Workshop 4

Containers

In this workshop, you code a container class that holds notifications and a class that holds separate messages.

LEARNING OUTCOMES

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

- design and code composition and aggregation class relationships
- use the 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 *in-lab* section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. If you do not attend the workshop, you can submit the *in-lab* section along with your *at-home* section (see penalties below). The *at-home* portion of the lab is due on the day that is four days after your scheduled in-lab workshop (23:59:59) (even if that day is a holiday).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.

You are responsible to back up your work regularly.

Late Submission Penalties:

- *In-lab* portion submitted late, with *at-home* portion: **0** for *in-lab*. Maximum of 7/10 for the entire workshop.
- If any of *in-lab*, *at-home* or *reflection* portions is missing, the mark for the workshop will be **0**/10.

SPECIFICATIONS – IN LAB

The in-lab part of this workshop consists of two modules:

- **w4** (supplied)
- **Message**

Enclose all your source code within the **sict namespace** and include the necessary guards in each header file. The output from your executable running Visual Studio with the following command line argument should look like

```
Command Line : C:\Users\...\Debug\in_lab.exe w4_test.dat

Messages loaded from file
=========================
>User  : jim
 Tweet : Workshop 4 is cool
>User  : harry
 Reply : jim
 Tweet : working on workshop 4 now
>User  : dave
 Tweet : what the ^#$%!
>User  : john
 Reply : harry
 Tweet : I'm done
```

The input for testing your solution is stored in a user-prepared file. The name of the file is specified on the command line as shown in red above. The file is supplied with this workshop. The contents of this particular file are

```
jim Workshop 4 is cool
harry @jim working on workshop 4 now

chris
dave what the ^#$%!
john @harry I'm done
```

The first message consists of a user name followed by a tweet. The second message consists of a user name, a reply name prefaced by an **@**, and followed by a tweet. Your solution ignores incomplete messages, such as the third message or fourth message here.

Message Module

Design and code a class named **Message** for managing a single message between users.

Your class design includes the following public member functions:

- A default constructor that places the object in a safe empty state.
- **Message(const std::string& str)** – a one-argument constructor that receives a reference to an unmodifiable string and parses the string into three substrings: the user who sent the message, the user to whom the message was a reply and the text of the message itself. The reply portion of the message is optional and prefaced by the '@' character. The user and reply substrings end with a space (are space terminated). The text of the message is newline terminated. If there is no text in the message, this function places the object into a safe empty state. For a good design (your instructor may deduct marks for a poor design):
  - Limit the parsing of the string received to the scope of this constructor – do not repeat parsing in other member functions.
  - Do not iterate through the characters of the string to find the delimiters. Instead use the **npos**, **find()**, **substr()** members of the **std::string** library directly.
- **bool empty() const** – a query that returns true if the current object is in a safe empty state; false otherwise
- **void display(std::ostream& os) const** – a query that inserts into the output stream **os** the message in the format shown above followed by a newline. If the current object is in a safe empty state, this function does nothing. The format of the output is as follows with the data highlighted in red:
  - `>User  : user`
  - ` Reply : reply (optional line)`
  - ` Tweet : message`

In-Lab Submission (30%)

To test and demonstrate 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 gcc compiler 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 345XXX_w4_lab<ENTER>

and follow the instructions. Replace **XXX** with the section letter(s) specified by your instructor.

SPECIFICATIONS – AT HOME

The at-home part of this workshop consists of four modules:

- **w4** (supplied)
- **Message** (from in-lab)
- **MessagePack** (a composition of **Message** objects)
- **Notifications** (an aggregation of **Message** objects)

Enclose all your source code within the **sict namespace** and include the necessary guards in each header file. The output from your executable running Visual Studio with the following command line argument should look like

```
Command Line : C:\Users\...\Debug\at_home.exe w4_test.dat

Messages loaded from file
=========================
>User  : jim
 Tweet : Workshop 4 is cool
>User  : harry
 Reply : jim
 Tweet : working on workshop 4 now
>User  : dave
 Tweet : what the ^#$%!
>User  : john
 Reply : harry
 Tweet : I'm done

Message Pack 1 Contents
=======================
 No of Messages 4
>User  : jim
 Tweet : Workshop 4 is cool
>User  : harry
 Reply : jim
 Tweet : working on workshop 4 now
>User  : dave
 Tweet : what the ^#$%!
>User  : john
 Reply : harry
 Tweet : I'm done

Message Pack 2 Contents
=======================
 No of Messages 4
>User  : jim
 Tweet : Workshop 4 is cool
>User  : harry
 Reply : jim
 Tweet : working on workshop 4 now
>User  : dave
 Tweet : what the ^#$%!
>User  : john
 Reply : harry
 Tweet : I'm done

Message Pack 1 Contents
=======================
 No of Messages 0

Notification Contents
=====================
 No of Messages 0

Notification Contents
=====================
 No of Messages 3
>User  : jim
 Tweet : Workshop 4 is cool
>User  : dave
 Tweet : what the ^#$%!
>User  : john
 Reply : harry
 Tweet : I'm done

Backup Contents
===============
 No of Messages 3
>User  : jim
 Tweet : Workshop 4 is cool
>User  : dave
 Tweet : what the ^#$%!
>User  : john
 Reply : harry
 Tweet : I'm done

Notification Contents
=====================
 No of Messages 0
```

The input for testing your solution is stored in a user-prepared file. The name of the file is specified on the command line as shown in red above. The file is supplied with this workshop.

MessagePack Module

Design and code a class named **MessagePack** that manages a composition of **Message** objects. The number of objects is defined at run-time by the client module.

In addition to the standard special functions (Constructor, Destructor, Copy and Move) that a class needs to manage dynamically allocated memory, your class includes the following public member functions and helper function:

Complete the main function supplied with the at-home project file by adding 4 statements:

- A default constructor that places the object in a safe empty state

- A two-argument constructor that receives the address of an array of **Message** objects and the number of elements in that array. If the number of messages is positive-valued and the address is a valid address, your function allocates dynamic memory for the specified number of **Message** objects and stores *copies* of those objects that are not empty. Otherwise, your function places the object in a safe empty state.
- **void display(std::ostream& os) const** – a query that inserts into the output stream **os** each element in the array of **Message** objects.
- **size_t size() const** – a query that returns the number of **Message** objects stored in the current object.
- **operator<<** - a non-friend helper function that inserts into **os** the contents of the **MessagePack** object.

Notifications Module

Design and code a class named **Notifications** that manages an aggregation of **Message** objects. The number of objects is defined at run-time by the client module.

As in the in-lab section, you are free to select the data members of your class as you consider most appropriate for your design. Hint: Consider using the following:

- **const Message**** - a pointer to a dynamically allocated array of addresses to unmodifiable **Message** objects
- **int** – the maximum number of addresses that can be stored in the aggregation
- **int** – the number of addresses currently stored in the aggregation

If you find ** syntax confusing, ask your instructor for an example of its usage.

In addition to the standard five functions that a class needs to manage dynamically allocated memory, your class includes the following public member functions and helper function:

- A default constructor that places the object in a safe empty state
- A one argument constructor that receives the maximum number of elements in the aggregation. If the number of messages is positive-valued, your function allocates dynamic memory for the specified number of pointers to **Message** objects. Otherwise, your function places the object in a safe empty state.
- **Notifications& operator+=(const Message& msg)** – a modifier that receives a reference to an unmodifiable **Message** object. If the object is not empty and the current object has room to store an address to a **Message** object, your function stores that address. Otherwise, this function does nothing. In both cases, this function returns a reference to the current object.
- **Notifications& operator-=(const Message& msg)** – a modifier that receives a reference to an unmodifiable **Message** object. Your function searches the current object for the

address of the **Message** object. If your function finds the address stored in the current object, it removes that object from the aggregation (replaces its address with **nullptr**). Otherwise, your function does nothing. In any case, your function returns a reference to the current object. (Hint: in your design, you may choose to compress the address set by shifting each subsequent address to fill the gap created by the address removal allowing newly added addresses to always be added to the end.)

- **void display(std::ostream& os) const** – a query that inserts into the output stream **os** each element in the array of **Message** objects.
- **size_t size() const** – a query that returns the number of **Message** objects pointed to by the current object.
- **operator<<** - a non-friend helper function that inserts into **os** the contents of the **Notifications** object.

Note that the one-argument constructor and the copy assignment operator for this class dynamically allocate memory for an array of *pointers to* **Message** objects, not an array of **Message** objects.

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. Explain in your own words what you have learned in completing this workshop. Include in your explanation but do not limit it to the following points (40%):

- 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.

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

Include all corrections to the Quiz(zes) you have received (30%).

At-Home Submission (70%)

To test and demonstrate 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 gcc compiler 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 345XXX_w4_home<ENTER>
```

and follow the instructions. Replace **XXX** with the section letter(s) specified by your instructor.