CPSC 121, OOP – Fall 2020
Pet Animal: Object-Based Programming Concepts Homework





Learning Goals:

- Object Analysis
- C++ class and object concepts and syntax
- Class constructors, destructors, and object construction & initialization
- Object composition
- Operator overloading
- Familiarization with Scope, Linkage, Lifespan
- Separation of interface, implementation, and client Code

Description:

Working within your groups, you will pick a specific animal that best personifies your group's personality. You will then create a C++ class that captures that concept. For example, if your team is wise and scholarly you may think an owl captures your team's personality and you would then create a C++ class called Owl. You will play both class consumer and class designer roles. As class consumer, you will script a scenario where you watch the birth of such an animal and adopt it as your pet. You'll, for example, give it a name, play with it, take it to the vet when it gets sick, and eventually as all pets do witness its death. And maybe you'll decide your pet is lonely and you'll adopt another. What happens after birth and before death is up to you, but as a minimum there are some prescribed actions you'll need to accommodate. After you decide what commands you want your pet to react to, you will switch roles from class consumer to class designer and define the class interface for your specific animal. Then finally you will implement the interface and build an executable program.

How to Proceed:

- Agree on an animal, then start writing pseudocode in function main() in main.cpp.
    - Create an instance or two of your animal and pretend to adopt as pets. Decide what parameters should be given during construction
        - Hint: there are usually 3 or 4 different ways you'll want to construct your pet animal. With or without name, is it a newborn or are you adopting an older pet (age), color, etc
        - Hint: Lifespan and curly braces are related. Create a set of curly braces, construct an object within the curly braces, play with it, and then watch the object be destroyed as the closing curly brace gets executed.

  - Tell your pet(s) to do tricks (e.g., call member functions on your objects). Decide what parameters to pass, e.g. speak(loudly) or speak (softly, twice). You decide what tricks to teach your pet.

- Next, define the class interface in a header file. This should be syntactically correct and should capture all the behavior pseudo coded above.
  - The behavior exposed to the client (construction, hop, speak, other tricks, etc) is part of the class's public interface.
  - You'll likely discover new or modified behaviors, parameters, etc. and that's okay. Just update your pseudocode in main() accordingly.
  - You will reuse and build upon this class in the next project making it important to get the function prototypes (return type, function name, number and type of arguments, and the order of the arguments) correct. An example is provided below.

- Now go back to your client code, include the header file, and convert your pseudocode into real syntactically correct C++ code. At the end of this step main.cpp should compile clean and exercise all capability. You can't execute yet, but be sure you get a clean compile before moving on to the next step.

- Finally, implement the member functions of the header file in the corresponding source file.
  - Now is the time you will discover what information, or state, each instance of your animal should maintain. Add private instance attributes to the class's definition in the header file, just don't modify the public part of the definition.
  - At the end of this step you should be able to compile the animal's source file cleanly: Owl.cpp in this example.

- Now put it all together and compile your complete program (main.cpp, Address.cpp, and Owl.cpp) creating an executable image (file). Run, debug, repeat.

Project Rules and Constraints:

- The name of your class should be the name of your animal, e.g., Owl. (No, you can't pick an owl, I've already used that one)
  - Best Practice: Classes start with an upper-case character

- Separate the class definition from the implementation
  - Best Practice: create .hpp (not .h) header and .cpp source files

- Clients (e.g. function main()) of your class must be able to perform at least the following operations:
  - Construct a newborn animal (e.g., Owl) as their pet with zero, one, or two arguments:
    - If one argument is provided it must be the pet's name
    - If two arguments are provided, the first must be the pet's name and the second the pet's address
  - Clone their pet using the copy constructor
  - Assign their pet using the assignment operator
  - Retrieve or modify their pet's name with these functions:
    - `std::string name() const;`
    - `void name(std::string_view newName);`
    - Best Practice: function and object names start with a lower-case character

  - Retrieve their pet's gender as an enumeration with this function:
    - `Gender gender() const;`
  - Retrieve or set their pet's address with these functions:
    - `Address address() const;`
    - `void address( const Address & newAddress);`
  - Retrieve their pet's government issued ID with this function:
    - `std::size_t id() const;`
  - Tell their pet to hop with this function:
    - `int hop();`
  - Sort and compare their pets using the 6 relational operators
  - Read and write their pet using the extraction and insertion operators

- As class designer and implementor, implement the functions above as follows:
  - Construction requirements:
    - Randomly (50/50) set the gender to either "Boy" or "Girl"
      - Gender is an enumerated type with values "Boy" and "Girl"
      - Do not use the horrible C-inherited rand() or srand() functions, be sure to use the C++ random number generation tools
      - Once initialized, this cannot be changed
    - Give him/her the next available unique government issued, sequential ID
      - Once initialized, this cannot be changed
      - The type of the ID must be the biggest unsigned integral type available
      - Make "next available ID" a class (vice instance) attribute
    - If no name is given, default his/her name to "Baby boy (or girl) ID". For example, "Baby boy 8354" or "Baby girl 8355".
    - If no address is given, default his/her address to a default constructed Address
  - Copy construction requirements:
    - Initialize each attribute's value to the corresponding attribute's value of the original object
  - Copy assignment operator requirements:
    - Replace each attribute's value with the corresponding attribute's value of the object on the right-hand side of the equal sign.
  - Function `hop()` requirements:
    Move forward or backwards a random number of squares as indicated below. Have function `hop()` simply return the number of squares moved.
    - 10% of the time advance 3 squares (e.g., return 3)
    - 30% of the time advance 1 square
    - 15% of the time retreat 1 square (e.g., return -1)
    - 10% of the time retreat 2 squares
    - 35% of the time ignore your master and do not move
  - Relational operator requirements:
    - Two pets are equal if and only if all instance attributes are equal

        - Pets are to be sort by
            - Name, then
            - Address, then
            - Gender, then
            - ID
        - To determine if one pet is less than the other, first look at the names. If the names are not equal then pet A is less than pet B if pet A's name is less than pet B's name. If the names are equal, move on to the address. If the addresses are not equal, then pet A is less than pet B if pet A's address is less than pet B's address. And so on.
    - Insertion and extraction operator requirements
        - Operations must be symmetrical. That is, you should be able to read what you write, and you should be able to write what you read.
        - Insert (write) the pet's name followed by address, gender, and finally ID.
        - Extract (read) the pet's name followed by address, gender, and finally ID.
            - Update the caller's object only after you are sure you encountered no errors during the extraction process and have validated your data.

- Additional requirements. You are to …

    - teach your pet two new tricks. That is, you need to choose then add two more functions (verbs) to your class. For example, teach your pet to speak on command (add a speak() function as mentioned above), or teach your pet to play fetch, or open her wings, puff his chest, roll over, find your slippers, and so on. Be sure to show off these tricks by calling these functions in main().

    - have your new tricks either return a value or change the state of your pet (change the value of an instance attribute).

    - Ensure your member functions write nothing. Either add queries to your class or have your mutator (verb) functions return an object. Then make these enquiries and/or issue these commands from the client (that is, from function main()), then have main()write to std::cout.

    - It's likely you'll add instance attributes to your class to help remember when the last time she ate, or how old she is, or his birth date, or whatever. Be sure to account for these additional attributes in your insertion and extraction operators, and your relational operations.

- You are given a complete Address class. Use it as both an example and as your pet's address type.

- Create a client program that uses the Animal and Address classes described above. Be sure to demonstrate all the things one can tell your animal to do, including
    - Comparing (all 6 relations)
    - Inserting (writing) and extracting (reading)
    - All the functions including all permutations of overloaded functions

Reminders:

- The C++ using directive `using namespace std;` is **never allowed** in any header or source file in any deliverable product. Being new to C++, you may have used this in the past. If you haven't done so already, it's now time to shed this crutch and fully decorate your identifiers.

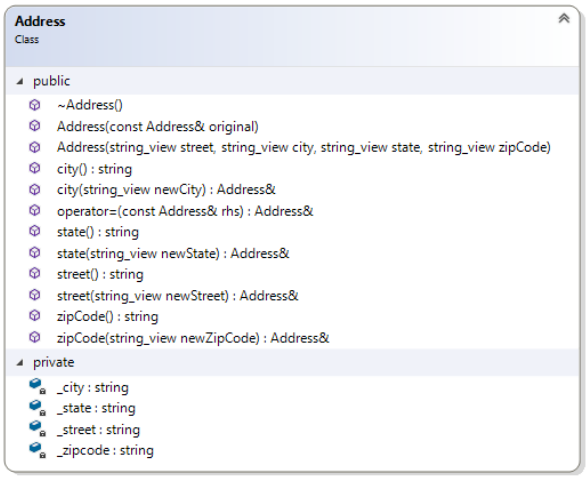
- Do not use the old, deprecated standard library functions rand() and srand() to generate random numbers. Use be sure to use std::uniform_int_distribution() instead. See the Pseudo-Random Number Generation Library.
- Use Build.sh on Tuffix to compile and link your program. There is nothing magic about Build.sh, all it does is save you (and me) from repeatedly typing the very long compile command and all the source files to compile. Using Build.sh is not required, but strongly encouraged. The grading tools use it, so if you want to know if you compile error and warning free (required to earn any credit) than you too should use it.
- Filenames are case sensitive, both in source code and in your OS file system. Windows doesn't care about filename case, but Linux (the system grading your work) does.
- You must redirect standard input from a text file (BowlingTeams.dat), and standard output to text files (output_run1.txt and output_run2.txt). Failure to include both output.txt files in your delivery indicates you were not able to execute your program and will be scored accordingly. A screenshot of your terminal window is not acceptable. See *I/O Redirection* and *How to Build and Run Your Programs with Build.sh* in Tom's Survival Guide in Canvas.

Deliverable Artifacts:

Provided files	Files to deliver	Comments
example_main.cpp	1. main.cpp	Create and deliver your main.cpp as describe above. You can use the example main.cpp file for inspiration and reference.
YourSpecificAnimal.hpp YourSpecificAnimal.cpp	2. <YourSpecificAnimal>.hpp 3. <YourSpecificAnimal>.cpp	Create and deliver these files as describe above where <YourSpecificAnimal> is replace by the name of your specific animal. Very light outlines have been provided to get you started. The Gender enumeration class, however, is fully functional and you should keep and use that directly.
Address.hpp Address.cpp	4. Address.hpp 5. Address.cpp	Use but do not modify these provided files. It's important that you deliver complete solutions, so even though you didn't modify them, be sure to include them in your delivery.
	6. output_run1.txt 7. output_run2.txt	Run your program twice. Capture the output of each run to these text files and include it in your delivery. Failure to deliver these files indicates you could not get your program to execute.
sample_output-1.txt sample_output-2.txt		These are sample runs corresponding to the example_main.cpp above. You don't need to do anything with these, but thought they might help decide what to print.

Example Interface

Address.hpp



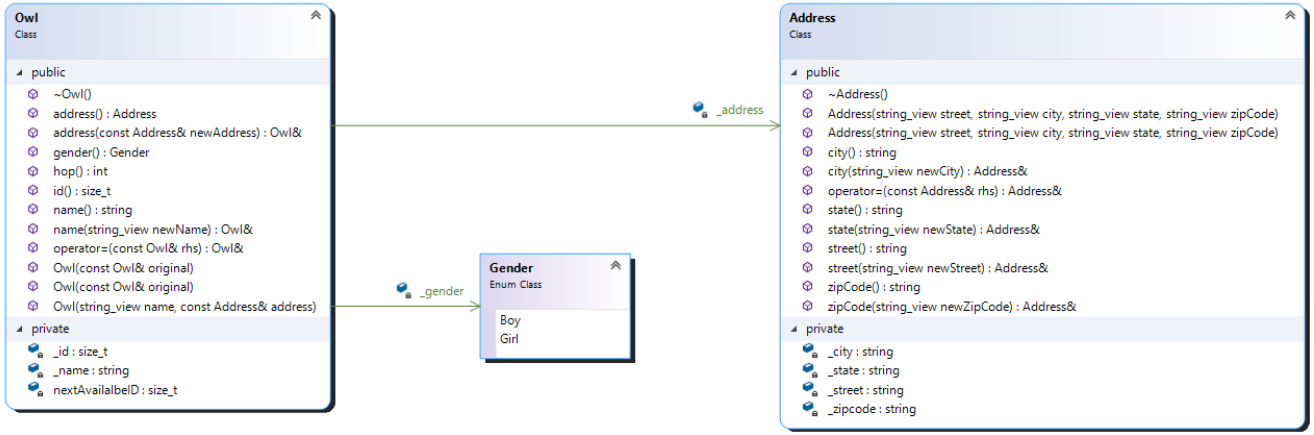

```
1  class Address
2  {
3    // Befriending non-member functions gives those non-member functions access private class members
4    // Insertion (write) and extraction (read) operators
5    friend std::ostream & operator<<( std::ostream & outputStream, const Address & address );
6    friend std::istream & operator>>( std::istream & inputStream,        Address & address );
7
8    // Relational operators need access to private members
9    friend bool operator==( const Address & lhs, const Address & rhs );
10   friend bool operator< ( const Address & lhs, const Address & rhs );
11
12   // Public interface definition
13   public:
14     // Constructors, destructor and copy assignment operator
15     Address            ( const Address & original );       // Copy constructor
16     Address & operator=( const Address & rhs );            // Copy assignment operator
17     Address( std::string_view street  = {},                // Default and up to 4 argument constructor
18              std::string_view city    = {},                //   the empty braces is a default constructed object
19              std::string_view state   = {},                //   which in this case is the empty string
20              std::string_view zipCode = {} );
21    ~Address();                                             // Destructor
22
23     // Queries
24     std::string street () const;                           // Returns the current street
25     std::string city   () const;                           // Returns the current city
26     std::string state  () const;                           // Returns the current state
27     std::string zipCode() const;                           // Returns the current zipcode
28
29
```

```
    // Mutators
    Address & street ( std::string_view newStreet );       // Updates the current street
    Address & city   ( std::string_view newCity   );       // Updates the current city
    Address & state  ( std::string_view newState  );       // Updates the current state
    Address & zipCode( std::string_view newZipCode);       // Updates the current zipcode

  // Private, implementation details not exposed to class clients
  private:
    std::string _street;
    std::string _city;
    std::string _state;
    std::string _zipcode;
};

// Address's Relational operators as non-member functions.  These are still part of the public interface
bool operator==( const Address & lhs, const Address & rhs );
bool operator!=( const Address & lhs, const Address & rhs );
bool operator< ( const Address & lhs, const Address & rhs );
bool operator<=( const Address & lhs, const Address & rhs );
bool operator> ( const Address & lhs, const Address & rhs );
bool operator>=( const Address & lhs, const Address & rhs );
```

Owl.hpp


```
1  enum class Gender {Boy, Girl};
2  std::ostream & operator<<( std::ostream & s, const Gender & gender );
3  std::istream & operator>>( std::istream & s,       Gender & gender );
```

Note: for now, put this enumeration definition in with your animal's class definition in your animal's header file.




```
4  class Owl
5  {
6    // Befriending non-member functions gives those non-member functions access private class members
7    // Insertion (write) and extraction (read) operators
8    friend std::ostream & operator<<( std::ostream & outputStream, const Owl & owl );
9    friend std::istream & operator>>( std::istream & inputStream,        Owl & owl );
10
11   // Relational operators need access to private members
12   friend bool operator==( const Owl & lhs, const Owl & rhs );
13   friend bool operator< ( const Owl & lhs, const Owl & rhs );
14
15   // Public interface definition
16   public:
17     // Constructors, destructor and assignment operators
18     Owl();                                                  // Default constructor
19     Owl( std::string_view name, const Address & address = {} );  // One or two argument constructor
20                                                             //   also a conversion constructor from any string type to an Owl
21     Owl            ( const Owl & original );                // Copy constructor
22     Owl & operator=( const Owl & rhs      );                // Copy assignment operator
23    ~Owl();                                                  // Destructor
24
```

```
    // Queries
    std::string name   () const;                                    // Returns the current name
    Gender      gender () const;                                    // Returns the current gender
    Address     address() const;                                    // Returns the current address
    std::size_t id     () const;                                    // Returns the unique identifier

    // Mutators
    Owl & name( std::string_view newName );         // Updates the current name
    Owl & address( const Address & newAddress );    // Updates the current address

    int hop();                                                      // moves and returns a random number of squares forward or backward

    // More tricks your pet can perform
    // ...

  // Private, implementation details not exposed to class clients
  private:
    // Instance Attributes
    Gender      _gender = Gender::Boy;  // Once set, gender cannot be changed
    std::size_t _id     = 0;            // Once set, the ID cannot be changed
    std::string _name;                  // Programming note: must be physically after _id and _gender
    Address     _address;

    // Class Attributes
    inline static std::size_t nextAvailalbeID = 1'350'001; // "inline static" creates and initializes nextAvailalbeID for all Owl objects to use
};

// Owl's Relational operators
bool operator==( const Owl & lhs, const Owl & rhs );
bool operator!=( const Owl & lhs, const Owl & rhs );
bool operator< ( const Owl & lhs, const Owl & rhs );
bool operator<=( const Owl & lhs, const Owl & rhs );
bool operator> ( const Owl & lhs, const Owl & rhs );
bool operator>=( const Owl & lhs, const Owl & rhs );
```