COMP 2404 A/C – Assignment #2

Due: Wednesday, February 24 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program, in the VM provided for this course, to manage the calls made by the subscribers of a telephone company (also known as a *telco*). Your program will simulate a telco that has multiple subscribers. Each subscriber stores a collection of its incoming calls (the calls received by the subscriber), and a collection of its outgoing calls (those made by the subscriber). You will implement your program using objects from the different design categories, based on a UML class diagram provided for you.

2. Learning Outcomes

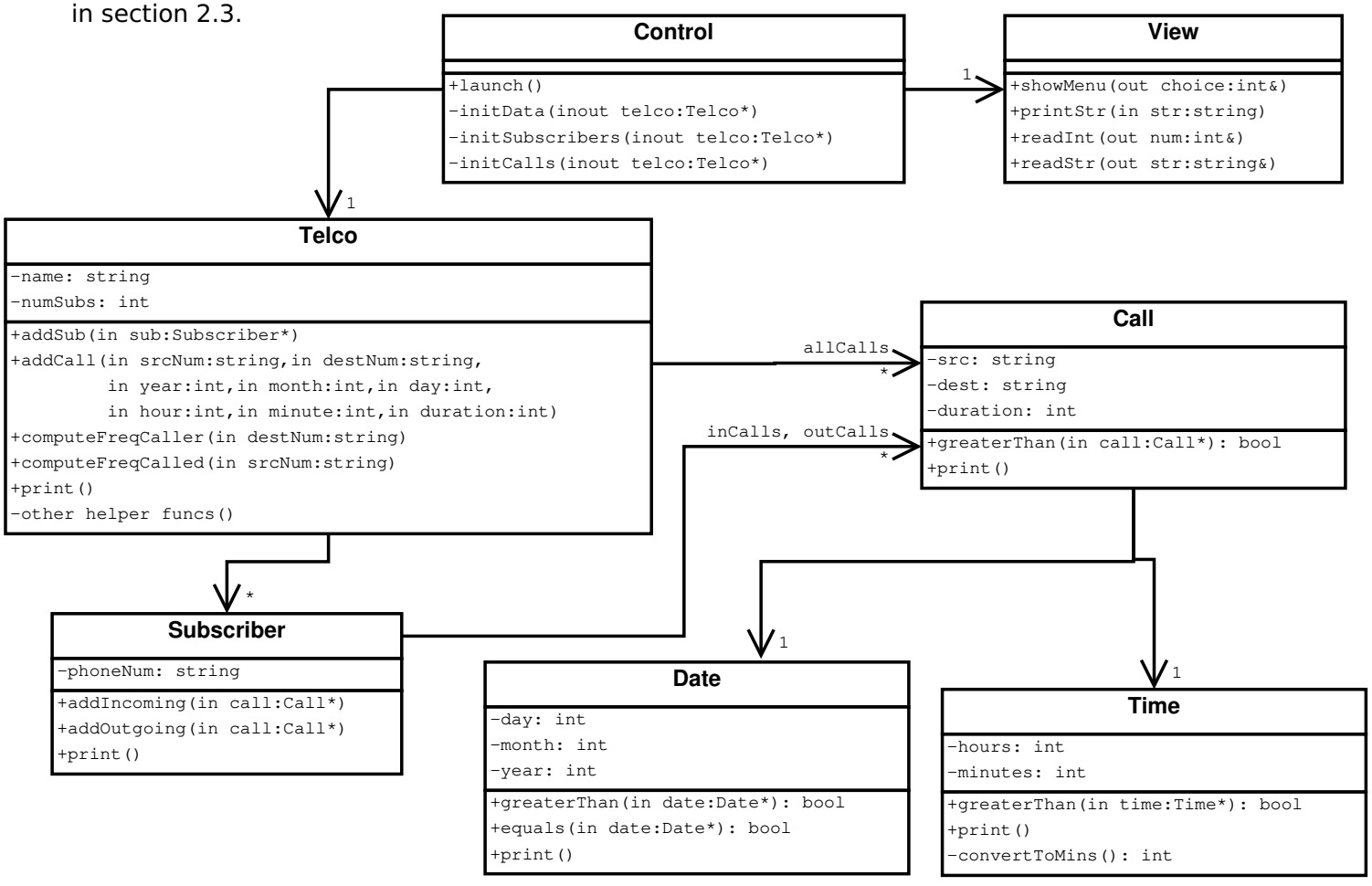
With this assignment, you will:

- practice implementing a design that is given as a UML class diagram
- implement a program separated into control, view, entity, and collection objects
- write code that follows the principle of least privilege

3. Instructions

3.1. Understand the UML class diagram

You will begin by understanding the UML class diagram below. Your program will implement the objects represented in the diagram, as they are shown. UML class diagrams are explained in the course material in section 2.3.



3.2. **Download the posted code**

The `a2-posted.tar` file has been provided for you in *cuLearn*. This file contains the `View` class that your code must use for most communications with the end user. It contains a skeleton `Control` class, with some data initialization member functions that your code is required to call. It also contains the `Time` class that your code will use, as indicated in the UML class diagram above.

3.3. **Modify the `Date` class**

You will begin with the `Date` class that we worked on during lectures, in section 3.1, program #3.

You will make the following changes to the `Date` class, in accordance with the UML class diagram:

3.3.1. implement the `greaterThan()` member function that compares two dates; one date is considered greater than another if it occurs later

3.3.2. you may require the implementation of the `equals()` member function

3.3.3. remove the printing functions, and replace them with a `print()` member function that prints out the date using the format YYYY-MM-DD; for example, the due date for this assignment is 2021-02-24

Remember: An object can always access the private members of another object of the *same class*. **Do not** provide unnecessary getters.

3.4. **Implement the `Call` class**

You will create a new `Call` class that represents a telephone call made between two parties. At least one of the two parties must be a subscriber in the telco, and it's possible that both parties are subscribers.

The `Call` class will contain all the data members and member functions indicated in the UML class diagram; in addition:

3.4.1. the source and destination (i.e. the two parties) of the call are stored as phone numbers; each phone number in this program will be represented as a 10-digit string, for example "6135202600"

3.4.2. the date and time indicate when the call began

3.4.3. the duration is the number of minutes that the phone call lasted

3.4.4. the default constructor must do the following:

(a) it must take all the parameters required to initialize the new `Call` object, as well as the parameters required to initialize the date and time when the phone call began

(b) it must initialize all the required data members

(c) it must dynamically allocate the corresponding date and time objects, using the given parameters

(i) **NOTE:** it is bad design to force the calling object to create the date and time objects; this is the responsibility of the `Call` constructor

3.4.5. the destructor deallocates the required dynamically allocated objects

3.4.6. this class may require getter member functions for the source and destination of the call

3.4.7. a call is considered to be greater than another call if it began later in date and time; think about the logic here

3.4.8. the `print()` function prints out to the screen the date and time of the call, as well as its duration, and its source and destination phone numbers

3.5. **Modify the `Array` class**

You will modify the `Array` class that we implemented in the coding example of section 2.2, program #1, as follows:

3.5.1. modify the collection class so that it stores `Call` object pointers as data

3.5.2. modify the destructor so that it no longer deallocates the data

(a) the same `Call` objects will be stored in multiple arrays, so the destructor must **not** deallocate the data, otherwise there will be double free errors

3.5.3. implement a copy constructor that stores the same `Call` pointers in a new array; do **not** make copies of the `Call` objects; remember that you must reuse existing functions everywhere possible

3.5.4. implement a `cleanup()` member function that deallocates the `Call` objects stored in the array; you must think carefully about **when** and on which `Array` object this function must be called; this will become clearer when we implement the `Telco` class in a later step

3.5.5. modify the existing `add()` function to add a new call as follows:

(a) take a `Call` pointer as parameter

(b) add the given call to the array in its correct place, in *descending* (decreasing) order

(i) you must **shift** the elements in the array towards the back of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort, as this is both inefficient and unnecessary

(ii) you must use the `Call` class's `greaterThan()` function to perform the comparison

(c) return true if no errors occurred, and false otherwise

3.5.6. implement a new `bool add(Array& arr)` member function that does the following:

(a) loop over the elements of the `arr` parameter

(b) add each `arr` element to the `this` array

(c) return true if no errors occurred, and false otherwise

(i) **NOTE:** if an error occurs when adding an element, then no subsequent element can be added

3.5.7. some parts of the program will need to iterate through the call array; to facilitate this, you will implement a getter for the current number of elements, as well as a `Call* get(int)` member function that returns the element at the given index

3.6. Implement the `Subscriber` class

You will create a new `Subscriber` class that represents a telco customer.

The `Subscriber` class will contain all the data members and member functions indicated in the UML class diagram; in addition:

3.6.1. the subscriber's phone number is stored as a 10-digit string; as always, all data members must be initialized in the constructor

3.6.2. the collection of incoming calls to this subscriber is stored as an `Array` object

3.6.3. the collection of outgoing calls from this subscriber is stored as a separate `Array` object

3.6.4. this class may require getter member functions for the phone number, as well as for the collections of incoming and outgoing calls; if so, the collections must be returned by reference, not by value

3.6.5. the `addIncoming()` and `addOutgoing()` member functions add a new call to the corresponding call collection

3.6.6. the `print()` function does the following:

(a) print out to the screen the subscriber's phone number

(b) print out the subscriber's *combined* incoming and outgoing calls, as follows:

(i) declare a temporary `Array` object to represent the combination of all the subscriber's incoming and outgoing calls together; to do this, your code **must** use the `Array`'s copy constructor to copy one of the two collections (either the incoming or the outgoing calls)

(ii) add the calls of the *other* collection (incoming or outgoing) to the temporary `Array` object; to do this, your code **must** use the `add(Array&)` member function

(iii) print out to the screen the resulting temporary collection of all the subscriber's calls

3.7. Implement the `Telco` class

You will create a new `Telco` class that represents a telephone company.

The `Telco` class will contain all the data members and member functions indicated in the UML class diagram; in addition:

3.7.1. the collection of all calls made or received through this telco is stored as a `Array` object

3.7.2. the collection of all subscribers that receive service from this telco is stored as a primitive array of `Subscriber` pointers; this requires that the current number of subscribers in the array be tracked

3.7.3. the constructor initializes the required data members

3.7.4. the destructor deallocates the required dynamically allocated objects, including cleaning up the calls

3.7.5. the `addSub()` member function adds the given subscriber to the *back* (the end) of the subscribers array

3.7.6. the `addCall()` member function adds a new call to the telco, as follows:

(a) the subscriber array is searched to find the `Subscriber` object(s) that represent the call's source, or its destination, or both; if neither the source nor the destination are subscribers in this telco, the call cannot be added

(b) a new `Call` object is dynamically allocated and added to the telco's calls array

(c) if the call's source phone number corresponds to a subscriber in the telco, the new `Call` object is added to that subscriber's outgoing calls

(d) if the call's destination phone number corresponds to a subscriber in the telco, the new `Call` object is added to that subscriber's incoming calls

3.7.7. the `computeFreqCaller(string destNum)` member function computes the telephone number which is the most frequent caller to the given destination number `destNum`; the function does the following:

(a) it takes either a destination phone number or the string "all" as a parameter

(b) if the parameter is a destination phone number, then it must correspond to a subscriber in the telco, otherwise results cannot be computed

(c) if the destination is "all", then the collection of all the telco's calls must be used for computing the results

(d) if a destination phone number is specified, then the corresponding subscriber's incoming calls must be used for computing the results; **do not** use the telco's collection of calls in this case

(e) the most frequent caller is not necessarily a subscriber in the telco

(f) once the most frequent caller is computed, its information is printed to the screen, including its phone number and all the calls placed from that frequent caller to the destination number (or to all phone numbers, if "all" is specified as destination)

(g) if there is a tie between multiple callers as the most frequent, the information for all the tied callers must be printed to the screen

3.7.8. the `computeFreqCalled(string srcNum)` member function computes the telephone number which is most frequently called by the given source number `srcNum`; the function does the following:

(a) it takes either a source phone number or the string "all" as a parameter

(b) if the parameter is a source phone number, then it must correspond to a subscriber in the telco, otherwise results cannot be computed

(c) if the destination is "all", then the collection of all the telco's calls must be used for computing the results

(d) if a source phone number is specified, then the corresponding subscriber's outgoing calls must be used for computing the results; **do not** use the telco's collection of calls in this case

(e) the most frequently called number is not necessarily a subscriber in the telco

(f) once the most frequently called number is computed, its information is printed to the screen, including its phone number and all the calls placed to that frequently called number from the source number (or from all phone numbers, if "all" is specified as source)

(g) if there is a tie between multiple most frequently called numbers, the information for all the tied called numbers must be printed to the screen

3.7.9. the `print()` function prints out to the screen all the information for every subscriber in the telco, including its phone number and all its incoming and outgoing calls combined

3.7.10. **NOTE:** The `Telco` class is special in the sense that it is both an entity class and a control class. It is the class that has the most knowledge about the purpose of this program. This has special consequences when it comes to implementing the `computeFreqCaller()` and `computeFreqCalled()` functions. Specifically:

(a) **no part** of these function implementations can be done by another entity class, nor by a collection class; the `Call` class should be reusable in any program that simulates the properties of phone calls, not just the programs that compute call frequency; the `Array` class should only know about storing and manipulating data, regardless of what that data is

(b) as a result, it is bad software engineering to involve either the `Call` or the `Array` class in these computations, other than for retrieving data from them

(c) similarly, **do not** implement unnecessary getter or setter functions; however, you may implement additional "helper" member functions (not global functions) as needed

3.8. **Implement the `Control` class**

You will implement the the `Control` class with all the data members and member functions indicated in the UML class diagram; in addition:

3.8.1. the constructor will dynamically allocate and initialize the telco to be managed

3.8.2. the destructor will clean up the necessary memory

3.8.3. the `launch()` member function will do the following:

(a) call the initialization functions that have been provided in the `a2-posted.tar` file; you must use these functions, *without modification*, to initialize the data in your program

(b) use the `View` object to display the main menu and read the user's selection, until the user exits

(c) if required by the user:

(i) compute the most frequent caller in the telco; this will require using the `View` object to prompt the user for a destination phone number and to read it from the user

(ii) compute the most frequently called number in the telco; this will require using the `View` object to prompt the user for a source phone number and to read it from the user

(iii) print out all the subscribers in the telco

3.9. **Write the `main()` function**

Your `main()` function must declare a `Control` object and call its `launch()` function. The entire program control flow must be implemented in the `Control` object as described in the previous instruction, and the `main()` function must do nothing else.

3.10. **Test the program**

You must provide code that tests your program thoroughly. For this program, the use of the provided initialization functions will be sufficient. Specifically:

3.10.1. Check that the subscriber and call data is correct, and that it is printed in the correct order

3.10.2. Check that the results of computing the most frequent caller and the most frequently called number are correct and complete.

3.10.3. Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

4. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

4.1. The code must be written in C++98. It must compile and execute in the default course VM, without any libraries or packages or any software that's not already provided in the VM.

4.2. Your program must comply with the principle of least privilege, and it must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. Your program must follow basic OO programming conventions, including the following:

4.4.1. Do not use any global variables or any global functions other than `main()`.

4.4.2. Do not use `struct`s. You must use classes instead.

4.4.3. Objects must always be passed by reference, never by value.

4.4.4. Functions must return data using parameters, not using return values, except for getter functions.

4.4.5. Existing functions must be reused everywhere possible.

4.4.6. All basic error checking must be performed.

4.4.7. All dynamically allocated memory must be explicitly deallocated.

4.5. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3. **DO NOT** place inline comments in your function implementations.

5. Submission

5.1. You will submit in *cuLearn*, before the due date and time, the following:

5.1.1. One `tar` or `zip` file that includes:

(a) all source and header files

(b) a Makefile

(c) a README text file that includes:

(i) a preamble (program author, purpose, list of source and header files)

(ii) compilation and launching instructions

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

6.1. **Marking components:**

- 2 marks: correct modifications to `Date` class
- 20 marks: correct implementation of `Call` class
- 14 marks: correct modifications to `Array` class
- 10 marks: correct implementation of `Subscriber` class
- 46 marks: correct implementation of `Telco` class
- 8 marks: correct implementation of `Control` class

6.2. **Execution requirements:**

6.2.1. all marking components must be called and execute successfully in order to earn marks

6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

6.3.1. Packaging errors:

(a) 10 marks for missing Makefile

(b) 5 marks for missing README

(c) up to 10 marks for failure to correctly separate code into header and source files

(d) up to 10 marks for bad style or missing documentation

6.3.2. Major design and programming errors:

(a) 50% of a marking component that uses global variables or `struct`s

(b) 50% of a marking component that consistently fails to use correct design principles

(c) 50% of a marking component that uses prohibited library classes or functions

(d) up to 100% of a marking component where Constraints listed are not followed

(e) up to 10 marks for bad style

(f) up to 10 marks for memory leaks and errors reported by `valgrind`

6.3.3. Execution errors: 100% of any marking component that cannot be tested because it doesn't compile or execute in the provided course VM, or the feature is not used in the code, or data cannot be printed to the screen, or insufficient datafill is provided.