Clever Crow

In this quest you get to implement an entire class.

This class is called `Pet`. In addition to implementing this class, you also get to implement a fun little function to make up names for the millions of pets you will be able to create.

Your first miniquest - Make a name for yourself

This is just warm up before you get to implement a class.

You must implement:

```
string make_a_name(int len);
```

When I invoke this method, I will supply it a `len` parameter which tells you my desired name length.

You must then algorithmically manufacture a name and return it to me.

Before you start, read about the `rand()` function (and `srand()`) in the book, modules or reference material online. Ask questions in the forums to clarify if necessary.

Since I need to match up names generated by me and you to make sure your implementation is to spec, there are a few precautions you have to follow in implementing this function.

Specifically, you must:

- define your vowels to be the characters in the string "aeiou"
- define your consonants to be the characters in the string "bcdfghjklmnpqrstvwxyz"
- The name you return must be created by alternately selecting a random letter from each of the above two strings.
- The very first letter of your name must be either a vowel or consonant chosen using the condition (rand() % 2 == 0). Specifically, if your random is even, then the first letter of the name must be a consonant.
- To select the index of the letter to use in each iteration, you must invoke `rand()` exactly once. For example: `rand() % consonants.length()`
- **IMPORTANT**: You must NOT invoke `srand()` anywhere in your code.

Now you must implement the Pet class as specified below and incorporate `make_a_name()` into it as a public static helper method.

The Pet Class

Your Pet object ought to contain the following four private members of which one is also static (static means that there is only one copy of the variable and it is attached to the class definition - the blueprint of the class - rather than several copies, one in each object created from that class definition. Discuss this in the forums to understand more clearly if necessary.)

- `string _name;`
- `long _id;`
- `int _num_limbs;`
- `static size_t _population;`

Note that I have prefixed each member's name with an underscore. This is a convention some programmers follow to help fellow programmers instantly spot private members in code manipulating an object. Try this strategy in this lab and see if it helps. My starter code will be set up to help you do it this way. It has the nice side-effect that you can now freely use the words `name`, `id` and `num_limbs` as names for your local variables without needing to use `this->` to disambiguate.

Miniquest 2 - Pet class constructors and destructor

Implement the default and non-default constructors of your Pet class as follows:

```
Pet(string name = "", long id = -1, int num_limbs = 0);
```

By providing the default values for your members in the declaration, you can skip defining an explicit default constructor. Your implementation of the above constructor gives you both the default and non-default varieties. Both will be tested. Make sure to retain the exact default values you see above.

```
~Pet();
```

The public destructor of the Pet object, with the above signature, will be automatically invoked by the compiler when the memory of a Pet object is to be released back into the resource pool. Most of the time you don't need to implement explicit destructors for simple tasks. In this case, we're using it because it gives us an opportunity to experiment with static (or class) variables. Read up what a class variable is and how it is different from regular class members.

Your destructor must decrement the value of your `_population` member - Just like constructor should increment it. Make sure that after you create and destroy an arbitrary number of Pet objects your population count always shows 0.

Miniquest 3 - Getters (aka accessors)

Implement getters for each of your members. They should simply return the requested value while leaving the object untouched.

To accidentally prevent this from happening, we often annotate such methods with the **`const`** keyword (purple below). This tells the compiler to have our back. It will look out for us if we do something stupid like make getters (or functions it calls) change an object's properties.

- `string get_name() const;`
- `long get_id() const;`
- `int get_num_limbs() const;`

Miniquest 4 - Setters (aka mutators)

Implement setters for each of the three members as follows:

- `bool set_name(string name);`
- `bool set_id(long id);`
- `bool set_num_limbs(int num_limbs);`

Note that each setter returns a boolean value. Whenever possible, I recommend you make your setters return a success/failure value so the caller can know whether they succeeded. I think it is usually overkill to throw an exception from a setter. So this habit will serve you well even after you learn about exceptions and start coding advanced structures.

Each setter needs to validate its input as follows:

- `set_name()` should refuse to set empty strings as names
- both `set_id()` and `set_num_limbs()` should refuse to set negative numbers.

Miniquest 5 - Stringification

Implement the method

```
string Pet::to_string() const;
```

When called on a Pet object it should return a string representation of the object formatted *exactly* as follows:

```
"(Name: [NAME], ID: [ID], Limb Count: [NUMBER OF LIMBS])"
```

where the parts in square brackets need to be replaced by the values of the members in the Pet object on which `to_string()` was invoked. To help you format properly I colored in the spaces in the string above. Each gray rectangle denotes exactly one space.

Miniquest 6 - Get a whole bunch of pets at once

Implement the method

```
static void get_n_pets(size_t n,
                       std::vector<Pet>& pets,
                       int name_length);
```

When this method returns the vector `pets` must be appropriately resized and each of its elements must be a `Pet` object. Eventually I'll figure out a better way to test this method, but for now, I'm afraid I have to choreograph much of its dance. Refer to the starter code for this function and complete it as directed.

In essence, it creates the requested number of pets, assigning them strictly increasing IDs and random names of the requested length.

We'll take advantage of this default ordering in the next quest.

Miniquest 7 - Population control

Make sure your constructors increment your population count and destructors decrement it. If you have your population under control you get extra points.

Miniquest 8 - Implement Equality for all

Implement

```
bool operator==(const Pet& pet1, const Pet& pet2);
```

See how c++ lets you redefine the functionality of basic operators you have grown accustomed to and love. Complete this global function to make it return `true` if and only if an object's components are each equal to the corresponding components of the object to which it is being compared. Note that this functionality is defined as a global function, not as an object method. What are the advantages and disadvantages of doing it each way? Discuss in the forums.

Miniquest 9 - Use equality to define what inequality is

Implement

```
bool operator!=(const Pet& pet1, const Pet& pet2);
```

However, don't implement the logic for it independently. You must define it in terms of the equals operator you defined earlier.

Miniquest 10 - Give everything a voice

Implement

```
ostream& operator<<(ostream& os, const Pet& pet) {
```

c++ even lets you define the functionality of the output (some call it insertion) operator (<<). When someone issues a statement like:

```
std::cout << my_pet <<std::endl;
```

where `my_pet` is a variable of type `Pet`, then the code in this function will be executed. Since you already have a `to_string()` defined from before, you know exactly what to do here.

Note that this function returns the output stream passed to it as a parameter. That is what allows us to chain output statements like this:

```
std::cout << a << b << c << . . .
```

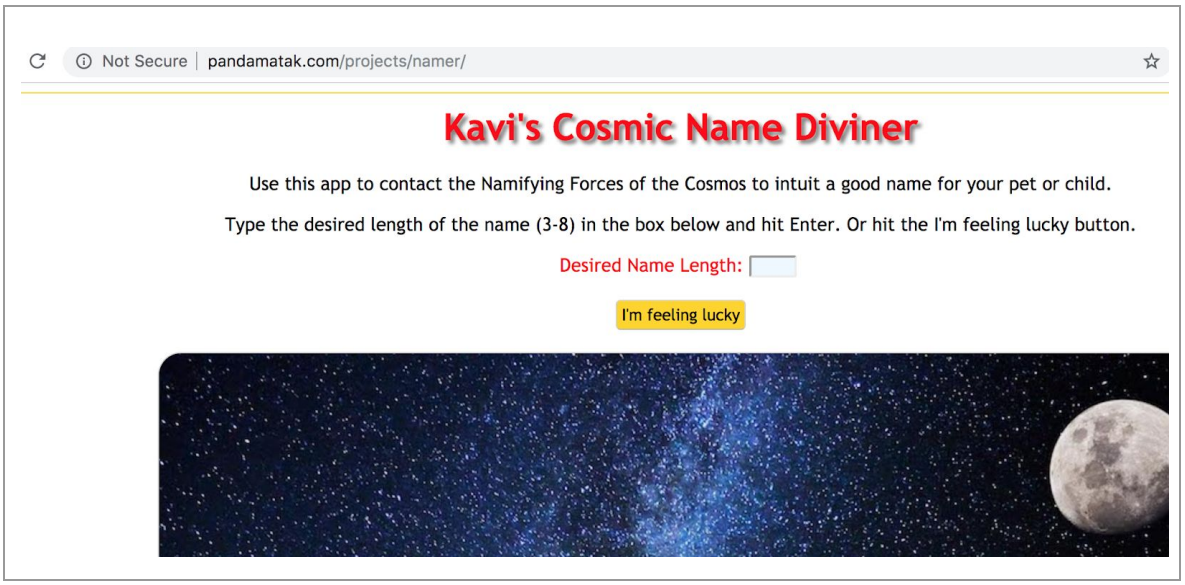
Important implementation note

Unfortunately, I have to impose this constraint. Terribly sorry about that.

You may not call `srand()` anywhere in your submitted code, and you have to adhere to the very strict guidelines about exactly where your code is allowed to call `rand()`.

Obviously you have to call these functions in your own testing code. Just make sure that they're not in the code you submit.

Finally, keep in mind that none of these methods is allowed to talk to the user. If you try to print something to the console or read from it in your submitted code, it will likely fail a whole bunch of test cases.



http://pandamatak.com/projects/namer

One of my former CS2A students ported the `make_a_name()` function to a Javascript based web-app that I hosted on my server. You can check it out at the above URL.

Starter code

You will submit two files: `Pet.h` and `Pet.cpp` - The `.h` file should contain the definition of your Pet class. The `.cpp` file should contain its implementation. Usually these two files go together.

Here is your `Pet.h` starter code

```
// Student ID: 12345678
// TODO - Replace the number above with your actual student ID
//
//  Pet.h
//
// Pets with an ID, a unique name and number of limbs.
//
// The Pet class also supports a static population member

#ifndef Pet_h
#define Pet_h

using namespace std;

class Pet {
private:
    string _name;
    long _id;
    int _num_limbs;

    static size_t _population;

public:
    Pet(string name = "", long id = -1, int num_limbs = 0);
    ~Pet();

    string get_name() const;
    long get_id() const;
    int get_num_limbs() const;

    bool set_name(string name);
    bool set_id(long id);
    bool set_num_limbs(int num_limbs);

    string to_string() const;

    static void get_n_pets(size_t n, std::vector<Pet>& pets, int name_length);
    static size_t get_population() { return _population; }

    static string make_a_name(int len);

    // Don't remove this line
    friend class Tests;
};

std::ostream& operator<<(std::ostream& os, const Pet& pet);
bool operator==(const Pet& pet1, const Pet& pet2);
bool operator!=(const Pet& pet1, const Pet& pet2);

#endif /* Pet_h */
```

And your Pet.cpp file:

```
//  Pet.cpp
//  2a-Lab-06-Pets
//

#include <iostream>
#include <sstream>
#include <vector>

#include "Pet.h"

using namespace std;

// This is a way to properly initialize (out-of-line) a static variable.
size_t Pet::_population = 0;

Pet::Pet(string name, long id, int num_limbs) {
    // TODO - Your code here
}

Pet::~Pet() {
    // TODO - Your code here
}

string Pet::get_name() const { return _name; }
long Pet::get_id() const {
    // TODO - Your code here
}

int Pet::get_num_limbs() const {
    // TODO - Your code here
}

bool Pet::set_name(string name) {
    // TODO - Your code here
}

bool Pet::set_id(long id) {
    // TODO - Your code here
}

bool Pet::set_num_limbs(int num_limbs) {
    // TODO - Your code here
}

string Pet::to_string() const {
    // TODO - Your code here
}

// Fill in the supplied pets vector with n pets whose
// properties are chosen randomly.
// Don't mess with this method more than necessary.
void Pet::get_n_pets(size_t n, std::vector<Pet>& pets, int name_len) {
    // TODO - Resize pets as necessary
    long prev_id = 0;
    for (size_t i = 0; i < n; i++) {
        long id = prev_id + 1 + rand() % 10;
        pets[i].set_id(id);
        pets[i].set_num_limbs(rand() % 9); // up to arachnids

        // TODO - make and set a name of the requested length
        // TODO - adjust prev_id as necessary
```

```
    }
}

// ------------------------------------------------------------------------

string Pet::make_a_name(int len) {
    // TODO - Your code here
}

// Optional EC points
// Global helpers
bool operator==(const Pet& pet1, const Pet& pet2) {
    // TODO - Your code here
}

bool operator!=(const Pet& pet1, const Pet& pet2) {
    // TODO - Your code here
}

ostream& operator<<(ostream& os, const Pet& pet) {
    // TODO - Your code here
}
```

Testing your own code

You should test your functions using your own `main()` function in which you try and call your methods in many different ways and cross-check their return values against your expected results. But when you submit you must NOT submit your `main()` function. I will use my own and invoke your functions in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
1. Enter the secret password for this quest in the box
2. Drag and drop your `Pet.*` files into the button and press it. (Make sure you're not submitting a `main()` function)
3. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

I monitor the discussion forums closely and award extra credit points for well-thought out and helpful discussions.

May the best coders win. That may just be all of you.

Happy Hacking,

&