COMP 2404 A/C – Assignment #3

Due: Wednesday, March 10 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program, in the VM provided for this course, to match together the animals available for adoption in an animal shelter with the clients that are best suited to adopt them. Your program will simulate an animal shelter that has multiple animals for adoption, and multiple clients looking to adopt an animal. Given each animal's unique characteristics and each client's preferences for adoption, your program will compute the best matches overall, as well as the best matches for each client.

2. Learning Outcomes

With this assignment, you will:

- practice drawing a UML class diagram
- work with inheritance, composition, and linked lists

3. Instructions

3.1. Draw a UML class diagram

Your program will be separated into objects from the different object design categories, including a control object, a view object, multiple entity objects, and some collection objects. You will implement a class representing an animal shelter, which will contain a collection of animals and a collection of clients.

Each animal and client will be uniquely identified with an automatically generated identifier. To capture this commonality, both classes will derive from a class representing an identifiable object. As well, each client will contain a collection of the criteria representing that client's adoption preferences.

You will begin by drawing a UML class diagram, using a drawing package of your choice, and your diagram will follow the conventions established in the course material covered in section 2.3. A partial diagram of the program has been provided for you in Figure 1, as a starting point.

Your diagram will represent all the classes in your program that belong in a UML class diagram, including the control, view, and entity classes in the program, as well as the hierarchy of identifiable object classes. Since we do not cover "uses" associations in this course, classes that are used solely as function parameters and that are not in composition and/or inheritance associations with other classes are not shown.

Your diagram will show all attributes, all operations (including the parameters and their role such as in, out, inout), and all associations between classes, including directionality and multiplicity, where applicable. As always, do not show collection classes, as they must be implied by multiplicity, and do not show getters, setters, constructors, or destructors.

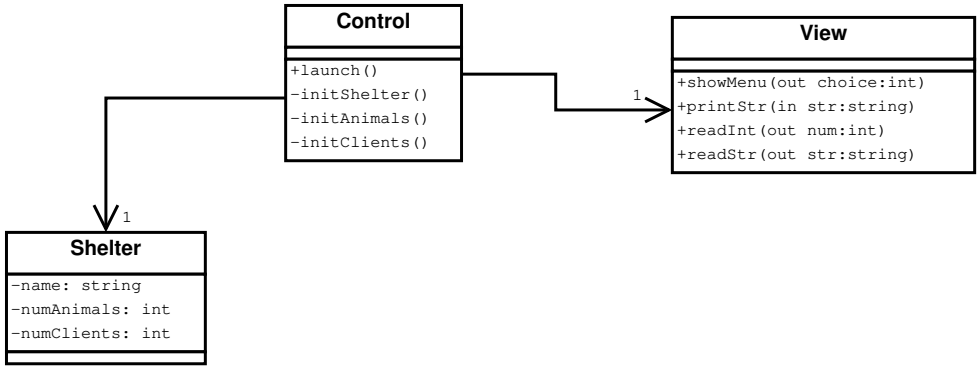

Figure 1: Partial UML class diagram

3.2. **Download and understand the posted code**

The `a3-posted.tar` file has been provided for you in *cuLearn*. This file contains the following:

3.2.1. the `View` class that your code must use for most communications with the end user

3.2.2. a skeleton `Control` class, with some data initialization member functions that your code is required to call

3.2.3. the `defs.h` header file that contains a required enumerated data type

3.2.4. the `CriteriaArray` class, which is a collection class that's based on the coding example of section 2.2, program #1, and contains a collection of `Criteria` object pointers

3.2.5. the `Criteria` class, which captures the details of a client's preference for one particular animal characteristic; this class is fundamental to the workings of this program, and the following should be noted:

(a) this class is designed to be scalable for any number of animal characteristics, and we will avoid hard-coding a specific set of characteristics as much as possible; your code must not violate this design principle

(b) each client preference, represented as one `Criteria` object, will include the following:

(i) the name of the animal characteristic (including species, breed, gender, and age)

(ii) the value of that characteristic that is preferred by the client (for example, "Dog" or "Cat" for the species characteristic); the value for the age characteristic must be stored in years

(iii) the weight that the client ascribes to that characteristic, compared to the other characteristics, as a number between 1 and 10, with 1 indicating least preferred, and 10 most preferred

(c) each client will store a collection of its preferences, as a collection of `Criteria` objects; this set of preferences will be used in a later step to compute the degree to which the client's preferences match the characteristics of each animal in the shelter

(d) a client can store at most one `Criteria` object for each animal characteristic, such as species, breed, gender, and age

(e) clients are not obligated to indicate a preference for each possible animal characteristic; where a client expresses no preference, there will be no corresponding `Criteria` object; inserting any kind of "dummy" values that are not supplied by the client would be very bad design, and will be avoided in this program

(f) **Example:** let's say a client wants to adopt an animal of species "Cat"; let's assume that the client prefers the breed "Domestic Short Hair", with a weight of 7 (out of 10), that they prefer to adopt a male animal, with a weight of 3 (out of 10), and that they have no preference for the animal's age; in our program, this client would have three `Criteria` objects in their collection:

- one criteria with the name "Species", a value of "Cat", and a weight of zero, because the species criteria is weighted differently from the other criteria when computing the matches, as we will see in a later step
- one criteria with the name "Breed", a value of "Domestic Short Hair", and a weight of 7
- one criteria with the name "Gender", a value of "M", and a weight of 3

(g) the weights of all of a client's criteria (excluding the species) must add up to exactly 10

3.3. **Implement the `Identifiable` class**

You will create a new `Identifiable` class to serve as a base class for objects that keep track of a unique id. The set of ids for each derived class will be maintained independently from each other. Specifically, the set of ids for animals will be assigned and incremented separately from the set of ids for clients.

The ids will be stored as strings, and they will use the format `X-NNNN`, where `X` is a one-character string that's unique to the derived class, and `NNNN` is a sequence of digits. For example, the animals in this program could be identified as `A-1001`, `A-1002`, `A-1003`, etc., and the clients could be identified as `C-7001`, `C-7002`, `C-7003`, etc.

The identifier for each derived object will be assigned automatically upon object creation, as we did in the coding example of section 3.1, program #6.

The `Identifiable` class will contain the following:

3.3.1. a unique identifier that's stored as a C++ standard library string
   (a) **NOTE #1:** As we saw in the course material, we cannot make this data member private, since that would make it inaccessible to the derived classes. We also cannot make it public, as that would violate the principle of least privilege. You must consider the remaining option here.
   (b) **NOTE #2:** This id will be inherited by every object of the derived classes. Do not re-declare this data member in the derived classes.

3.3.2. a constructor that does the following:
   (a) take a string and an integer reference as parameters, as follows:
      (i) the string represents the character unique to the derived class, for example `"A"` for animals and `"C"` for clients
      (ii) the integer reference is the next numerical identifier in the sequence for the derived class
      (iii) **NOTE:** Both parameters are supplied by the derived class.
   (b) use the `stringstream` class to combine together the parameters to create an identifier of the format `X-NNNN`, where `X` is the string parameter, and `NNNN` is the integer reference parameter
      (i) you will find a code snippet using the `stringstream` class here, as an example
   (c) initialize the unique identifier data member to this combined id string
   (d) increment the next id parameter

3.3.3. a getter member function for the id

3.4. Implement the `Animal` class

You will create a new `Animal` class that represents an animal that is available for adoption. This class must be derived from the `Identifiable` base class.

The `Animal` class will contain the following:

3.4.1. an integer static data member called `nextId`, to store the identifier of the next animal to be created

3.4.2. several data members representing the unique characteristics of the animal:
   (a) the animal's name, breed, gender, and colour, each represented as a string
   (b) its species, stored as a `SpeciesType`, which is defined in the posted code
   (c) its age, represented as an integer, and stored as a number of months

3.4.3. an initialization of the `nextId` data member at file scope, in a separate statement in the source file, as we did in the coding example of section 3.1, program #6

3.4.4. a default constructor that does the following:
   (a) take one parameter for each data member, except for the age, and initialize those data members
   (b) take two parameters for the age: the number of years and number of months since the animal was born; the `age` data member must be computed and stored as the total number of months since birth, using these two parameters
   (c) use *base class initializer syntax* to call the base class constructor with the correct string and the next id
      (i) an example of base class initializer syntax can be found in the coding example of section 3.2, program #1

3.4.5. getter member functions for the animal's name, breed, gender, and age

3.4.6. a `print()` member function that prints to the screen all the animal's data members, including the id; the species must be printed as an informative string, for example "Dog" or "Cat"; the age, which is stored in months, must be converted to years and months for printing

3.4.7. any additional helper member functions for formatting purposes, as needed

3.5. Implement the `Client` class

You will create a new `Client` class that represents an individual who is ready to adopt an animal from the animal shelter. This class must be derived from the `Identifiable` base class.

The `Client` class will contain the following:

3.5.1. an integer static data member called `nextId`, to store the identifier of the next client to be created

3.5.2. a data member for the client's name, represented as a string

3.5.3. a data member to store the client's collection of adoption criteria, as a `CriteriaArray` object

3.5.4. an initialization of the `nextId` data member at file scope, in a separate statement in the source file, as we did in the coding example of section 3.1, program #6

3.5.5. a default constructor that does the following:

(a) take a client name as parameter, and initialize the corresponding data member

(b) use base class initializer syntax to call the base class constructor with the correct string and the next id

3.5.6. getter member functions for the name and criteria collection

(a) remember, **do not** make a copy of the collection; it must be returned by reference

3.5.7. a `void addCriteria(Criteria*)` member function that adds the given criteria to the collection

3.5.8. a `print()` member function that prints to the screen all the client's information, including the id, name, and all the criteria information

NOTE: You must make sure that you reuse existing functions everywhere possible.

3.6. Implement the `Match` class

You will create a new `Match` class that contains the details of the match between one client and one animal, and the degree to which they are well suited to each other. The class will contain the following:

3.6.1. a data member that represents the animal involved in the match, stored as an `Animal` pointer

3.6.2. a data member that represents the client involved in the match, stored as a `Client` pointer

3.6.3. **NOTE: DO NOT** dumb down the above two data members by simply storing identifiers; they must be pointers to the actual corresponding objects

3.6.4. a data member that represents the score (out of 20) of the match between the animal and the client, stored as a `float`

3.6.5. a default constructor that takes parameters for each data member and initializes them

3.6.6. a getter member function for the score

3.6.7. a getter member function that returns the client's id

3.6.8. a `print()` member function that prints to the screen all the match information; this includes printing out the client's id and name, the animal's id and name, and the score first out of 20 points, then as the corresponding percentage

3.7. Implement the `MatchList` classes

You will implement an inheritance hierarchy of linked list collection classes, which will contain `Match` objects as data. Most of the functionality will be contained in the `MatchList` base class. The two derived classes, `MatchListByScore` and `MatchListByClient`, will use different techniques to order the data within the collections.

You will begin with the `List` class that we implemented in the coding example of section 3.1, program #7. You will rename this class to `MatchList`, and you will make the following changes:

3.7.1. you will modify the list so that it stores `Match` objects as data

3.7.2. you will change the access specifier of the nested `Node` class to protected instead of the default of private; this is necessary to allow our derived list classes to access their own nodes

3.7.3. you will change the access specifier of the `head` data member to protected, again so that the derived list classes can access it

3.7.4. you will remove the `del()` member function

3.7.5. you will remove the `add()` member function, but don't lose it completely! we will simply be moving this function out of the base class, to the derived classes

3.7.6. because the `Match` objects will be shared by multiple lists, we need to separate the deallocation of the nodes from the deallocation of the data; you must modify the destructor so that it no longer deallocates the data

3.7.7. implement a new `void cleanup()` member function that traverses the list and deallocates only the list data

You will implement two new classes called `MatchListByScore` and `MatchListByClient`, both derived from `MatchList`. The two derived classes will contain only one member function each, as follows:

3.7.8. both derived classes will contain a `void add(Match*)` member function

3.7.9. the `add()` member function in the `MatchListByScore` class will insert the given match into the correct position in the list so that it remains in *descending* (decreasing) order by the match's score

3.7.10. the `add()` member function in the `MatchListByClient` class will insert the given match into the list in the correct position so that it remains first in ascending order by client id, and within each client's matches, the list remains in descending order by the match's score

3.8. **Implement the `Shelter` class**

You will create a new `Shelter` class that represents the animal shelter and contains the functionality required for the program features.

The `Shelter` class will contain the following data members:

3.8.1. the name of the shelter, stored as a string

3.8.2. a collection of all the animals in the shelter, stored as a primitive array of `Animal` object pointers

3.8.3. a collection of all the clients of the shelter, stored as a primitive array of `Client` object pointers

3.8.4. as the two collections above are primitive arrays, best practice tells us that we must track the current number of elements in each

The `Shelter` class will contain the following member functions:

3.8.5. a default constructor that takes the required parameters and initializes all the data members

3.8.6. a destructor that deallocates the necessary dynamically allocated memory

3.8.7. an `bool add(Animal*)` member function that adds the given animal to the back of the animal collection; this function returns true if no errors occurred, and false otherwise

3.8.8. an `bool add(Client*)` member function that adds the given client to the back of the client collection; this function returns true if no errors occurred, and false otherwise

3.8.9. a `void printAnimals()` member function that prints to the screen all the data for each element in the animal collection

3.8.10. a `void printClients()` member function that prints to the screen all the data for each element in the client collection

3.8.11. a `void computeMatches(MatchListByScore&, MatchListByClient&)` member function that computes the degree to which every animal in the shelter meets the preferences of each client; this function will loop over all the clients and all the animals in the shelter, and for every client-animal pair, it will do the following:

(a) loop over the client's criteria and, starting with a match score of zero, accumulate the client's match score with that animal, for each criteria, as described in the "Score calculation" step below

(b) once the client-animal match score is computed, if the score is zero, do nothing and continue to the next client-animal pair

(c) dynamically allocate a new `Match` object for the current client and animal pair, with the accumulated match score

(d) add the new `Match` object to **both** parameter lists

3.8.12. **Score calculation:** the score for each client-animal pair is accumulated as follows:

(a) if the criteria name indicates that it's the species criteria, and if the client's preferred value matches the animal's species, add 10 points to the current score

(b) if the criteria name indicates that it's the breed criteria, and if the client's preferred value matches the animal's breed, add the criteria's weight to the current score

(c) if the criteria name indicates that it's the gender criteria, and if the client's preferred value matches the animal's gender, add the criteria's weight to the current score

(d) if the criteria name indicates that it's the age criteria, we must compute how closely matched in age the animal is to the client's preferred value, using the following algorithm:

(i) the criteria value is in years, but the animal's age is stored in months; we begin by computing the animal's age in years by performing an integer division of the animal's age by 12

(ii) compute the absolute value of the difference between the animal's age in years and the client's preferred value, and compute the floating point division of this result by 10; that will give us the age difference as a proportion of a standard animal lifetime of 10 years

(iii) if the result above is greater than 1, meaning that the age difference is greater than 10 years, set the result to 1 instead; let's call the resulting proportion `prop`

(iv) subtract the resulting proportion from 1, and multiply this value by the criteria's weight; so assuming the criteria's weight is `w`, add to the current score the following: `w * (1 - prop)`

3.9. Implement the `Control` class

You will implement the `Control` class as follows:

3.9.1. the data members will be as shown in the partial UML class diagram

3.9.2. the constructor will dynamically allocate the animal shelter to be managed

3.9.3. the destructor will clean up the necessary memory

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

(a) call the initialization functions that have been provided in the `a3-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 and print out the match scores for all clients and animals, both overall in order by match score, and in order by client and their match scores; this requires the declaration of two temporary match linked lists, the population of these linked lists with computed matches, the printing of both lists, and the clean up of the match list data

(ii) print out all the animals in the shelter

(iii) print out all the clients of the shelter

3.10. 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.

3.11. 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:

3.11.1. Check that the animal and client data is correct, and that it is printed in the correct order.

3.11.2. Check that the results of computing all the animal and client matches are correct, complete, and printed in the correct order.

3.11.3. 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. A UML class diagram (as a PDF file), drawn by you using a drawing package of your choice, that corresponds to the entire program design.

5.1.2. 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:**

- 22 marks: UML class diagram
- 21 marks: correct implementation of `Identifiable`, `Animal`, and `Client` classes
- 7 marks: correct implementation of `Match` class
- 10 marks: correct implementation of `MatchList` classes
- 30 marks: correct implementation of `Shelter` class
- 10 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.