**Workshop 4: Containers**

In this workshop, you will code three classes that are in composition and aggregation relations. The classes will simulate a very simplified form of reservation management for a restaurant. The restaurant will manage a collection of reservations (composition); a messaging system will send confirmations for the reservation.

**Learning Outcomes**

Upon successful completion of this workshop, you will have demonstrated the abilities to:

* design and code composition and aggregation class relationships
* use member functions of the string class to parse a string into tokens based on simple rules
* design and code a class that manages a dynamically allocated array of pointers to objects

**Submission Policy**

The workshop is divided into two coding parts and one non-coding part:

* *Part 1*: worth 0% of the workshop's total mark, is optional and designed to assist you in completing the second part.
* *Part 2*: worth 100% of the workshop's total mark, is due on **Sunday at 23:59:59** of the week of your scheduled lab. Submissions of *Part 2* that do not contain the *reflection* are not considered valid submissions and are ignored.
* *reflection*: non-coding part, to be submitted together with *Part 2*. The reflection does not have marks associated to it, but can incur a **penalty of max 40% of the whole workshop's mark** if your professor deems it insufficient (you make your marks from the code, but you can lose some on the reflection).

The workshop should contain ***only work done by you this term*** or provided by your professor. Work done in another term (by you or somebody else), or work done by somebody else and not **clearly identified/cited** is considered plagiarism, in violation of the Academic Integrity Policy.

Every file that you submit must contain (as a comment) at the top **your name**, **your Seneca email**, **Seneca Student ID** and the **date** when you completed the work.

* If the file contains only your work, or work provided to you by your professor, add the following message as a comment at the top of the file:

I have done all the coding by myself and only copied the code that my professor provided to complete my workshops and assignments.

* If the file contains work that is not yours (you found it online or somebody provided it to you), **write exactly which parts of the assignment are given to you as help, who gave it to you, or which source you received it from.** By doing this you will only lose the mark for the parts you got help for, and the person helping you will be clear of any wrong doing.

**Compiling and Testing Your Program**

All your code should be compiled using this command on matrix:

/usr/local/gcc/10.2.0/bin/g++ -Wall -std=c++17 -g -o ws file1.cpp file2.cpp ...

* -Wall: compiler will report all warnings
* -std=c++17: the code will be compiled using the C++17 standard
* -g: the executable file will contain debugging symbols, allowing *valgrind* to create better reports
* -o ws: the compiled application will be named ws

After compiling and testing your code, run your program as following to check for possible memory leaks (assuming your executable name is ws):

valgrind ws

To check the output, use a program that can compare text files. Search online for such a program for your platform, or use *diff* available on matrix.

**Part 1 (0%)**

The first portion of this workshop consists of modules:

* w4 (supplied)
* Reservation

Enclose all your source code within the sdds namespace and include the necessary guards in each header file.

**w4 Module (supplied)**

**Do not modify this module!** Look at the code and make sure you understand it.

**Reservation Module**

This module includes the definition of a class that manages information about a single reservation at a restaurant for a date/time in February.

Design and code a class named Reservation that can store the following information (for each attribute, chose any type that you think is appropriate--you must be able to justify the decisions you make):

* **reservation id**: an array of characters
* **the name on the reservation**
* **the email** to be used to send a confirmation that the reservation can be honored or cannot
* **the number of people** in the party
* **the day** when the party expects to come (for simplicity, the day is an integer between 1 and 28)
* **the hour** when the party expects to come (for simplicity, the hour is an integer between 1 and 24)

***Public Members***

* a default constructor
* void update(int day, int time): a modifier that receives as parameters a new day and time for the reservation and updates the attributes with received values. Assume the parameters are correct and don't require validation.
* Reservation(const std::string& res): A constructor that receives the reservation as a string; this constructor is responsible for extracting information about the reservation from the string and storing the tokens in the instance's attributes. The string will always have the following format:
* ID:NAME,EMAIL,PARTY\_SIZE,DAY,HOUR

This constructor should remove all leading and trailing spaces from the **beginning and end** of any token extracted from the string.

When implementing the constructor, consider this following functions:

* + [std::string::substr()](https://en.cppreference.com/w/cpp/string/basic_string/substr)
  + [std::string::find()](https://en.cppreference.com/w/cpp/string/basic_string/find)
  + [std::string::erase()](https://en.cppreference.com/w/cpp/string/basic_string/erase)
  + [std::stoi()](https://en.cppreference.com/w/cpp/string/basic_string/stol)

***Friend Helpers***

* overload the insertion operator to insert the contents of a reservation object into an ostream object:
  + if the hour is between 6AM and 9AM (inclusive), the kitchen serves breakfast:
* Reservation ID: NAME <email> Breakfast on day DAY @ HOUR:00 for #PARTY\_SIZE people.
  + if the hour is between 11AM and 3PM (inclusive), the kitchen serves lunch:
* Reservation ID: NAME <email> Lunch on day DAY @ HOUR:00 for #PARTY\_SIZE people.
  + if the hour is between 5PM and 9PM (inclusive), the kitchen serves dinner:
* Reservation ID: NAME <email> Dinner on day DAY @ HOUR:00 for #PARTY\_SIZE people.
  + at any other time the kitchen is closed and only drinks can be served:
* Reservation ID: NAME <email> Drinks on day DAY @ HOUR:00 for #PARTY\_SIZE people.
  + the ID on the reservation should display on a field of size 10, aligned to the right
  + the name on the reservation should display on a field of size 20, aligned to the right
  + the email on the reservation (including the characters < and >) display on a field of size 20, aligned to the left.
  + this operator should insert the end line character before returning control.
  + if the reservation is for one person, your output should say "person" instead of "people".

**Sample Output**

When the program is started with the command (the file data.txt is provided):

ws data.txt

the output should look like the one from the sample\_output.txt file.

**Test Your Code**

To test the execution of your program, use the same data as shown in the output example above.

Upload your source code to your matrix account. Compile and run your code using the latest version of the g++ compiler (available at /usr/local/gcc/10.2.0/bin/g++) and make sure that everything works properly.

Then, run the following command from your account (replace profname.proflastname with your professor’s Seneca userid):

~profname.proflastname/submit 345\_w4\_p1

and follow the instructions.

***This part represents a milestone in completing the workshop and is not marked!***

**Part 2 (100%)**

The second part of this workshop upgrades your solution to include two more modules:

* Restaurant
* ConfirmationSender

The module Reservation doesn't need any change.

**Restaurant Module**

Add a Restaurant module to your project. This module should maintain a dynamically allocated array of objects of type Reservation: Reservation\* (each element of the array is an object of type Reservation).

***Public Members***

* Restaurant(const Reservation\* reservations[], size\_t cnt): a constructor that receives as a parameter an array of pointers to objects of type Reservation (i.e., each element of the array is a pointer). If you need a refresh on arrays of pointers, re-read the material from the last term (chapter **Abstract Base Classes**, section **Array of Pointers**).
  + this constructor should store **copies** of all reservations
* add any other special members that are necessary to manage the reservations stored
* size\_t size() const: a query that returns the number of reservations in the system.

***Friend Helpers***

* overload the insertion operator to insert the content of a Restaurant object into an **ostream** object. This operator should use a local to function variable to count how many times this operator has been called (CALL\_CNT below).
  + if there are no reservations:
* --------------------------
* Fancy Restaurant (CALL\_CNT)
* --------------------------
* This restaurant is empty!
* --------------------------
  + if there are reservations:
* --------------------------
* Fancy Restaurant (CALL\_CNT)
* --------------------------
* RESERVATION
* RESERVATION
* ...
* --------------------------

**ConfirmationSender Module**

Add a ConfirmationSender module to your project. The purpose of this module is to receive all the reservations from multiple restaurants, and contact the recipients with a confirmation message.

This module should maintain a dynamically allocated array of **pointers** to objects of type Reservation: const sdds::Reservation\*\* (each element of the array is a pointer to an object of type Reservation).

***Public Members***

* add any special members that are necessary to manage the resource (the resource is an **array of pointers**; your class must manage this array, but the objects at the addresses stored in the array are managed outside this class)
* ConfirmationSender& operator+=(const Reservation& res): adds the reservation res to the array by adding its address.
  + if the address of res is already in the array, this operator does nothing
  + resizes the array to make room for res if necessary
  + stores the **address** of res in the array (your function should not make copies of the reservation itself)
* ConfirmationSender& operator-=(const Reservation& res): removes the reservation res from the array by removing its address
  + if the address of res is not in the array, this operator does nothing
  + searches the array for the address of res, sets the pointer in the array to nullptr if res is found. **To challenge yourself, try to actually resize the array.**

***Friend Helpers***

* overload the insertion operator to insert the content of a ConfirmationSender object into an **ostream** object:
  + if there are no reservations to confirm:
* --------------------------
* Confirmations to Send
* --------------------------
* There are no confirmations to send!

--------------------------

* + if there are reservations to confirm

--------------------------

Confirmations to Send

--------------------------

RESERVATION

RESERVATION

...

--------------------------

**Sample Output**

When the program is started with the command (the file data.txt is provided):

ws data.txt

the output should look like the one from the sample\_output.txt file.

**Reflection**

Study your final solution, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. **This should take no less than 30 minutes of your time and the result is suggested to be at least 150 words in length.**

Create a **text** file named reflect.txt that contains your detailed description of the topics that you have learned in completing this particular workshop and mention any issues that caused you difficulty and how you solved them. Include in your explanation—**but do not limit it to**—the following points:

* the difference between the implementations of a composition and an aggregation.
* the difference between the implementations of move and copy functions in a composition and an aggregation.

Identify places in your code where you have used **association**, **aggregation** and **composition** relations.

To avoid deductions, refer to code in your solution as examples to support your explanations.

**Submission**

To test and demonstrate execution of your program use the same data as shown in the output example above.

Upload the source code and the reflection file to your matrix account. Compile and run your code using the latest version of the g++ compiler (available at /usr/local/gcc/10.2.0/bin/g++) and make sure that everything works properly.

Then, run the following command from your account (replace profname.proflastname with your professor’s Seneca userid):

~profname.proflastname/submit 345\_w4\_p2

and follow the instructions.