THE UNIVERSITY OF SYDNEY

INFO1910 S2 2023 Assignment 1

Due: Sunday the 17th of September, 11:59pm AEST

This assignment is worth 10%% of your final assessment

Task Description

In this assignment you must construct an asteroid defence system for Flatland.

Flatland asteroids follow paths dictated by polynomials of a single variable. Radar systems in Flatland only return the distance to the nearest object (up to some maximum range), they do not return the bearing. You are in charge of an array of radar dishes (scanners) and must predict the future positions of asteroids.

Once you have guessed the future position of the asteroid, you are to report the coordinates and a team of miners will be sent up to divert its path. If the asteroid makes it to the ground then it is a game over.

Structure

The directory structure will take the following form:

- **Makefile** - Your makefile
- **lib** - A directory containing your header files, this may contain subdirectories if needed
- **src** - A directory containing your source files, this may contain subdirectories if needed
- **build** - A directory containing your object files as you compile them
- **test** - A directory containing test cases, you may add your own test cases here

You can find a template at:
git.edstem.org:challenge/96747/assignment-1-flatland
Modifications to the template for bugfixes and testcases will be pushed to:
git.edstem.org:challenge/96747/scaffold/assignment-1-flatland-scaffold

Task Information

The task takes place on a 2D x-y plane. Flatland is the infinite area for $y \leq 0$, while asteroids initially exist at $y > 0$. Time moves by increments of one arbitrary unit dt. Performing a scan takes $1dt$, sending a mission takes $1dt$. Scans have a maximum range of 1000 arbitrary units from the position of the scanner. Updates to the positions of the asteroids occur **before** the scan or intercept action is performed.

Your primary goal in this assignment is to write a `flatland_protect` function that will successfully clear a cluster of asteroids. This function may only call the scanner, and attempt to intercept the cluster of asteroids.

The problem is subdivided across a number of files, each of which deal with one aspect of the problem. These are the polynomials that describe the path of the asteroids, the individual asteroids, the cluster of asteroids and the scanner. You may create any other source and header files that you deem necessary for this task.

We will be using the term 'object' here in a loose fashion to describe a collection of bytes and the functions that act on them. This may be distinct from strict definitions of 'objects' in various languages. Any object that is created with a number of bytes defined by a macro will be created on a stack frame above the `flatland_protect` function.

Polynomials

You are guaranteed that the path of each asteroid follows some function of the form

$$x(t) = \sum_{i}^{i_{\max}} a_i t^i \qquad\qquad y(t) = \sum_{j}^{j\max} b_j t^j$$

Where t is an integer, $i_{\max}$ and $j_{\max}$ are compile time constants and the maximum distance travelled by any asteroid between t and $t+1$ is less than 100. For each asteroid cluster t always begins at 0. Updates to the position of the polynomial occur *before* any other action takes place. As a result the minimum t that can be detected or intercepted is $t = 1$.

Polynomials are defined by two functions; one to store the required data for a polynomial at a specified location in memory, the other to evaluate a polynomial at a given time t. The template for these functions can be found in the `polynomial.c` file. Additionally the number of bytes required to create a polynomial of n elements should be specified by the `SIZEOF_POLYNOMIAL(n)` macro, this ensures that all arrays sizes are compile time constants.

All polynomials will start with $y \geq 1000$. All asteroids will spend a reasonable amount of time between $y < 1000$ and $y = 0$. We additionally guarantee that $|a_i| > |a_{i+1}|$ and $|b_j| > |b_{j+1}|$. No test cases will be sending pathologically fast, impossible to detect asteroids.

The polynomial array is defined from highest to lowest order of coefficients. So for a three element array `[a_2, a_1, a_0]` this would be associated with the polynomial $a_2t^2 + a_1t + a_0$.

Asteroids

An asteroid is defined in terms of two polynomial objects.

To hit an asteroid you will need to provide a point in space that is less than the `tolerance` value away from the position of the asteroid. Different clusters may have different tolerances.

Test cases will be weighted such that there are a more asteroids with low order polynomial paths (much easier to predict) and fewer asteroids with difficult paths.

Asteroids require the following functions `asteroid_create` which takes a pointer and the data needed to construct an asteroid. The `asteroid_intercept` function attempts to intercept an asteroid. `asteroid_impact` which checks if an asteroid has impacted with flatland. Lastly, `asteroid_distance` which checks the distance from the position of a scanner to an asteroid and `asteroid_update` which updates the position of an asteroid.

Clusters of Asteroids

Each asteroid inhabits a 'cluster' of one or more asteroids. Other objects in this problem cannot interact with asteroids individually, they may only act on the cluster.

A cluster of asteroids requires an `asteroid_cluster_create` function, that will be called to set up an initially empty cluster. The number of bytes required to create a cluster for n asteroids must be specified in the macro `SIZEOF_ASTEROID_CLUSTER(n)`. Asteroids may then be added to the cluster using `asteroid_cluster_add_asteroid`.

`asteroid_cluster_update` updates the positions of each asteroid in the cluster. Extending from the asteroid object `asteroid_cluster_intercept` attempts to intercept an asteroid in the cluster at an x, y position. `asteroid_cluster_scan` performs a scan on the cluster from a scanner at an x, y position.

`asteroid_cluster_clear` determines whether all asteroids in the cluster have been intercepted, while `asteroid_cluster_impact`checks if an asteroid has impacted at $y \leq 0$. Lastly, the `get_tolerance` function returns the tolerance of that cluster.

Scanners

A scanner is defined by a point on the line $y = 0$, each scanner reports the distance from it to the asteroid closest to that scanner. If no asteroids are within 1000 of the scanner then it returns INF if an asteroid has already hit flatland, then the scanner returns NaN.

The scanners will be provided in order from lowest x position to highest.

The `SIZEOF_SCANNER(n)` macro will be invoked when allocating memory for the scanner array, n should be the number of bytes you believe that n scanners will require. Scanners are simple objects that may only be created, or used to perform a scan. You should pay careful attention to how to handle the results of a scan while avoiding segmentation faults.

Constraints

As a constraint for this assignment; you are not permitted to use any libraries except for `stdio.h`. You may use other libraries when writing your tests. You may opt to use the `math.h` library but will have your maximum mark for this assessment capped at 75%. You are also not permitted to use structs or unions for this task.

Dynamic memory and variable length arrays (VLAs) are also expressly prohibited. Unless otherwise stated, breaking these conditions will result in a mark of zero for your correctness and performance testing.

Makefile

A template Makefile has been provided with the following rules. Some of these are essential and should not be modified.

- **all** - Currently runs the test rule
- **tests** - Compiles all object files and tests
- **build** - Creates the build directory, useful to set as a dependency
- **build/tests** - Creates the directory for storing built test files
- **clean** - Cleans up any object files and removes the build directory
- **debug** - Compiles a version of the code with debug messages enabled (optional, but useful)

In addition you may define any other rules that you find useful.

Object Files

It is essential that the object files associated with the objects described above are located in the `build` directory, otherwise the test cases may not be able to link to the rest of your code. Otherwise, so long as the sections of the make file related to testing are left alone, you are free to manage these directories as you see fit.

Marking Details

This assignment will subjected to both auto marking and manual marking.

- **Makefile** - 10% : You will be awarded marks for the correct performance of your makefile.
- **Style** - 20% : Your solution will be hand marked for readability and consistency of style
- **Correctness** - 30% : Your solution will be run against a number of test cases to ensure the correctness of each component.
- **Test Cases** - 10% : Marks will be allocated for a reasonable suite of test cases that you write for your program.
- **Performance** - 30% : Your solution will be tested against benchmarks involving single and multiple asteroids. Some tests may have a variable mark assigned base on the number of moves it took to clear the cluster.

While some test cases will be available before the due date, a number will only be run after you have completed your final submission.

Style

Style marks will include meaningful variable names, comments, spacing and defensive programming practices.

This will also include the cleanliness of your submission; the submission should only consist of header files, source files and your make file, no object files! Style marks will also include the appropriate use of header and source files. Use of global variables guarantees a 0 on the style section.

Please consult the INFO1910 C style guide for more details.

Submission and Testing

Your code will be submitted to Ed using git. Ed will perform the marking for the correctness component.

Testing will be performed by running the make command, then linking your code to the test cases. It is imperative that functions and macros maintain their correct names. Beyond this the rest of the directory structure is yours to do with as you wish.

Testing will be conducted by running `make` and then compiling our our own tests against the your object files. You can see an example of how this might occur in the scaffold.

Where to Start?

Here are some tips for how to approach this assignment:

- Get the sample code compiling, this will help you design your makefile, try to build to the build directory to make cleaning up easier. Links in previous tutorials will help here.
- It's easy to keep your git repo clean with a .gitignore, look into this.
- Start with the polynomials, they do not depend on any other code.
- Consider that constraints on the maths library do not prevent you from implementing your own maths functions. You may want to consult Wikipedia for how to write your own simple mathematical functions. You only need to be accurate up to some relatively generous tolerance, so exact implementations are not required.
- Get each of the different components working before worrying about bringing everything together.
- Test each of your components independently and write your own tests. This will also greatly simplify picking up the testing marks.
- Predicting the future position of the asteroid is the first open problem of this assignment. You'll need to come up with your own strategy here. Attempting to exactly determine the function of the asteroid is not necessarily the best approach. If you implemented your own `sqrt` function this might give you an idea about how to approach this problem.
- You do not need any physics knowledge for this task - real objects do not follow arbitrary polynomials.
- Do not worry about the multiple asteroid cases until you have everything else working.
- Manage your time well, you almost certainly cannot do this task at the last minute. You will need time to let your thoughts percolate.

Feel free to ask questions (though don't post your solution), and remember that it is generally better to ask sooner rather than later!

Academic declaration

By submitting this assignment you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.