Assignment 2: Membership System

Student Name: Luca J. Ricagni

Student Number: 200894968

Module Name: Introduction to Object Orientated Programming

Module Code: ECS793P

Due Date: 14/12/2020

Table of Contents

Directions for use 3
Hierarchy design 3
Approach to programming 3
Issues faced and limitations 4
Code Explanation of each function in the code base 5
Class – Date 5
Class – User Inventory 6
Class – User 6
Common functions implemented in all User derived classes: 7
Class – Free 7
Class – Silver 7
Class – Gold 7
Testing 7
Appendices 8
Appendix 1 – UML Class Diagram 8
Appendix 2 – Table of tests 9

Directions for use

Ensure you have GCC and Linux/Linux emulator installed, open a terminal, navigate to the folder the file is stored in, then compile the file using the command:

g++ MembershipAssignment.cpp -o myalias

Then run using the same alias: ./myalias

A main menu will then appear on the screen, type in “8” and press Enter to fill the system with some example users.

Hierarchy design

The first step for completing this project was to identify which main objects would be required and then to decide on a suitable data structure for their storage. Based off the project description it was clear that there would be three types of member: Free, Silver and Gold, which could all be implemented as derived classes inheriting from User which would be an abstract base class.

Furthermore, stipulated as a bonus feature in the specification, all searches, adding and removing of users etc must be done from a User Inventory class, so this was also included from the start.

Due to the many different dates which needed to be stored, and in some of the searches compared, it became clear that a way to compare dates would be needed, but the base C++ data comparison functions seemed cumbersome. Therefore, an additional class Date was implemented which would allow for this comparison easily and efficiently via the use of operator overloading.

A full UML class diagram detailing the hierarchy design can be found in Appendix 1.

Approach to programming

As previously mentioned, five key classes were identified as being essential to this project: the abstract base class User, the three membership type classes Free, Silver and Gold, and the User Inventory class. These classes were all implemented simultaneously as when functionality was added it was often necessary to add or change things in all these classes.

Next a decision had to be made regarding which container to use to store the users in. Two main containers where considered as a way of storing objects: a vector (list) or a map (dictionary). A vector is just simply a list of objects where each element has pointers which point to the next and previous element in the vector. A map stores objects as key-value pairs where the key must be unique, allowing for efficient object retrieval.

Using a map container keyed on the unique username would result in a very efficient User lookup when searching by username, with a complexity of $O(1)$, compared to using a vector where the worst case would be that the whole list would have to be walked which would have a complexity of $O(n)$. The other types of searches that would require iterating over either entire container thus would have largely the same time requirement of $O(n)$.

A map was thus chosen using the unique username as the key and a pointer to the base class User as the value. This was done because the “value” of a map can only be of a single type, and as there are three types of members a pointer to the base class must then be stored. An alternative would have been to use three maps, one for each member type, but this would have complicated matters.

The reason the map value stored is a pointer to the base class rather than the base class itself is because these members would need to be dynamically created and stored on the *heap* using the *new* keyword rather than on the *stack*. This meant that the User objects would not get destroyed when they went out of scope but also that they had to be cleaned up in the destructor using the delete keyword to prevent memory leaks.

Apart from a username lookup (time complexity of $O(1)$), all the other searches work by iterating over the entire map of users and checking each against the given criteria, printing the user's information if the criteria is met, and ignoring it otherwise (complexity of $O(n)$).

To save users in between program end and restart, the User objects had to be saved to a file upon program end and read into the system upon program start. To achieve this each user's information was saved on a new line as a comma separated string to a .txt file.

While using the program the operator should be able to print the desired information in a readable format. Each search type prints the relevant information as a formatted string.

As various points during runtime, such as searches or adding a user, the program may ask the operator for input, some of these are simple *1-n* menu selections in which the operator's input is checked against the options given, and the operator is asked to retry their selection if the input is not valid. Other types of operator input would be for things such as getting information like name and date of birth when adding a user. In these cases, the name would be checked to ensure it only contains alpha characters (letters), and the date of birth would be checked to ensure it is a valid date in the correct format.

The main UI with which the operator would use was then added, which consists of a menu of options where all the functionality can be accessed, such as searches, adding or removing users, adding, or removing friends, and saving and exiting the program.

While coding new functionality suitable test functions were also added. A menu item to run them all was then added for convenience to enable a quick regression test of the entire codebase; highlighting any new issues or confirming that previously implemented code still worked as intended, giving a far greater level of confidence in the codebase as it evolved.

Several enumerated types were also used to increase code readability: membership type, payment type and music preferences which make it much easier to read and understand.

Issues faced and limitations

The first major issue was deciding how to deal with the various dates that needed to be stored such as date of birth, date of registry etc. Several library implementations were examined but they were overly complicated and did not quite have the functionality desired, so it was decided to create a bespoke Date class. This Date class would contain integers for year, month, day and would allow for effective date comparisons via overloading the less that (<), greater than (>) and equals to (==) operators. This class also evolved to contain several useful member functions to check if a date is valid, a leap year or a future date. These functions were also made *static*, meaning that they can be used outside of an instance of the class, allowing dates to be verified without instantiating a Date object.

The second major issue faced was how to store the different types of users in the map (dictionary). As the map can only store a single type as the value it must then be of type User (the base class), allowing any derived classes to also be stored. It then became clear that the map must store pointers

to dynamically created user types as an issue where objects went out of scope was encountered which proved hard to find. This meant that for a user to be removed from the system the dynamic pointer to it must be deleted to stop memory leaks occurring, before erasing it from the dictionary.

A result of storing dynamic pointers to the User base class was an issue where overridden member functions within the derived classes were not be called (the one in the base class was used instead). It was therefore necessary in these situations to cast to the relevant derived User type to call the correct derived member function.

The third major issue faced was how to implement a user's friends list. The two options considered for this were either to store a list of strings (usernames), or a list of pointers to other users. Using pointers may have simplified things somewhat as a user's friends can be directly accessed, however if not delt with properly they could also have led to memory leaks or pointed to deleted objects which could cause a lot of issues. Due to the way User objects are stored in the map and keyed on username, it is very easy and efficient to search for a user's friends by performing a simple search, having a complexity of $O(1)$. It was therefore decided to store a user's friends as a list (vector) of strings. For added efficiency when removing or adding friends it was also assumed that friendship is a bi-directional relationship, i.e. if user A is a friend of user B, then user B must also be a friend of user A. This would allow friends to be updated easily when a user is deleted by simply searching through their friend list, looking up each friend in the map in turn and then removing this User's entry from them all before deleting the target User, rather than having to check the entire User collection for friend links to remove.

Another major issue faced was how to write all the user's information to a file upon exiting the program in such a way that it could be read back into the program upon start-up. The easiest way to do this was to write all the necessary information as a comma separated string, which could be split at the commas when read back in. However, this caused some issues when writing/reading the list of friends and music preferences, they could not be comma separated as they are of unknown length and are part of the same member variable. It was therefore decided to delimit these lists using the ':' character instead.

This solution brought to light some additional problems which needed addressing. Namely the need to ensure that the ',' and ':' characters were not found anywhere they were not supposed to be. This was accomplished by using the *isalpha* function on names and last names, guaranteeing that all the characters entered were alphabetic characters, and using the *isalnum* function on usernames, to make sure all characters in a username are alphanumeric.

Lastly it became clear that there would have to be two read passes of a file on loading, one to read in all users and a second to update all the friend lists (as only Users in the map can be added as friends).

Code Explanation of each function in the code base

Class – Date

Constructor(): Takes as input a string in the format YYYY-MM-DD, extracts the relevant numbers from it and creates an instance of the date class with them.

IsLeap(): Takes as input a year as an integer and returns true if it is a leap year, i.e. if it is a multiple of 4 and not a multiple of 100, or is a multiple of 400. This function is *static* meaning it can be declared outside of an instance of the class.

IsValidDate(): Takes as input a date as a string in the format YYYY-MM-DD and returns true if it is a valid date. This function is *static* meaning it can be declared outside of an instance of the class.

IsFutureDate(): Takes as input a date as a string in the format YYYY-MM-DD and returns true if it is a future date. This function is *static* meaning it can be declared outside of an instance of the class.

GetDateAsString(): Formats the year, month and day integers stored into a string of the format YYYY-MM-DD and returns it.

Operator<(): Overloads the less than (<) operator allowing instances of the Date class to be compared.

Operator>(): Overloads the greater than (>) operator allowing instances of the Date class to be compared.

Operator==(): Overloads the less than (==) operator allowing instances of the Date class to be compared.

Class – User Inventory

~UserInventory(): Destructor for this class, required to clean up the dynamically created User objects as this class contains the map (dictionary) which effectively 'owns' the entire User list.

SearchUsers(): Searches for members of a given membership type, searches for a specific user by username, or searches users by registry range, depending on the parameters supplied.

SearchMembers(): Searches members by monthly fee, payment type, or members with a member since date in a given range, depending on the parameters supplied.

DisplayAllMembers(): Iterates through the map (dictionary) of Users calling DisplayInfo() on each to print their details sequentially.

DisplayInformation(): Displays the information of a single user by username.

ResetDictionaryOfUsers(): Completely clears the dictionary of all Users.

AddUser(): Takes the required parameters adds a user if one of that username does not exist already.

GetUser(): Returns a pointer to a given user if it exists, returns NULL otherwise.

RemoveUser(): Removes a user from the map (dictionary) if it exists.

SaveUsers(): Saves all users to a file.

ReadUsers(): Reads all users from a file and adds them to the map (dictionary).

AddFriends(): Adds bi-directional friendship between a user and a list of friends.

RemoveFriends(): Removes bi-directional friendship between a user and a list of friends.

Class – User

Constructor(): Takes as input the username, name, last name, gender, date of birth, music preferences and date of registry and creates an instance of the class. Date of registry is defaulted to an empty string so if one is not supplied it is set to today's date.

GetUserName(): Returns the username as a string.

GetFriends(): Returns a user's friends as a vector (list) of strings.

GetDateOfBirthAsString(): Returns the user's date of birth as a string by calling the GetDateAsString() function on the nested Date class.

GetDateOfRegistryAsString(): Returns the user's date of registry as a string by calling the

GetDateOfRegistry(): Returns a pointer to the instance of the Date class containing the date of registry.

AddFriend(): Takes as input the username of a friend to add.

RemoveFriend(): Takes as input the username of a friend to remove.

Common functions implemented in all User derived classes:

DisplayInfo(): Returns the information of the user as a formatted string.

SaveInfo(): Returns the comma separated string of the user's information to be saved.

GetMembershipType(): Returns the membership type as a string.

GetMonthlyFeeAsString(): Returns the monthly fee as a string.

GetPaymentType (): Returns the payment type as a string.

Class – Free

Constructor(): Sets the monthly fee to £0 and calls the User constructor with the rest of the information.

Class – Silver

Constructor(): Sets the monthly fee and the payment type, checks to see if date of registry and member since dates are supplied, if not then sets them to today, passes the remaining arguments to the User constructor.

GetMemberSinceDate(): Returns the member since date as a formatted string.

GetDateOfMembership(): Returns a pointer to the instance of Date which contains the member since date.

Class – Gold

Constructor(): Sets the monthly fee and the payment type, checks to see if date of registry and member since dates are supplied, if not then sets them to today, passes the remaining arguments to the User constructor.

GetMemberSinceDate(): Returns the member since date as a formatted string.

GetDateOfMembership(): Returns a pointer to the instance of Date which contains the member since date.

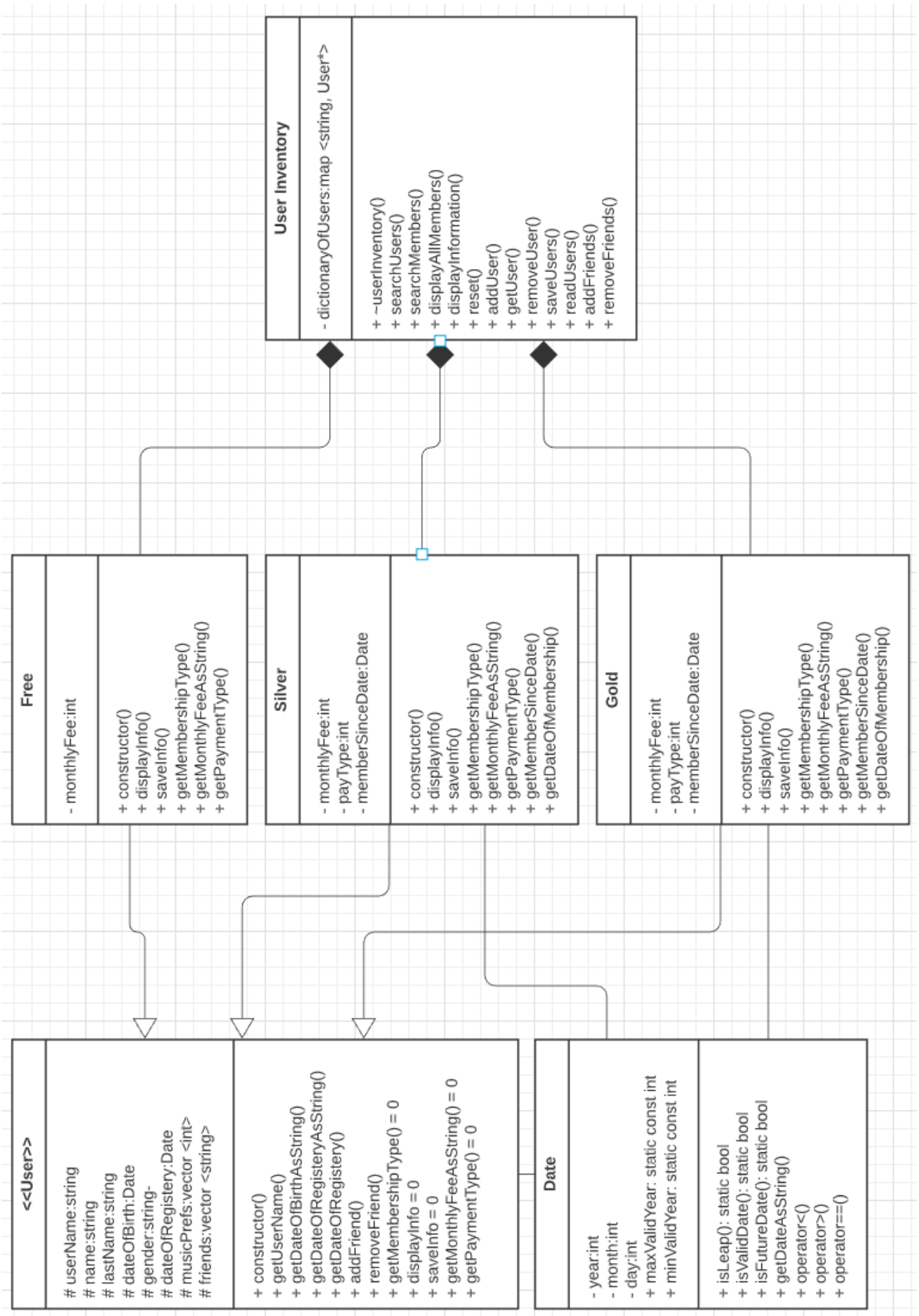
Testing

In addition to the automated tests triggered from menu option 7, many manual tests were also conducted on this application, testing many different inputs and cases.

This full set of tests performed can be found in Appendix 2.

Appendices

Appendix 1 – UML Class Diagram



Appendix 2 – Table of tests

Function	Inputs	Expected behaviour	Observed behaviour
Search users by type	Select main menu 1 Select sub-menu 1 Select sub-menu 1 (Free)	Print all users of the “Free” type	Prints all users of the “Free” type
Search users by type	Select main menu 1 Select sub-menu 1 Enter 10	Application will ask for valid input	Application asks for valid input
Search users by username	Select main menu 1 Select sub-menu 2 Enter “Crystal02” (user in example users)	Print info of given user	Prints info of given user
Search users by username	Select main menu 1 Select sub-menu 2 Enter “Invalid”	Application will say User does not exist	Application says User does not exist
Search users by registry range	Select main menu 1 Select sub-menu 3 Enter “2015-01-01” Enter “2020-01-01”	Application will print information of users in the registry range	Application prints information of users in the registry range
Search users by registry range	Select main menu 1 Select sub-menu 3 Enter “2015-01-01” Enter “20202323-01-01”	Application will ask for valid input for second date	Application asks for valid input for second date
Search members by monthly fee	Select main menu 2 Select sub-menu 1 Select sub-menu 1	Print all users with monthly fee of £0	Prints all users with monthly fee of £0
Search members by monthly fee	Select main menu 2 Select sub-menu 1 Enter 10	Application will ask for valid input	Application asks for valid input
Search members by payment type	Select main menu 2 Select sub-menu 2 Select sub-menu 1	Print all users with payment type “Card”	Prints all users with payment type “Card”
Search members by payment type	Select main menu 2 Select sub-menu 2 Enter 10	Application will ask for valid input	Application asks for valid input
Search members by membership date range	Select main menu 2 Select sub-menu 3 Enter “2015-01-01” Enter “2020-01-01”	Application will print information of users with member since date in range	Application prints information of users with member since date in range
Display all members	Select main menu 3	Print information of all users	Application prints information of all users
Add User	Select main menu 4 Enter “Luca::--,.”	Application will catch non alphanumeric characters	Application catches non alphanumeric characters
Remove User	Select main menu 5 Enter “Luca::--,.”	Application will print “User does not exist”	Application prints “User does not exist”
Add friends	Select main menu 6 Select sub-menu 1 Enter “Crystal02”	Application will add bi-directional friendship between the two users	Application adds bi-directional friendship between the two users

	Enter “ShinZap”		
Remove friends	Select main menu 6 Select sub-menu 2 Enter “Crystal02” Enter “ShinZap”	Application will remove bi-directional friendship between the two users	Application removes bi-directional friendship between the two users
Run test functions	Select main menu 7	Application will run test functions and print number passed/failed	Application runs test functions and print number passed/failed
Add sample users	Select main menu 8	Application will add sample users to system	Application adds sample users to system
Save and exit	Select main menu 9	Application will save users to file and exit	Application saves users to file and exits