COMP 2404 -- Assignment #2

Due: Tuesday, March 10, 2020 at 12:00 pm (noon)

Goal

For this assignment, you will write a C++ program to manage the data for an agricultural cooperative. You will implement your program using objects from the different object design categories, based on a UML diagram provided for you.

Learning Outcomes

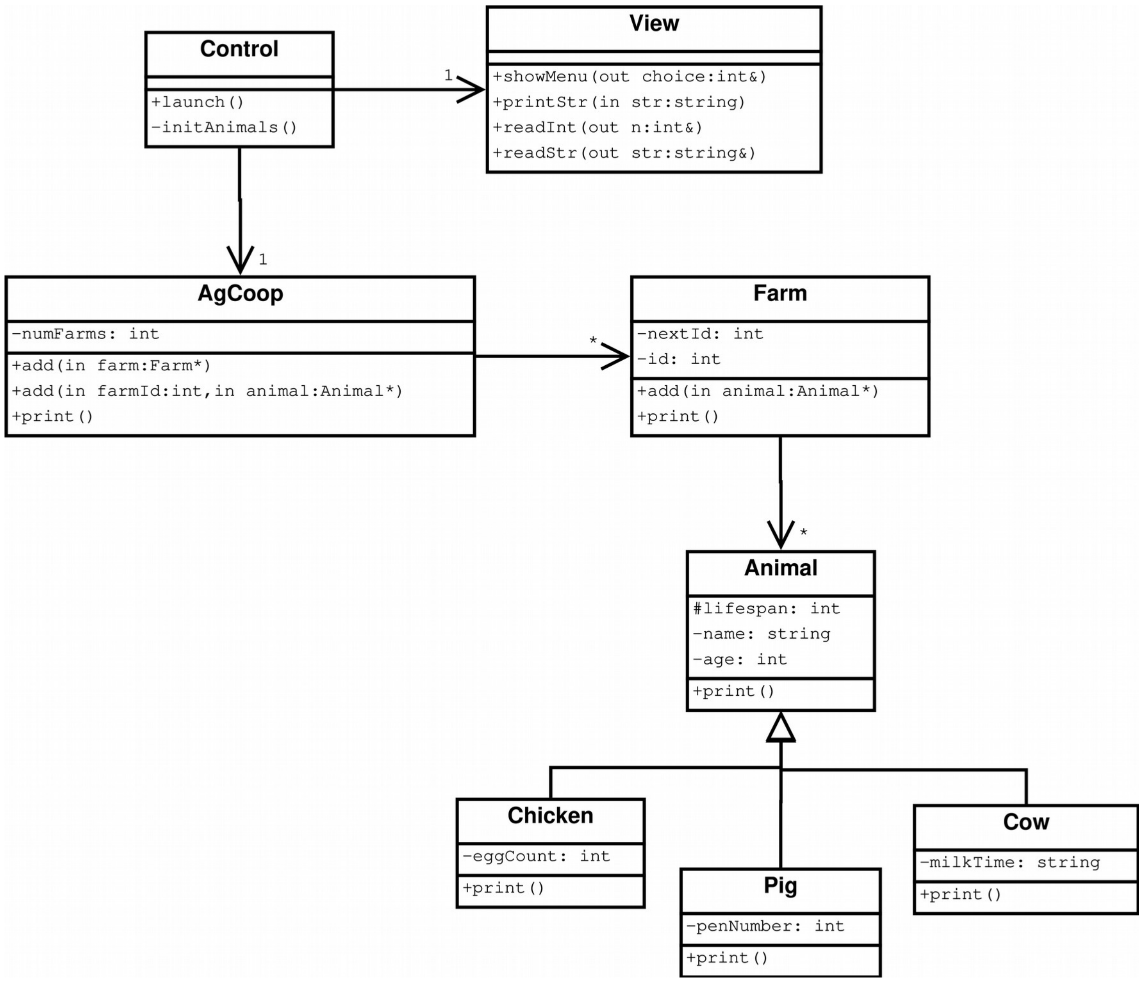
With this assignment, you will:

- practice implementing a design that is given as a UML class diagram
- implement a program separated into control, view, entity, and collection objects
- work with inheritance and composition, as well as linked lists

Instructions

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.



2. **Implement the `Animal` derived classes**

   You will begin with the `Animal` and `Chicken` classes that we worked on during the lectures. You can find these classes in the coding examples posted in *cuLearn*, in section 3.2, program #1 (`S3.2.Inherit/p1-basic`). You will be adding two additional classes, both derived from `Animal`.

   You will implement the new `Cow` class as derived from `Animal`. The `Cow` class will contain the following:

   (1) a data member to represent the time at which the cow must be milked, stored as a string in the format HH:MM, where HH represents the hours and MM the minutes

   (2) a constructor that initializes all the data members, including those inherited from the `Animal` class; you must use *base class initializer syntax* to do this
      (a) see the coding examples in section 3.2, program #1, for an example of base class initializer syntax

   (3) a `print()` function that prints out all the data members; the members inherited from `Animal` must be printed using the `Animal` class's `print()` function

   You will implement the new `Pig` class as derived from `Animal`. The `Pig` class will contain the following:

   (1) a data member to represent the pen number where the pig lives, stored as an integer

   (2) a constructor that initializes all the data members, including those inherited from the `Animal` class; you must use base class initializer syntax to do this

   (3) a `print()` function that prints out all the data members; the members inherited from `Animal` must be printed using the `Animal` class's `print()` function

3. **Modify the `List` class**

   You will modify the `List` class that we implemented in the coding example of section 3.1, program #7. The `List` and `Node` classes must be modified to use `Animal` pointers as data, and the `add()` function must add each animal in descending (decreasing) order by age.

4. **Implement the `Farm` class**

   You will create a new `Farm` class that manages a collection of animals. The `Farm` class will contain the following:

   (1) a data member for the unique identifier of the farm; this will be represented as an integer, and it must be generated by the constructor when a farm object is created

   (2) a static data member called `nextId`, to store the identifier of the next farm to be created
      (a) see the coding example in section 3.1, program #6, for an example of how identifiers can be generated and assigned using a static data member

   (3) a data member that is a linked list of `Animal` pointers; you must use the `List` class modified in a previous step

   (4) a constructor that initializes the farm identifier from the next available id and increments this next available id for the next farm to be created

      NOTE: The next available id will have to be initialized as a separate statement at file scope in the source file, as we did in the coding example of section 3.1, program #6.

   (5) a getter member function for the farm id

   (6) a `void add(Animal* a)` member function that adds the given animal `a` to the linked list, in descending order by age; you must reuse existing functions here

   (7) a `void print()` member function that prints to the screen the farm id, and prints the data for all the animals of the farm; you must reuse existing functions for this

5. Implement the `AgCoop` class

You will create a new `AgCoop` class that manages a collection of farms. The `AgCoop` class will contain:

(1) a data member that holds a collection of farms
  - (a) this will be a *statically allocated array* of `Farm` object **pointers**
  - (b) you will define a preprocessor constant for the maximum number of farms; this can be set to a reasonable number such as 16 or 32
  - (c) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

(2) a data member to track the current number of farms in the array

(3) a constructor that initializes the current number of farms

(4) a destructor that deallocates the dynamically allocated farms contained in the array

(5) a `void add(Farm* f)` member function that adds the given farm `f` to the back (the end) of the farm array; you must perform all basic error checking

(6) a `void add(int id, Animal* a)` member function that adds the given animal `a` to the farm with the unique identifier matching the parameter `id`
  - (a) you must perform all basic error checking
  - (b) you must reuse existing functions everywhere possible

(7) a `print()` member function that prints out all the data in every `Farm` object contained in the array

6. Implement the `Control` class

You will create a new `Control` class that implements the control flow for the entire program. The `Control` class will contain the following:

(1) a data member for the `AgCoop` object that represents the agricultural cooperative to be managed

(2) a data member for the `View` object that will be responsible for most user I/O
  - (a) the `View` class is provided for you and can be downloaded from *cuLearn* in the `a2-posted.tar` file

(3) an `initAnimals()` member function that initializes the farms and animals contained in the agricultural cooperative; this function will do the following:
  - (a) dynamically allocated at least five (5) farm objects and add them to the `AgCoop` object
  - (b) dynamically allocate at least 10 different animals of each of the following kinds: `Chicken`, `Cow`, and `Pig`, and add them to the different farms in the `AgCoop` object; every animal must have unique data; the ordering must test the `add()` function thoroughly

(4) a `launch()` member function that implements the program control flow and does the following:
  - (a) call the `initAnimals()` function to initialize the data in the `AgCoop` object
  - (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:
    - ▪ use the `View` object to read from the user the farm id, the name, the age, and the lifespan of the animal to be created
    - ▪ depending on the type of animal to be created, as selected by the user, read from the user the additional information required; for example, for a new `Chicken` object, the egg count must be provided by the user
    - ▪ create the new `Chicken`, `Cow`, or `Pig` object from the information specified by the user
    - ▪ add the new object to the `AgCoop` object
  - (d) once the user chooses to exit, print the content of the `AgCoop` object to the screen

  NOTE: A skeleton `launch()` function is provided for you and can be downloaded from *cuLearn* in the `a2-posted.tar` file. If you use this skeleton function, you must add the code required so that the function performs all the tasks described above.

7. 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.

8. Test the program

(1) make sure that the animal data you provide exercises all your functions thoroughly; failure to do this will result in **major deductions**, even if the program appears to be working correctly

(2) check that the animal information is correct when it is printed at the end of the program

(3) make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used; use `valgrind` to check for memory leaks

NOTE: You will notice a very interesting phenomenon here! Because the list of animals in each farm is stored as an `Animal` pointer, only the `Animal` class's `print()` function is called, and not the individual animal's `print()` function from the `Chicken`, `Cow`, or `Pig` classes. Getting this to work correctly requires *polymorphism*, which we haven't yet covered in class. For the correct function to be called, you must change all the `print()` member function prototypes in the `Animal` class definitions to: `virtual void print() const;` We will look at why this works in section 3.4.

Constraints

1. your program must follow correct encapsulation principles, including the separation of control, UI, entity, and collection object functionality
2. do not use any classes, containers, or algorithms from the C++ standard template library (STL)
3. do not use any global variables or any global functions other than `main()`
4. do not use structs; use classes instead
5. objects must always be passed by reference, never by value
6. functions must return data using parameters, not using return values, except for getter functions
7. existing functions must be reused everywhere possible
8. all basic error checking must be performed
9. all dynamically allocated memory must be explicitly deallocated
10. your classes must be thoroughly documented in every class definition, as discussed in the course material, section 1.3

Submission

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

- one **`tar`** or **`zip`** file that includes:
  - all source and header files, including the code provided
  - a Makefile
  - a README file that includes:
    - a preamble (program and revision authors, purpose, list of source/header/data files)
    - compilation and launching instructions

NOTE: Do **not** include object files, executables, duplicate files, or unused files in your submission.

Grading (out of 100)

Marking components:

- 12 marks: correct implementation of `Animal` classes
- 18 marks: correct implementation of `Farm` class
- 40 marks: correct implementation of `AgCoop` class
  - 7 marks: correct class definition
  - 2 marks: correct implementation of constructor
  - 5 marks: correct implementation of destructor
  - 4 marks: correct implementation of `print()` function
  - 8 marks: correct implementation of `add(Farm*)` function
  - 14 marks: correct implementation of `add(int, Animal*)` function
- 30 marks: correct implementation of `Control` class

Execution requirements:

- all marking components must be called, and they must execute successfully to receive marks
- all data handled must be printed to the screen for marking components to receive marks

Deductions:

1. Packaging errors:
   (1) 10 marks for missing Makefile
   (2) 5 marks for a missing README
   (3) 10 marks for consistent failure to correctly separate code into source and header files
   (4) up to 10 marks for bad style or missing documentation
2. Major programming and design errors
   (1) 50% of a marking component that uses global variables, or structs
   (2) 50% of a marking component that consistently fails to use correct design principles
   (3) 50% of a marking component that uses prohibited library classes or functions
   (4) 100% of a marking component that is *replaced* by prohibited library classes or functions
   (5) 50% of a marking component where unauthorized changes have been made to the provided code
   (6) up to 100% of a marking component where Constraints listed are not followed
   (7) up to 10 marks for memory leaks
   (8) up to 10 marks for bad style
3. Execution errors
   (1) 100% of a marking component that cannot be tested because it doesn't compile or execute in VM
   (2) 100% of a marking component that cannot be tested because the feature is not used in the code
   (3) 100% of a marking component that cannot be tested because data cannot be printed to the screen
   (4) 100% of a marking component that cannot be tested because insufficient datafill is provided