Lab - Containers, Iterators & Functors

Topics and References

- Standard Template Library Containers
- STL Iterators
- Function Objects, a.k.a. Functors

Learning Outcomes

- Practice using STL containers
- Familiarize with iterators
- Learn to program with function objects

Overview

Prime Number Generation

Cryptocurrencies have been gaining in popularity in recent years. A cornerstone of cryptocurrencies is encryption, and this makes use of prime numbers as crypto keys.

This necessitates the very widespread use of prime numbers whose search for has been automated as follows:

1. Random positive integers are generated
2. The numbers are tested for primality
3. Those that pass the test are used

The first task is on using containers, iterators and function objects to check for and store prime numbers.

Look-up Data

A technique to get demanding applications like computer graphics to work is to pre-compute and store data such as sines for fast look-up when needed. Sines are demanding to calculate accurately from scratch so it really helps to have the values ready to use.

The second task is on utilizing containers in the generation and storage of look-up information. Calculating sines is complicated so for this exercise we calculate powers.

Task 1 - Prime Numbers

In this exercise we follow the above prime discovery process by taking in a collection of numbers and returning the primes. In the process we have to do the following:

1. Check for and remove duplicates to ensure we do not use the same key twice
2. For the sake of orderliness arrange the numbers in descending order

The above behavior must be implemented by the following function template:

```
1 template<typename It>
2 std::vector<unsigned> prime(It begin, It end);
```

where `begin` and `end` are iterators demarking the half open range of numbers to check for primes in. The return value is a container with the prime numbers in descending order. Implement your solution in a file called `q.h`.

Implementation Details

Use the `<algorithm>` header to sort and remove duplicates in the numbers. Implement a function object to make `std::sort` sort in reverse order.

A function object is one that implements `operator()` which is known as the *call operator* or *application operator*. A function object allows us to pass a function encapsulated as an object.

Being an object, we can have the function object maintain state information in its data member(s) if we needed to. This is a main advantage of function objects.

For `sort` to produce values in descending order, we are to pass into `sort` a half open range followed by a function object where the call operator follows the following signature:

```
1 bool operator()(T const& lhs, T const& rhs);
```

and where `operator()` returns `true` when we want `lhs` ordered *before* `rhs`, otherwise it returns `false`.

A number *n* may be tested for primality as follows:

1. We check for a factor of *n* by using a number *f* and setting it to the value of 2, and we test whether or not *f* is a factor of *n*.
2. If it is not, we increment *f* and repeat step 1.
3. This process goes on until *f* is greater than the square root of *n*.

Complete `prime` in `q.h`.

Task 2 - Look-up Data

As a simple example of generating look-up information, implement `pow` to calculate the specified power of input numbers and return the data in the form of a `std::map`. `pow` takes in a `std::array` of numbers whose powers are to be calculated and stored followed by an integer indicating the required power. A `map` is a container that takes in an entry in the form of a `std::pair` through member function `insert` consisting of a search key as the first element and the associated data as the second.

The entry is stored in the `map` in a search tree using the key to set its position in the tree. In the tree information is organized for fast retrieval using the search key.

Study the driver file to see how `pow` is used, what it takes in, and what it returns, and implement `pow` as a template in `q.h`.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file

Submit the completed `q.h` on Moodle.

Source file

There is no `cpp` file to submit.

Compiling, executing, and testing

Create a `makefile` using a `makefile` from a previous task. If your code is correct `diff` will report no differences.

File-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This documentation serves the purpose of providing a reader the raison d'être of this source file at some later point of time (could be days or weeks or months or even years later). This module will use Doxygen to tag source and header files for generating html-based documentation. An introduction to Doxygen and a configuration file is provided on the module web page. Here is a sample for a C++ source file:

```
/*!*****************************************************************************
****
\file    scantext.cpp
\author  Prasanna Ghali
\par     DP email: pghali\@digipen.edu
\par     Course: CSD1170
\par     Section: A
\par     Programming Assignment #6
\date    11-30-2018

\brief
  This program takes strings and performs various tasks on them via several
  separate functions. The functions include:

  - mystrlen
      Calculates the length (in characters) of a given string.

  - count_tabs
      Takes a string and counts the amount of tabs within.

  - substitute_char
      Takes a string and replaces every instance of a certain character with
      another given character.

  - calculate_lengths
      Calculates the length (in characters) of a given string first with
tabs,
      and again after tabs have been converted to a given amount of spaces.

  - count_words
      Takes a string and counts the amount of words inside.
*****************************************************************************
***/
```

Function-level documentation

Every function that you declare, define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. In team-based projects, this information is crucial for every team member to quickly grasp the details necessary to efficiently use, maintain, and debug the function. Certain details that programmers find useful include: what does the function take as input, what is the output, a sample output for some example input data, how the function implements its task, and importantly any special considerations that the author has taken into account in implementing the function. Although beginner programmers might feel that these details are unnecessary and are an overkill for assignments, they have been shown to save considerable time and effort in both academic and professional settings. Humans are prone to quickly forget details and good function-level documentation provides continuity for developers by acting as a repository for information related to the function. Otherwise, the developer will have to unnecessarily invest time in recalling and remembering undocumented details and assumptions each time the function is debugged or extended to incorporate additional features. Here is a sample for function `substitute_char`:

```
/*!*****************************************************************************
****
\brief
   Replaces each instance of a given character in a string with
   other given characters.

\param string
  The string to walk through and replace characters in.

\param old_char
  The original character that will be replaced in the string.

\param new_char
  The character used to replace the old characters

\return
  The number of characters changed in the string.
*****************************************************************************
***/
```

Since you are to submit `q.h` and `q.cpp`, add the documentation specified above to them.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `q.h`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - `F` grade if your `q.h` don't compile with the full suite of `g++` options.

- `F` grade if your `q.h` don't link to create an executable.
    - A proportional grade if your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests).
- `A+` grade if output matches the correct output of the auto grader.
    - Expected deduction of one letter grade for each missing documentation block in `q.h`. Your submission of `q.h` must have **one** file-level documentation block and at least **one** function-level documentation block. Each missing or incomplete or copy-pasted (with

irrelevant information from some previous assessment) block may result in a deduction of a letter grade. For example, if the automatic grader gave your submission an `A+` grade and one documentation block is missing, your grade may be later reduced from `A+` to `B+`. Another example: if the automatic grader gave your submission a `C` grade and the two documentation blocks are missing, your grade may be later reduced from `C` to `E`.