Assignment 1

Due: September 27th, 2015 before 11:59pm

Objectives

- Review structures, loops and functions
- Introduction to input and output in C++
- Practice reading a specification and implementing a specification
- Introduction to Scalable Vector Graphics (SVG)
- Introduction to software testing and debugging techniques

Introduction

In this assignment you will implement a simplified version of Turtle Graphics. More information about Turtle Graphics can be found here: https://en.wikipedia.org/wiki/Turtle_graphics

A turtle has three basic attributes:

- a location
- an orientation
- a pen, which is either up or down

In our implementation we will generate Scalable Vector Graphics (SVG) as output. SVG files can be viewed with many image programs and most web browsers.

Quick Start:

1. Read this entire document
2. Download all the provided files into a new directory
3. Read `turtle.h` carefully
4. Compile and run the test program `part1_tester.cpp`
5. If no errors reported by test program move on to Part 2
6. Implement one of the functions in `turtle.cpp`
7. goto 4

Multi-file Programs

Most non-trivial programs contain more than one source code file.

By convention, `.h` files contain function prototypes and type definitions[1] and the implementation of the functions is contained in a `.cpp` file of the same name.

While there are more complicated tools for dealing with building multi-file programs (like Make, scons, Eclipse projects etc), the simplest way to build a multi-file program is to specify all the `.cpp` files on the command line to the compiler.

For example:

```
% g++ -Wall part1_tester.cpp turtle.cpp svg_file.cpp -o part1
```

Will compile all the files used for Part 1 and create an executable called `part1_test` (`part1_test.exe` on Windows), which can be run to see if your implementation passes the test cases.

Run the resulting program by typing:

```
% part1
```

The output you should see is:

```
Testing turtle_init:

Test 0 failed at line: 42 in file: part1_tester.cpp


0 tests passed.
```

Because you haven't implemented anything yet, none of the test cases will pass.

1 C++ puts no restriction on what goes in .h files.

Part 1

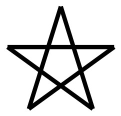
In Part 1, you are implementing a set of functions that implement turtle graphics. The only code you have to write for Part 1 is in the file `turtle.cpp`.

The specifications for what you are to write in Part 1 are fully contained in the file `turtle.h`. Be sure to read this specification carefully.

You've been supplied with a test program that will give you feedback on your progress on Part 1. When you've finished Part 1, the output of the test program will be:

```
Testing turtle_init:
Test 0 passed.
Test 1 passed.

Testing turtle_pen_up and turtle_pen_down:
Test 2 passed.
Test 3 passed.

Testing turtle_rotate_left and turtle_rotate_right:
Test 4 passed.
Test 5 passed.
Test 6 passed.
Test 7 passed.

Testing turtle_move_forward and rounding:
Test 8 passed.

Testing svg output.  You need to validate this test by hand.
View the file part1.svg.  It should contain a star.

9 tests passed.
```

In the directory you ran Part 1, you should find a file named “`part1.svg`”. If you open this file in a web browser, you should see the following image:



Compiling and running Part 1:

To compile Part 1, type:

```
% g++ -Wall part1_tester.cpp svg_file.cpp turtle.cpp -o part1
```

To run Part 1 type:

```
% part1
```

Part 2

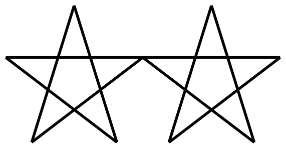
In Part 2, you will use what you completed in Part 1 to write a program that reads a text file and that text file will describe actions the turtle will take. This will allow us to draw interesting images without having to recompile. Most real programs use files as input in this way.

See the section below on "Turtle Input Files" for details on the content of the input files. Note that in Part 2, you do not have to implement the `repeat` command.

When you've finished Part 2, and you run it like this:

```
% part2 ex1.txt
```

Your program will create a file named `ex1.txt.svg` which contains the following image:



To compile Part 2, type:

```
% g++ -Wall part2.cpp svg_file.cpp turtle.cpp -o part2
```

Part 3

When you have finished Part 2, copy your `part2.cpp` file to a new file named `part3.cpp`.

In Part 3, you will implement the `repeat` command. Again see the section on "Turtle Input Files" below for details on the `repeat` command.

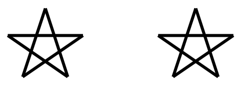
In Part 3, `repeat` commands cannot be nested. That is, you can't have a `repeat` command inside a `repeat` command.

For Part 3, you need to create your own input file that draws a unique image. You should create this in the file `creation.txt`. You need to submit your creation and the resulting .svg file produced by your Part 3.

When you've finished Part 3, and you run it like this:

```
% part3 ex2.txt
```

Your program will create a file named ex2.txt.svg which contains the following image:



To compile Part 3, type:

```
% g++ -Wall part3.cpp svg_file.cpp turtle.cpp -o part3
```

Bonus

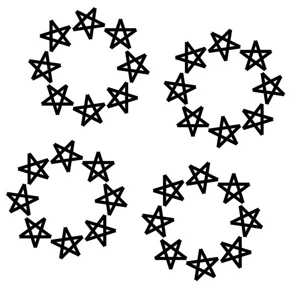
Copy your `part3.cpp` file to `bonus.cpp`.

For the bonus, you should change your code so that it handles nested loops. To keep it simple, you may assume that loops will be nested no more than 100 deep.

When you've finished your bonus and you run it like:

```
% bonus bonus.txt
```

Your program will create a file named `bonus.txt.svg` which contains the following image:



To compile the bonus, type:

```
% g++ -Wall bonus.cpp svg_file.cpp turtle.cpp -o bonus
```

Turtle Input files

Part 2 and Part 3 of the assignment take text files as input to control the turtle.

You may assume that any text files used for testing follow these specifications exactly. Your program is not responsible for detecting errors in the input files. The result of providing an invalid input file to your program is undefined.

You are not responsible for handling the case where the input file causes the turtle to move outside the boundaries of the canvas.

The properties of the text file is as follows:

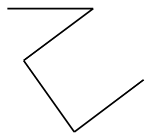
- Each line of the input file contains exactly one command
- There are no blank lines in the input file
- Where a command contains more than one item on the line, the items are separated by exactly one space

The possible commands that may be included are:

Command	Parameters	Desciption	Example
pen	up or down	Set the pen to be either up or down	`pen up`
right	an integer between 0 and 360	Rotate the turtle to the right	`right 180`
left	an integer between 0 and 360	Rotate the turtle to the left	`left 90`
forward	a positive integer	Move the turtle forward	`forward 100`
repeat	a positive integer	Repeat a set of commands	*See below*

An example input file and the output it produces:

```
pen down
forward 100
right 144
forward 100
left 90
forward 100
left 90
forward 100
```



Loops

Loops provide the ability for commands to be repeated, which allows for more complex pictures to be generated.

The loop command itself takes up three lines, and has the following format:

```
repeat 4
{
}
```

This loop will repeat the commands four (4) times.

The commands to be repeated are placed on new lines between the { and the }.

A reminder that each command must be on its own line and that commands inside the loop cannot be indented.

Loops are only implemented in Part 3 and the Bonus parts of the assignment. For Part 3, loops cannot be nested. For bonus marks, loops can be nested.

Submission

Submit your `turtle.cpp, part2.cpp, part3.cpp, creation.txt and creation.svg` files using Connex.

If you completed the bonus, submit your `bonus.cpp` file.

A reminder that it is OK to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution.

We will be using plagiarism detection software on your assignment submissions.

Grading

If you submit something that does not compile, you will receive a grade of 0 for the assignment. It is your responsibility to make sure you submit the correct files.

Requirement	**Marks**
You submit something that compiles	2
Part 1: Your code passes the part1_tester test cases	Up to 10
Part 2: Your code generates the correct SVG for test case similar to `ex1.txt`	2
Part 3: Your code generates the correct SVG for test case similar to `ex2.txt`	2
You submit a unique text (named `creation.txt`) file and resulting image file (named `creation.txt.svg`) where the `creation.txt` file creates the `creation.txt.svg` file using your Part 3*	2
Bonus: Your code generates the correct SVG for test case similar to `bonus.txt`	2

Total 18**

* if you complete the bonus, submit a text file and image generated using your bonus implementation

** with bonus marks, it is possible to score 20/18 on this assignment