ITP 365: Managing Data in C++

Homework 1: Sieve of Eratosthenes Due: 9/8/2017 @ 11:59PM

Goal

You will write a graphical implementation of the classic Sieve of Eratosthenes algorithm to determine prime numbers. This will provide experience using C++, Vectors, and the Stanford C++ Library.

Lab Setup

- Make sure you have followed the videos to install either Visual Studio or Xcode.
- Follow these video instructions to get started on the homework:
  - Windows: http://youtu.be/xd1WC7FyVVM
  - Mac: http://youtu.be/IY8J-P-Zyes
- You will notice that the project already has a **main.cpp** file with almost no code in it. For this project, you will need to create two additional files:
  - **sieve.h** – This will contain an enum declaration as well as function prototypes
  - **sieve.cpp** – This will contain function implementations
- At the top of every file make sure you add comments in the following format (using your name, email, and specify whether you did your homework on Windows or Mac):
  ```
  // Name: Tommy Trojan
  // Email: ttrojan@usc.edu
  // ITP 365 Spring 2017
  // HW1 – Sieve of Eratosthenes
  // Platform: Windows
  ```
- If you have trouble getting up and running, I recommend posting on Piazza as soon as you encounter the issue so that we can help!
- The lab instructions on the subsequent pages are broken down into parts so that you can incrementally test your code. It is strongly recommended to implement the code for each part in order.
- ***Note that since this is the first homework assignment, I have provided a detailed breakdown of how I want you to implement it (what functions, what they should do, and so on). This will not necessarily be the case for future assignments.***

Part 1: NumberType enum

✓
- Create an enum in sieve.h called `NumberType` that has three values: `UNKNOWN`, `PRIME`, and `COMPOSITE`

Part 2: drawSquare function

✓
- Create a function called `drawSquare` that draws a single 50x50 square (later, we will call this function multiple times to draw multiple squares). Declare the prototype in **sieve.h** and implement it in **sieve.cpp**.

✓
- `drawSquare` returns void

- ✓ `drawSquare` takes the following parameters:
  - A `GWindow` (by reference) – the window in which to draw the square
  - An `int` called `number` (by value) – the number to display in the center of the square
  - A `NumberType` (by value) – tells what type of number this square corresponds to. This determines what color the square should be filled in with. `UNKNOWN` should be white, `PRIME` should be green, and `COMPOSITE` should be red.
  - A `double` called `x` (by value) – the x coordinate of the top-left corner of the square
  - A `double` called `y` (by value) – the y coordinate of the top-left corner of the square
- ✓ You will need to use several GWindow member functions in `drawSquare`. Specifically:
  - `setColor` to set the current color
  - `fillRect` to draw a filled rectangle
  - `drawRect` to draw an outline of a rectangle
  - `drawLabel` to draw a `std::string` (remember this is a function I added, so check Week 1, Lecture 2 for the syntax if you don't remember it)
- As for how `drawSquare` should be implemented, I'd suggest the following algorithm:
  1. Draw the filled rectangle, based on `NumberType`
  2. Draw the outline of a rectangle (always in black)
  3. Draw the number label (also in black)
- Make sure you test your draw square function by calling it with different parameters (from main), and make sure the results are as you expect.

Part 3: initVectors function

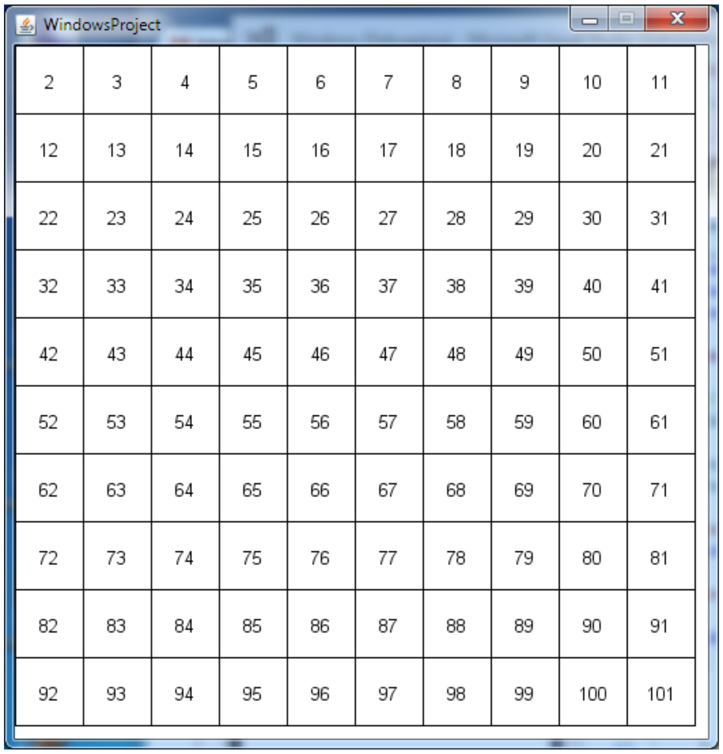
- ✓ In **main.cpp**, add an include for **vector.h**, which is the StanfordCPP implementation of the vector data structure
- ✓ Declare two `Vector` instances in `main`. One should be a vector of `int`s (that will store all of the numbers we are testing if they are prime), and the other a vector of `NumberType`s (that we will use to flag which numbers are composite or prime).
- ✓ In **sieve.h/cpp**, create a function called `initVectors` that will initialize these two vectors to their initial states.
- ✓ `initVectors` returns `void`
- ✓ `initVectors` takes the following parameters:
  - A `Vector` of `int`s (by reference)
  - A `Vector` of `NumberType`s (by reference)
- ✓ This function should then use the Vector member functions to add elements to the two vectors. The vector of `int`s should store every number from 2 up to and including 101 (sequentially). The vector of `NumberType`s should store 100 elements that are all initially `UNKNOWN` (since you haven't determine the primes yet)
- ? Call `initVectors` in `main`

Part 4: drawGrid function

- ✓ Create a function called `drawGrid`, which is responsible for drawing the current state of the grid. Declare the prototype in **sieve.h** and implement it in **sieve.cpp**.
- ✓ `drawGrid` returns `void`

✓

- `drawGrid` takes the following parameters:
    - A `GWindow` (by reference) – the window in which to draw the grid
    - A `Vector` of `ints` (by reference) – these are the numbers in the sieve
    - A `Vector` of `NumberTypes` (by reference) – these are the types of the numbers in the sieve
- Draw grid should call `drawSquare` once for each element in the vectors. There are multiple ways to implement this behavior. Here is one idea:
    - Start at x = 0 and y = 0
    - For every index from 0 to 99…
        - Call drawSquare, passing in the number and type for the current index, as well as the current x/y
        - Increase x by 50. If x is greater than 450, this means you are at the end of the row. In this case, set x back to 0 and instead increase the y by 50
- Call `drawGrid` in main, after the call to `initVectors`.
- If this was implemented properly, you should end up with a grid of numbers that looks something like this:



2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41
42	43	44	45	46	47	48	49	50	51
52	53	54	55	56	57	58	59	60	61
62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81
82	83	84	85	86	87	88	89	90	91
92	93	94	95	96	97	98	99	100	101

Part 5: calcNextPrime function

- Create a function called `calcNextPrime`, which will find the next prime number and mark its multiples as composite.
- `calcNextPrime` returns an `int`, which corresponds to the prime number it found. If there are no new prime numbers found, it should return `-1`.
- `calcNextPrime` has the following parameters:
    - A `Vector` of `ints` (by reference) – the numbers in the sieve

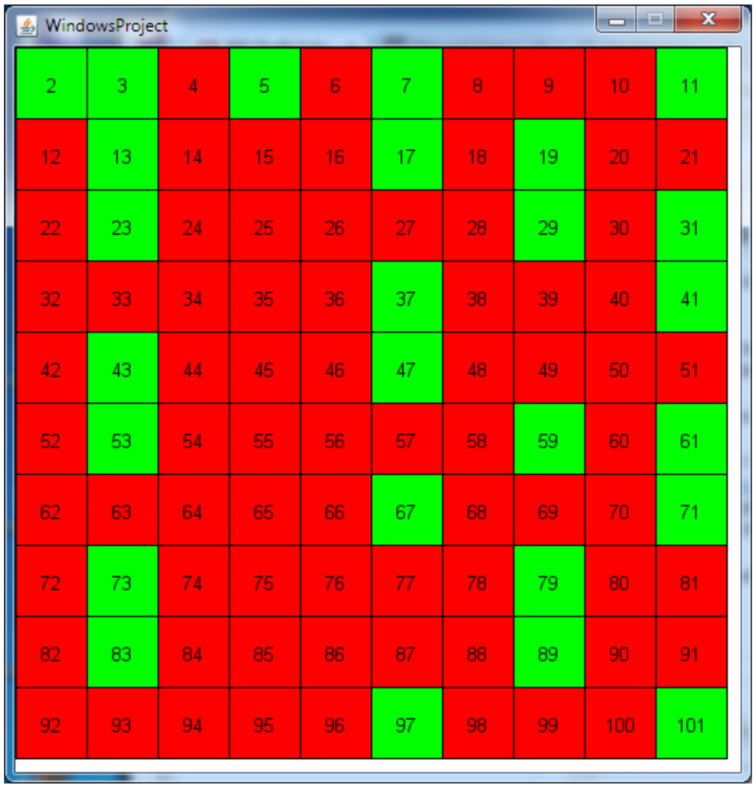
    - A `Vector` of `NumberTypes` (by reference) – the types of the numbers in the sieve
    - An `int` called `startAt` (by value) – the number to start testing at
- `calcNextPrime` should implement the following algorithm:
    1. Starting at the index corresponding to the value of `startAt`, find the first index that is marked as `UNKNOWN`. The number at this index is guaranteed to be prime. Don't forget that a number isn't the same as its index (it's off by 2). If you reach the end of the vector and there are no `UNKNOWN` numbers left, it means `calcNextPrime` should return `-1` to signify it's done.
    2. Once you find the next prime number, every multiple of that prime number should be set to `COMPOSITE`. So for example, when the number 2 is "discovered" as prime, that means the indices that contain the numbers 4, 6, 8, etc. should all be set as `COMPOSITE`.
    3. Return the prime number that was found (or `-1` if there are no prime numbers left).
- You should then call this function in `main`, inside a loop.
    - Keep looping until there are no new prime numbers left (eg. the function returns `-1`).
    - In each iteration, you should also call `drawGrid` (to update the displayed grid) and `pause(1000.0)` in order to pause for one second between iterations.
- Your final output should look like the sample output below.

A Note on Style

Style will be a small portion of the overall homework grade (5%). If you write spaghetti code, don't provide comments for function declarations, and so on, you will lose some points. We will similarly make sure you use the separate files correctly.

Sample Output

When the sieve finishes its animation, it should look something like this:



🟩 2	🟩 3	🟥 4	🟩 5	🟥 6	🟩 7	🟥 8	🟥 9	🟥 10	🟩 11
🟥 12	🟩 13	🟥 14	🟥 15	🟥 16	🟩 17	🟥 18	🟩 19	🟥 20	🟥 21
🟥 22	🟩 23	🟥 24	🟥 25	🟥 26	🟥 27	🟥 28	🟩 29	🟥 30	🟩 31
🟥 32	🟥 33	🟥 34	🟥 35	🟥 36	🟩 37	🟥 38	🟥 39	🟥 40	🟩 41
🟥 42	🟩 43	🟥 44	🟥 45	🟥 46	🟩 47	🟥 48	🟥 49	🟥 50	🟥 51
🟥 52	🟩 53	🟥 54	🟥 55	🟥 56	🟥 57	🟥 58	🟩 59	🟥 60	🟩 61
🟥 62	🟥 63	🟥 64	🟥 65	🟥 66	🟩 67	🟥 68	🟥 69	🟥 70	🟩 71
🟥 72	🟩 73	🟥 74	🟥 75	🟥 76	🟥 77	🟥 78	🟩 79	🟥 80	🟥 81
🟥 82	🟩 83	🟥 84	🟥 85	🟥 86	🟥 87	🟥 88	🟩 89	🟥 90	🟥 91
🟥 92	🟥 93	🟥 94	🟥 95	🟥 96	🟩 97	🟥 98	🟥 99	🟥 100	🟩 101

Submission

1. Create a ZIP file for Blackboard called **HW1Submit.zip** containing *only* the following files:
    - main.cpp
    - sieve.cpp
    - sieve.h
2. Submit the ZIP file using the designated Homework 1 submission link in the “Assignments” section.

Grading

Item	**Points**
Part 1: NumberType Enum	5
Part 2: drawSquare	15
Part 3: initVectors	10
Part 4: drawGrid	15
Part 5: calcNextPrime	50
Comments, style, and correct use of separate files	5
Total	**100**