented with required arguments. **An option with a required argument is still optional and might not be specified at the command line, but if it is specified, an argument must be provided.**

The complicated task of parsing options and arguments is made easier with a classic library `<getopt.h>` that provides both `getopt()` for handling short options and `getopt_long()` for handling short and long options. A helpful reference can be found at getopt man page.

Using Standard Input, Standard Output, Standard Error, and Redirection

Most command line shells allow programs to send output to or read input from files. This is accomplished by using the Input Redirection Operator (<) or the Output Redirection Operator (>). The following example runs `hunt` , with the `-v` (verbose) option specified, while reading input from the file `input.txt` and writing output to the file `output.txt` :

```
$ ./hunt -v < input.txt > output.txt
```

Reading input from a file can be thought of as temporarily disconnecting the keyboard and getting all input from the file specified. The file must exist or an error will occur. Also, problems will arise if the data in the input file is not saved in the exact order that it is requested by the program. This technique is known as input redirection, and replaces keyboard input with "standard input". Standard input can be accessed in C++ by using `std::cin` .

Writing output to a file can be thought of as temporarily disconnecting the screen and sending everything that would be printed directly to a file on disk. The file need not exist before redirection, it will be created as necessary. If the file already exists, its original contents will be replaced (gone forever) with the new output. This technique is known as output redirection and sends text that would go to "standard output" and display on the screen, directly to the given file. Standard output can be accessed in C++ by using `std::cout` .

Redirection can be done on input, output, both, or neither, as well as on "standard error" (which is accessed with `std::cerr`) and redirected with the Error Redirection Operator (2>). A good reference can be found at Thoughtbot.

Performing input redirection is especially useful with large input files: you wouldn't want to retype 10,000 lines of input, and even copying and pasting it could become irritating. Redirection is generally transparent to your program while it is running. The only exception to this is that many IDEs, such as Xcode, cannot perform redirection while running *inside* of the IDE; we fix this by using `xcode_redirect.hpp` .

Input File

The input file representing the grid where the hunt will take place will be provided in one of two ways: a 2D ASCII map, or a list of coordinate/terrain triples (CTT). Both files will have similar front matter:

1. Zero or more lines of comments, each of which...
   - have an octothorpe (#) in column zero
   - can contain any combination of zero or more printable ascii characters
   - end with a newline character
2. A filetype specifier, which is...
   - a single ASCII character
   - either M or L (for map or list files)
   - followed by a newline character
3. A map size value, which is...
   - a positive integer, >= 2
   - both the width and the height of the map
   - followed by a newline character

The starting location is always a water square, and the treasure location is always a land square.

Map Format

If the first non-comment byte is 'M', the input file is in map format. There will be N rows of N characters (plus a newline), where N is the map size value that is on the line immediately following the filetype specifier. Each of the N characters must be from the Terrain Legend above.

See the Example Map above for a sample of a map format file.

List Format

If the first non-comment byte is 'L', the input file is in list format. Following the map size integer, there will be two or more lines (start location and treasure location at least) that are coordinate/terrain triples (CTT), and any number of blank lines. A CTT is two non-negative integers (row and column) followed by a single character to represent the terrain at that location (refer to the Terrain Legend above). These three values will be separated by spaces and followed by a newline character. When reading list format files, ignore any blank lines that occur.

```
CTT: <row> <col> <terrain>
```

List format files may specify a subset of all CTT in a map. If fewer than NxN CTT are provided, missing locations are assumed to be water (`.`). No location will be specified more once in a list format file, so no more than NxN locations can be included. The order of CTT is unspecified.

One possible list format file of the example map above follows.

Example List Input

```
# Spec Map
# Two small islands with treasure in the NW corner and start in the SE corner.
L
5
0 1 $
4 4 @
2 0 #
0 0 o
1 0 o
1 1 o
2 2 .
2 3 o
3 2 o
3 3 o
```

The Command Line Interface (CLI)

```
usage: ./hunt [options] < inputfile
```

Supported Options

- --help, -h: Print a useful help message and exit, ignores all other options
- --captain <"QUEUE"|"STACK">, -c <"QUEUE"|"STACK">: The route-finding container used while sailing in the water (if unspecified, container default is stack)
- --first-mate <"QUEUE"|"STACK">, -f <"QUEUE"|"STACK">: The route-finding container used while searching on land (if unspecified, container default is queue)
- --hunt-order <ORDER>, -o <ORDER>: The order of discovery of adjacent tiles around the current location, a four character string using exactly one of each of the four characters 'N', 'E', 'S', and 'W' (if unspecified, the default order is: North->East->South->West)
- --verbose, -v: Print verbose output while searching
- --stats, -s: Display statistics after the search is complete
- --show-path <M|L>, -p <M|L>: Display a treasure map or the list of locations that describe the path

Output Format

Treasure Hunt Results

After a treasure hunt, one line is printed describing the success or failure of the hunt:

```
No treasure found after investigating 5 locations.
```

or

```
Treasure found at 0,0 with path length 8.
```

The number of investigated locations, the treasure location, and the path length are all calculated or found based on a hunt of the input file, given the command line options provided. Whenever a location is needed, it is always given in `<row>,<col>` format.

Option: Verbose

If the verbose option is specified at the command line (`--verbose` or `-v`), output additional information while the hunt is happening. Verbose output, if requested, will always appear before the results and will consist of a start message, zero or more ashore messages, one search party result for every trip ashore, and a failure message if no treasure is found.

```
Treasure hunt started at: 4,4
Went ashore at: 3,3
Searching island... party returned with no treasure.
```

```
Went ashore at: 1,1
Searching island... party found treasure at 0,0.
```

or

```
Treasure hunt started at: 4,4
Went ashore at: 3,3
Searching island... party returned with no treasure.
Treasure hunt failed
```

Option: Stats

If the stats option is specified at the command line (`--stats` or `-s`), output a statistical summary after the hunt has completed. This will appear after verbose messages (if both are specified), and before the "Treasure Hunt Results."

```
--- STATS ---
Starting location: 4,4
Water locations investigated: 8
Land locations investigated: 5
Went ashore: 2
Path length: 7
Treasure location: 0,1
--- STATS ---
```

The total lines show the number of tiles that were investigated, or were the current location at some point in time. If a location is added to a container but it is never removed, it is *NOT* counted in its respective total. A line displaying the number of times a search party went ashore must be included. If a path to the treasure is found, include a line with the length of the path, and another line with the location of the treasure. Path length doesn't count the starting and ending locations, but rather the the steps between them. A hunt with the start location adjacent to the treasure location will have a path length of one.

```
# Treasure hunt with path length one
M
2
.$
.@
```

```
# Treasure hunt with path length two
M
3
.$o
.o.
.@.
```

If no path to the treasure is found, omit the last two lines (Path length and Treasure location) of stats output.

```
--- STATS ---
Starting location: 4,4
Water locations investigated: 8
Land locations investigated: 5
Went ashore: 2
--- STATS ---
```

Option: Show Path as Map or List

If the show path option is specified at the command line (`--show-path` or `-p`), output the path discovered from the start location to the treasure. **If no path to the treasure is found, nothing should be printed.** The show path option requires an argument which will display a treasure map when `M` is provded, or display the locations that define the path when an `L` is provided. If any other argument value is provided, the program should exit with error. The path will appear after verbose messages and stats (if either or both are specified), and before the "Treasure Hunt Results."

Treasure Map

A "treasure map" is the original map with a path overlaid from the start location to the treasure. The path is drawn using vertical lines (`|`), horizontal lines (`-`), and corners (`+`), with the start location unchanged (`@`) and the treasure overwritten with an `X` .

```
oX...
o|...
#|.o.
.|oo.
.+--@
```

Coordinate List

A "coordinate list" displays the path as a collection of row/column coordinates that trace from the start location to the treasure location. The list should be divided into two sections, with sailing locations listed first, followed by the search locations of the final search party. Include labels at the beginning of each section.

The start location will be the first coordinate and the treasure location will be the last coordinate.

```
Sail:
4,4
4,3
4,2
4,1
3,1
2,1
Search:
1,1
0,1
```

The number of coordinates should be one more than the length of the path. For more on the path length versus the number of locations, and the off-by-one error that can result from this type of problem, read off-by-one.

Error Handling and Assumptions

All input files will be well formed. This means that any test cases given will represent a map exactly as described above, with no extraneous or invalid characters, no invalid coordinates, and no formatting that conflicts with the descriptions in "Input File" above.

Working with users can be a challenge as they are prone to error, oversight, and even bad judgement. Since it is assumed that input files will not contain errors, it will only be important to handle errors at the command line.

1. If Captain or First Mate option is specified, the argument provided must be either "QUEUE" or "STACK"
2. If Hunt Order option is specified, the argument provided must be four characters long
3. If Hunt Order option is specified, the argument provided must contain one and only one of each of "NESW" (in any order)
4. If Show Path option is specified, the argument provided must be 'M' or 'L'
5. The Show Path option can only be specified once
6. All short or long options not in the spec should result in program termination with a non-zero exit code

If any of these errors are discovered, you can immediately `exit(1)`, or have `main()` perform a `return 1`.

Standardized Error Messages

If an input is valid (not one of the `INV` test cases) and you output one of these allowed error messages (and have an exit status of 1), the autograder will show you the error message, which will help you with debugging. For example: you rejected a valid input, thinking that we gave an invalid argument to `--captain`. If you receive one of these error messages for a valid test case, you know more about where to start debugging. If you want to have more debugging help, you can add a second line after the first. The autograder won't show it to you, but it could be useful for your own testing. For example:

```
    cerr << "Invalid argument to --captain" << endl;  // autograder displays
    cerr << "  Invalid argument is: " << optarg << endl;  // personal debug
```

Some of these (the last few) may not be in the Error Handling and Assumptions section, but are still useful for debugging purposes. For instance, if you made your own test file with list mode input of size 5, but specified row 5, it would be easier to print an error message about invalid input than to try to debug why your program is segfaulting.

Approved messages

```
Invalid argument to --captain
Invalid argument to --first-mate
Invalid argument to --hunt-order
Invalid argument to --show-path
Specify --show-path only once
Unknown option
Map does not have treasure
Map does not have a start location
Invalid coordinates in list mode input
Invalid terrain type
```

Submission to the Autograder

Keep all source files, header files, test files, and a Makefile in a directory dedicated to this project. This will be the "submit directory". The autograder will manage each submission by using `make` and the included Makefile.

Makefile requirements

The following targets and behaviors are required of the Makefile by the autograder.

- `make -R -r` : default target that builds the project without errors and generates an executable file named `hunt` as a "release build"; options -R and -r disable automatic build rules, which do not work on the autograder; additionally, the release build should specify -O3 for optimization and should not specify -g to include debug symbols
- `make clean` : delete all .o files and executable(s)
- `make debug` : optional; builds the project without errors and specifies the -g option, generating an executable that includes debug symbols (with no change in program output) named `hunt_debug` ; this may be used by the autograder to help diagnose problems

If the Makefile provided with the project specification is used, the following targets (and more) will be included as well as the three listed above.

- `make help` : displays a help and usage message for targets in the Makefile
- `make all` : builds both `hunt` and `hunt_debug` as specified above
- `make fullsubmit` : builds a "tarball" named `fullsubmit.tar.gz` that contains all source and header files, test files, and the Makefile; this file is to be submitted to the autograder for any completely graded submission
- `make partialsubmit` : builds a "tarball" named `partialsubmit.tar.gz` that contains only source and header files, and the Makefile; test files are not included, which will speed up the autograder by not checking for bugs; this should be used when testing the simulation only

The Makefile must compile all code using version 6.2.0 of the g++ compiler. This is available on the CAEN Linux systems (accessible via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 6.2.0 running on CAEN Linux. Note: In order to compile with g++ version 6.2.0 on CAEN and the autograder, include the following at the top of the Makefile:

```
PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

Source and Header Files

By default, the Makefile recognizes source and header files with these extensions: .h .hpp .cpp. As a general rule, header files should never be compiled directly and source files should never be `#include` 'd. Each header and source file include must have the "Project Identifier" included in a comment. This can be done by adding this line:

```
// PROJECT IDENTIFIER: 40FB54C86566B9DDEAB902CC80E8CE85C1C62AAD
```

The autograder will not compile any submission that does not contain the project identifier in every source and header file. If additional file types are preferred (eg.: .hxx, .cxx, cc), those must be included by manually editing the Makefile.

Buggy Solutions and Test Files

A small portion of the grade is be based on creating test files that can expose bugs. In addition to the correct solution, the autograder has several buggy implementations of the project. Each buggy implementation is based on the instructor solution, but has small bugs (either based on implementation mistakes or misreadings of the specification).

The goal is to submit input files that cause the buggy implementations to produce different output than the correct implementation. Each submission can include up to 15 test files to the autograder (though it is possible to get full credit with fewer test files). Each test file must be at most size 10. The instructor solution will reject test files that are invalid. If a test file is submitted that causes the instructor solution to exit 1, then that file will not be used to detect bugs.

Part of the reason to submit these test files is to create tests that can be used locally, without using the autograder, to verify submissions. For the first test file submitted that produces wrong output (if any), the autograder will return both the correct output and the wrong output with its feedback! Please use the autograder as the helpful tool it is, and submit test files early and often! Don't wait until the last minute to submit these.

Each test file should be a valid input file named `test-n-options.txt` where 1 <= n <= 15. The "options" portion should specify any command line options that should be applied with the given test. Use only lowercase, short options, and if an option has an argument, append that all uppercase.

Valid options in the filename are:

c: Set captain sail container to given (eg. `cSTACK` or `cQUEUE`) f: Set first mate sail container to given (eg. `fQUEUE` or `fSTACK`) o: Set hunt order to given (eg. `oNESW` or `oENWS`) v: Show verbose output s: Show stats output p: Show path in mode given (eg. `pM` or `pL`)

These may be combined with or without hypens between options like `test-1-vspLcQUEUE.txt` or `test-2-v-s-pL-cQUEUE.txt`).

The flags specified as part of the test filename should only produce valid command lines. Don't include any option more than once or with missing or invalid arguments. Any options not specified in the filename will not be included and runtime will use the defaults specified above.

There are ten buggy implementations on the autograder, to receive all ten points for bugs, nine of them must be found. To receive some points, five bugs must be found.

Tarball Requirements

A "tarball" is a compressed archive file with either a .tar.gz or .tgz extension. The provided Makefile creates tarballs named `fullsubmit.tar.gz` or `partialsubmit.tar.gz`, but the autograder will accept any file of this format.

- Include only the source and header files that are required to build the `hunt` executable
- Include up to 15 properly named test files
- Include a Makefile
- The total size of all files included cannot exceed 2MB
- No unnecessary C++ files, text files, or other junk from the submit directory
- All files are submitted in a single "flat" directory, with no subfolders

This can be done manually at the command line with the following command inside the submit directory:

```
$ tar cvzf submit.tar.gz *.cpp *.h *.hpp Makefile test*.txt
```

How to Submit

Submit a tarball directly to either of the two autograders at: ag1 or ag2. Load balancing is done at the time of submission: if there are 10 submissions in the queue on autograder 1 and none for autograder 2, submit to autograder 2. Do not try to submit to both autograders at once! It is safe to ignore and override any warnings about an invalid security certificate.

The autograders are identical and the daily submission limit will be shared (and kept track of) between them. The autograders will accept three Project 1 submissions per day, per student (more in Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). The highest graded submission will be used during "final grading" to determine the score of the project.

When the autograders are turned on and accepting submissions, there will be announcements made in lecture, in lab, and on Piazza. There is *NEVER* any reason to contact the course staff to ask when the autograder will be up. It is always a top priority and either up and running or being worked on.

Grading

Please be sure to read all messages shown at the top of the autograder results! The messages before the scoring of individual test cases are equally as important as the scores below, and will often explain outstanding issues the autograder finds (such as losing points for having a bad Makefile).

- 80 points: Grades will be primarily based on the correctness of algorithm implementation. For full points, each submission must have correct and working stack and queue algorithms, support both types of input and output modes, and properly handle all command line options. Additionally: Part of the grade will be derived from the runtime performance of the submission. Fast, correct implementations will receive all possible performance points. Slower implementations may receive only a portion of the performance points.
- 10 points: Test file coverage (effectiveness at exposing buggy solutions).
- 10 points: Memory leak check with valgrind

All grading will be done by the autograder. By default, we will use your best submission for final grading. If you would like us to use your LAST submission instead, use this Google form.

Memory Leak Check

Testing

Hints and Advice

- Design data structures and work through algorithms on paper first. Draw pictures and make bulleted lists. Consider different possibilities before beginning to code. If there are problems at the design stage, come to office hours. After some design work is done and a general understanding of the assignment and its scope is achieved, re-read this document. Consult it often during development to ensure that all code is in compliance with the specification.
- Always think through data structures and algorithms before coding. It is vital that efficient algorithms and implementations are used in this project and in this course, and coding before thinking often results in inefficient algorithms and/or implementations.
- Before trying to use linked lists, be sure to review the lecture slides or measure their performance against `vector<>` first (theoretical complexities and actual runtime can tell different stories).
- Only print the specified output to standard output.
- Any diagnostic information may be printed to standard error (`cerr`) without penaltly. However, make sure it does not scale with the size of input, so that time and memory limits are not exceeded.
- If the program does find a route, be sure to have `main()` perform a `return 0`. If the input is valid but no route exists, also have `main()` perform a `return 0`. The only time you should use `exit()` is when you've found a command-line error (see Error Handling and Assumptions above); in these situations you should `exit(1)`.
- It is strongly recommended that some form of version control (eg.: git, SVN, etc.) is used, and that files are committed at least as often as submissions are made to the autograder. If an online version control system is used, make sure that all projects and files are

PRIVATE; many sites make them public by default! Any code found by another student and used, even without permission, could trigger Honor Code proceedings for both parties.
- This is not an easy project. Start it immediately! As students pointed out in the Computing CARES video, this means start understanding and planning immediately, but start coding and submitting test files soon. Do not wait until the day before it is due to get started.

Have fun coding!

Libraries and Restrictions

Unless otherwise stated, you are allowed and *encouraged* to use all parts of the C++ STL and the other standard header files for this project. You are **NOT** allowed to use other libraries (eg: boost, pthread, etc). You are **NOT** allowed to use the C++17 regular expressions library or the thread/atomics libraries (they may spoil runtime measurements).

Appendix A: Another Example

The Map

```
# Appendix A Map: 8x8 with 4 islands
M
8
...o$oo.
o..oooo.
...ooo..
........
....oo..
....o...
...oo...
..o...@.
```

```
# Appendix A Map: 8x8 with 4 islands
L
8
0 3 o
0 4 $
0 5 o
0 6 o
1 3 o
1 4 o
1 5 o
1 6 o
2 3 o
2 4 o
2 5 o

1 0 o

4 4 o
4 5 o
5 4 o
6 3 o
6 4 o

7 2 o

7 6 @
```

The Hunts

Hunt 1: with verbose, stats, and treasure map (output found in `appA.hunt1-vspM.sol.txt`).

```
$ ./hunt -svp M < appA.map.txt
Treasure hunt started at: 7,6
Went ashore at: 6,4
Searching island... party returned with no treasure.
Went ashore at: 7,2
Searching island... party returned with no treasure.
Went ashore at: 2,5
Searching island... party found treasure at 0,4.
--- STATS ---
```

```
Starting location: 7,6
Water locations investigated: 13
Land locations investigated: 11
Went ashore: 3
Path length: 13
Treasure location: 0,4
--- STATS ---
...oX+o.
o..oo|o.
...oo|..
.....+-+
....oo.|
....o+-+
...oo|..
..o..+@.
Treasure found at 0,4 with path length 13.
```

Hunt 2: with Captain using a queue, verbose, stats, and treasure map (output found in `appA.hunt2-vspMcQ.sol.txt`).

```
$ ./hunt -p M --stats -v -c QUEUE < appA.map.txt
Treasure hunt started at: 7,6
Went ashore at: 6,4
Searching island... party returned with no treasure.
Went ashore at: 7,2
Searching island... party returned with no treasure.
Went ashore at: 1,6
Searching island... party found treasure at 0,4.
--- STATS ---
Starting location: 7,6
Water locations investigated: 15
Land locations investigated: 11
Went ashore: 3
Path length: 9
Treasure location: 0,4
--- STATS ---
...oX-+.
o..ooo|.
...ooo|.
......|.
....oo|.
....o.|.
...oo.|.
..o...@.
Treasure found at 0,4 with path length 9.
```

Hunt 3: with First Mate using a stack, verbose, stats, and treasure map (output found in `appA.hunt3-vspMfS.sol.txt`).

```
$ ./hunt -v -sp M < appA.map.txt --first-mate STACK
Treasure hunt started at: 7,6
Went ashore at: 6,4
Searching island... party returned with no treasure.
Went ashore at: 7,2
Searching island... party returned with no treasure.
Went ashore at: 2,5
Searching island... party found treasure at 0,4.
--- STATS ---
Starting location: 7,6
Water locations investigated: 13
Land locations investigated: 12
Went ashore: 3
Path length: 15
Treasure location: 0,4
--- STATS ---
...+Xoo.
o..|ooo.
...+-+..
.....+-+
....oo.|
....o+-+
...oo|..
```

```
..o..+@.
Treasure found at 0,4 with path length 15.
```

Hunt 4: with search order "SWEN" and coordinate list (output found in `appA.hunt4-pLoSWEN.sol.txt`).

```
$ ./hunt -o SWEN --show-path L < appA.lst.txt
Sail:
7,6
6,6
5,6
4,6
3,6
2,6
Search:
2,5
2,4
1,4
0,4
Treasure found at 0,4 with path length 9.
```

Appendix B: Tips

If your code "works" when you don't compile with -O3 and breaks when you do, it means you have a bug in your code! Even if you think that your program is always working correctly, and especially when there are problems, you should use `valgrind` to check for errors. For example:

```
$ make debug
$ valgrind ./hunt_debug -v < spec.map.txt
```

Appendix C: Test Case Legend

Each test case will be labeled like

```
<map_size><map_index>-<output_option(s)>
```

or

```
<'spec'|'appA'>-<output_option(s)>
```

or

```
<'INV'|'H'><map_index>
```

- map_size is one of 'S', 'M', or 'L', to give a general idea of the time and memory required during the hunt
- map_index is a two-digit, zero-based number that denotes either different input files, different command line options, or both
- output_options is a collection of zero or more characters (`v` , `s` , and `p`) representing CLI options above, where the order provided in the test case label need not match the order in the command line, and the options provided at the command line may be in short or long format
- spec & appA are examples given in this document
- INV are test cases that provide illegal command line options
- 'H' are handwritten test cases that pose unique challenges

Examples

- S00-vspL: A small test that requires verbose output, stats, and a path displayed in coordinate list format, in addtion to "Treasure Hunt Results"
- M04-v: A medium test that requires verbose output in addition to results
- M09-pM: A medium test that requires the path in treasure map format, in addition to results
- L00: A large test that requires only results

Appendix V: The Example Video

A video with a small example that uses a stack and queue, handles "dead ends" while both sailing and searching, ignoring impassable terrain, multiple trips ashore, and an ultimately successful treasure hunt.

Video on YouTube