Homework #3

Released: 01-24-2017
Due: 01-31-2017 11:59pm

In this homework, we are going to turn our effort in homework 2 into two tiny libraries, the `circle` library and the `prime` library. Both libraries comes with three files and a main program:

- `circle_lib.h` contains the declarations of the APIs of the `circle` library. These include `read_circle()` and `overlapped(c1, c2)`.
- `circle_lib.cpp` contains the actual implementation of the `circle` APIs.
- `circle_test.cpp` contains the unit tests of the `circle` library.
- `circle.cpp` is the main program that uses the `circle` library to perform computation. In other words, this program is the "client" of the `circle` library.

Similarly,

- `prime_lib.h` contains the declarations of the APIs of the `prime` library. These include `is_prime(p)`, `generate_primes(n)` and `check_is_prime(p)`.
- `prime_lib.cpp` contains the actual implementation of the `prime` APIs.
- `prime_test.cpp` contains the unit tests of the `prime` library.
- `prime.cpp` is the client of the `prime` library.

To start this homework, first fill in the implementation of the functions `read_circle`, `overlapped` and `is_prime` using your homework 2 solution. Also, complete the stub `main` function in `circle.cpp` by using your code from homework 2 so that `circle.cpp` will run the same as in homework 2 except that it invokes the `circle` library.

In the following sections, we are going to extend the `prime` library to implement Eratosthenes sieve algorithm, improve the APIs to handle incorrect arguments and write unit tests for these two libraries.

1 Find Primes, Again

1.1 Generate All Primes

Implement the function `vector<int> generate_primes(int n)` that uses the sieve of Eratosthenes to generate all primes between 2 and `n`. The concept of the sieve of Eratosthenes is to identify all composite numbers by marking the multiples of prime numbers on a table instead of using trial division to find factors. First, the algorithm will create a big table containing every integers between 2 and `n`. Then, for the first number `p` that is not crossed out, record it as prime and cross all its multiples out from the table.

Every composite number must have a prime factor that is smaller than itself. Thus, all composite numbers would have been crossed out by that prime factor before we examine it. Therefore whenever we see a number that is not crossed out from the table, it must be a prime.

To implement this,

1. Declare a vector of booleans, `vector<bool> sieve;`, to represent the big table. The size of `sieve` should be `n+1` so that we can use `sieve[i]` to represent whether `i` is crossed out from the table or not.
2. Mark `sieve[i]` as `true` for `i` between 2 and `n`.
3. Loop from 2 to `n`. Whenever we encounter a number `i` such that `sieve[i] == true`, mark `sieve[i*2]`, `sieve[i*3]`, `sieve[i*4]`, ... as `false`.
4. Now, all `i` such that `sieve[i] == true` are prime numbers.

1.2 Test Primality, Again

Implement the function `bool check_is_prime(const std::vector<int>& primes, int p)` that uses trial division over primes to test whether `p` is a prime. So instead of dividing `p` by all integers between 2 and `p` − 1, this function should only check the primes between 2 and `p` − 1.

1.3 The `main` Function

`prime.cpp` contains a provided `main` function that calls the `prime` library. First, `main` will read a positive integer `n` and use `generate_prime` to generate all primes up to `n`. Then, for every subsequent positive integer `p`, `main` will invoke `check_is_prime` with the generated primes and `p` to check whether `p` is a prime and print the corresponding string. `main` will terminate upon reading a zero.

Examples

1

When given the input

```
10
2
3
4
96
97
98
0
```

`./prime` should print

```
yes
yes
no
no
yes
no
```

2

When given the input

```
1000000
1000000007
1000000009
1000000011
2147483643
2147483647
0
```

`./prime` should print

```
yes
yes
no
no
yes
```

2 Handle Errors

In this section, we are going to improve the APIs by checking whether their arguments are valid or not. For simplicity, if any of the arguments is not reasonable, throw a `runtime_error` with the error message you like (which, of course, should be informative and readable).

- Modify `bool overlapped(Circle c1, Circle c2);` to ensure that both `c1` and `c2` have non-negative radius. If not, throw a `runtime_error`.
- Modify `bool is_prime(int p);`, `bool check_is_prime(const vector<int>& primes, int p);` and `vector<int> generate_primes(int n);` to ensure that they are only called with positive integers (i.e. $\texttt{p}, \texttt{n} \geq 1$). If not, throw a `runtime_error`.
- Modify `bool check_is_prime(const vector<int>& primes, int p);` to ensure that `primes` is non-empty. If not, throw a `runtime_error`.

3 Write Unit Tests

In this section, please improve the quality of the two libraries by writing appropriate unit tests to check that the APIs work well under specified conditions.

3.1 `bool overlapped(Circle c1, Circle c2);`

- Write unit tests for this function. Also do check that this function does not treat tangent circles as overlapping.
- Write unit tests to check that this function does throw an exception for erroneous arguments.

3.2 `bool is_prime(int p);`

- Write unit tests to check that this function works for $1 \leq \texttt{p} \leq 5$.
- Write unit tests to check that this function does throw an exception for erroneous arguments.

3.3 `vector<int> generate_primes(int n);`

- Write unit tests to check that this function works for some `n` which is not a prime.
- Write unit tests to check that this function works for some `n` which is a prime.
- Write unit tests to check that this function does throw an exception for erroneous arguments.

3.4 `bool check_is_prime(const vector<int>& primes, int p);`

- Write unit tests to check that this function works for $1 \leq \texttt{p} \leq 5$.
- Write unit tests to check that this function and `is_prime` agrees with each other.
- Write unit tests to check that this function do throw an exception for erroneous arguments.