FACULTY OF COMPUTERS AND INFROMATION, CAIRO UNIVERSITY

CS213: Programming II
Year 2022-2023
First Semester

Problem Sheet 2 – Version 1.0

Course Instructors:
Dr. Mohammad El-Ramly

Revision History

Version 1.0	By Dr Mohammed El-Ramly	20 Oct. 2022	Main Doc
Version 2.0	By Dr Mohammed El-Ramly	20 Nov. 2023	Adapted for 23/24

CS213: Object Oriented Programming

Problem Sheet 2 – OOP Modeling and C++ OOP Features

Cairo University, Faculty of Artificial Intelligence

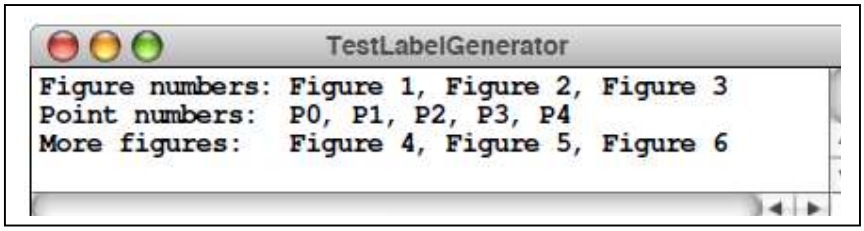
Objectives: This sheet includes programming problems for training on C++ concepts.

Problems

1. **Label Generator.** For certain applications, it is useful to be able to generate a series of names that form a sequential pattern, e.g., you may want to number the figures in a report as "**`Figure 1`**", "**`Figure 2`**", "**`Figure 3`**", and so on. You might also need to label points in a geometric diagram, in which case you would want a similar but independent set of labels for points such as "**`P0`**", "P1", "**`P2`**", and so forth.

   Thinking generally, we need a tool or label generator that allows the client to define arbitrary sequences of labels, each of which consists of a prefix string ("**`Figure `**" or "**`P`**" for the examples in the preceding paragraph) coupled with an integer used as a sequence number. Because the client may want different sequences to be active simultaneously, it makes sense to define the label generator as an abstract type called **`LabelGenerator`**. To initialize a new generator, the client provides the prefix string and the initial index as arguments to the **`LabelGenerator`** constructor. Once the generator has been created, the client can return new labels in the sequence by calling **`nextLabel`** on the **`LabelGenerator`**. As an illustration of how the interface works, the main program shown here followed by the output:

```
int main() {
      LabelGenerator figureNumbers("Figure ", 1);
      LabelGenerator pointNumbers("P", 0);
      cout << "Figure numbers: ";
      for (int i = 0; i < 3; i++) {
            cout << figureNumbers.nextLabel() << ", ";
      }
      cout << endl << "Point numbers: ";
      for (int i = 0; i < 5; i++) {
            cout << pointNumbers.nextLabel() << ", ";
      }
      cout << endl << "More figures: ";
      for (int i = 0; i < 3; i++) {
            cout <<
      figureNumbers.n
      extLabel() <<
      ", ";
      }
      cout << endl;
      return;
}
```



   Separate specifications from implementation.

   Now inherit from this class a new class called **`FileLabelGenerator`** which has an extra attribute that is a file name with lines of captions of labels to use in generation. This will override method **`nextLabel`** to take the next label from` the file and added it to the generated label that you by calling the parent's **`nextLabel.`** So, the following code run and produce the following output:

```
FileLabelGenerator figureLabels ("Figure ", 1, "labels.txt");
cout << "Figure labels: \n";
for (int i = 0; i < 3; i++) {
      cout << figureLabels.nextLabel() << endl;
```


```
Figure 1 Sign vs Cosine Functions.
Figure 2 Sigmoid Function.
Figure 3 …….
```

2. **Document Similarity.** Document similarity measures are very important in the field of information retrieval and search engines. They are measures that tell us how similar to documents are in terms of their word content. They can be used to find similar documents or to find how close a document is to a query on a search engine. There are many of such measures. In this problem, you will develop:

   - Define a class called **`StringSet`** that will store a set of C++ strings. Use an array or a vector to store the strings. Create a constructor that takes a file name and loads the words in it (ignoring punctuation and turning text to lower case). Write another constructor that takes a string and loads it and breaks it to tokens. Write member functions to add a string to the set, remove a string from the set, clear the entire set, return the number of strings in the set, and output all strings in the set. Overload the **`+`** operator to return the union of two **`StringSet`** objects. Overload the **`*`** operator so that it returns the intersection of two **`StringSet`** objects. Write a program to test all functions.
   - Add a member function that computes the similarity between the current **`StringSet`** and an input parameter of type **`StringSet`**. Similarity is measured by ***binary cosine coefficient***. The coefficient is a value between 0 and 1, where 1 indicates that the query (or document) is very similar to the document and 0 indicates that the query has no keywords in common with the document. This approach treats each document as a set of words. For example, given the following sample document:

     "Chocolate ice cream, chocolate milk, and chocolate bars are delicious."

     This document would be parsed into keywords where case is ignored and punctuation discarded and turned into the set containing the words {chocolate, ice, cream, milk, and, bars, are, delicious}. An identical process is performed on the query to turn it into a set of strings. Once we have a query **Q** represented as a set of words and a document **D** represented as a set of words (each word counts once even if repeated in document multiple times), the similarity between **Q** and **D** is computed by:

     $$\textbf{Sim} = \frac{|\mathbf{Q} \cap \mathbf{D}|}{\sqrt{|\mathbf{Q}|}\sqrt{|\mathbf{D}|}}$$

     The size of set of common words / (sqrt size of **D** * sqrt size of **Q**)

     Write a program to test the classes and run sample queries on sample documents.

3. **STL – Map.** Farah Mohamed works remotely with a foreign company that is writing a search engine. One of the criteria for ordering the retrieved documents is how many times a word exists in the document. To find that out, she needs to develop a frequency table for a given text file. A frequency table lists words and the number of times each word appears in a document or a file. Help her write a program that creates a frequency table for a file whose name is entered by the user. Use STL **`map`** that stores <Key, Value> pairs of **`string`** and **`int`**. First, read the input file and clean it of all punctuation and non-alphanumeric characters except "-" which can be part of a word. You may find functions **`ispunct()`**, **`isalnum()`**, **`tolower()`**, etc (see **`cctype`** library) useful to use. Test your program on multiple different files.

4. **Template – Set.** Abdullah Mohamed uses a C++ version that does not support STL and he needs a set class for his programming job. Help him by writing a template-based class that implements a set of items. A set is a collection of items in which no item occurs more than once. Internally, you may represent the set using the data structure of your choice (for example, list, vector, arrays, etc.). However, the class should externally support the following functions:

   - Add a new item to the set. If the item is already in the set then nothing happens.
   - Remove an item from the set.
   - Return the number of items in the set.
   - Determine if an item is a member of the set.
   - Return a pointer to a dynamically created array containing each item in the set. The caller of this function is responsible for de-allocating the memory.

   Test your class by creating different sets of different data types (for example, strings, integers, or other classes). If you add objects to your set, then you may need to overload the == and != operators for the object's class so your template-based set class can properly determine membership.

5. **Task Manager.** Write your task manager to see running processes on operating system (OS). A program is an executable file stored on hard disk. A process is a program running in the memory by OS. A process has a name, Process ID (PID) and memory usage and other info depending on the OS as shown for Linux and Windows. On windows, get the running processes in the command line with **`tasklist`** command.

   Create a class to represent process, create a class that represents list of running processes. Add functions to fill it with processes information and to allow displaying the processes as in the picture below (1) sorted by name, (2) sorted by PID and sorted by memory use. Add a function that gets the list of processes from OS and loads the process list. Write a program to test your classes.

```
chrome.exe       10100   Running     marius.bancila    227756   32-bit
cmd.exe          512     Running     SYSTEM                48   64-bit
explorer.exe     7108    Running     marius.bancila     29529   64-bit
skype.exe        22456   Suspended   marius.bancila       656   64-bit
```

```
Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0          8 K
System                           4 Services                   0         24 K
Registry                        96 Services                   0     10,792 K
smss.exe                       416 Services                   0        284 K
```

6. **Game of Life.** (See https://playgameoflife.com/) Write a program that implements the *Game of Life* cellular automaton proposed by *John Horton Conway*. The universe of this game is a grid of square cells that could have one of two states: **dead** or **alive**. Every cell interacts with its adjacent neighbors, with the following transactions occurring on every step:
   - Any live cell with fewer than two live neighbors dies, as if caused by underpopulation
   - Any live cell with two or three live neighbors lives on to the next generation
   - Any live cell with more than three live neighbors dies, as if by overpopulation
   - Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction

   The status of the game on each iteration should be displayed on the console, and for convenience, you should pick a reasonable size, such as 20 rows x 50 columns or 30 x 30 columns or 20 x 20.

   Implement a class called **`Universe`** with these functions (add more if needed):

   - **`initialize(…)`** generates a starting layout

- **`reset(…)`** sets all the cells as dead.
- **`count_neighbors(…)`** returns the number of alive neighbors.
- **`next_generation(…)`** produces a new state of the game based on the transition rules.
- **`display(…)`** shows the game status on the console (It is better to erase screen and rewrite, and NO it is console, no graphics is needed)
- **`run(…)`** to start the game for a certain number of runs.