C++ Final Assignment (v1.3.2)

The assignment is broken into two tasks:

Task1 - Quadratic Equation - will test your knowledge in operator-overloading, const-correctness, rule-of-three & rule-of-five. **[50 points]**

Task2 - Vet Program - will test your knowledge in inheritance, abstract classes, RTTI, and the singleton design pattern **[50 points]**

The project will be composed of the following source files:

- Task 1:
  - `Equation.hpp`
  - `Equation.cpp`
- Task 2:
  - Animal.cpp, Animal.hpp
  - Vet.cpp, Vet.hpp
  - Dog.cpp, Dog.hpp
  - Cat.cpp, Cat.hpp
  - Cow.cpp, Cow.hpp

When sending files online for checking, send only the specified source files zipped as .zip or .7z. Do not include any executables or binaries. you may include the files in project VSCode project folders with its .vscode folder (each task in its own project folder). You may include makefile files.

The files for each task are assumed to be compilable using the the following command:
`g++ *.cpp -Wall -Wextra -Weffc++ -std=c++14 -pedantic -o program`

- Assuming all the files are located directly in the current folder

- Assume the tester includes their own main.cpp files with a main() function for testing purposes. You can make your own main.cpp files for testing.

Important Notes:

- No compilation errors allowed.
- Avoid any compilation warnings with the compilation command explained above as much as possible. If a warning exists you will need to justify its existence.
- The code should compile under both Windows and Linux. Use standard C++14 libraries and code only. Do not utilize compiler-specific libraries, functionality or features.
- The code should be properly separated into files.
- Header files should contain declarations only, global/friend/member function definitions, static member variables initializations should all be done in the source file. This includes functions such as getters and setters.
- Where it is correct to use them, defaulted & deleted constructors and operators are allowed to be declared in the header.
- The code should be well commented while keeping a good style: Enough comments to explain to the reader what's going on, but not too many that it makes the code harder to read or explains the obvious, etc.
- Maintain a good consistent style, proper variable and function names. You are allowed to make helper member functions not specified in the project, but keep them private.
- as per the project specification, the following style is restrained:
    - function & variables including members will follow the lower_case style
    - Types will follow the UpperCamelCase style.
    - You are welcome to prefix/suffix private members. But the Task 2's Vet class' member "animals" will retain its name without prefixes or suffixes.
    - You are allowed to choose your own style for constants but it should be identifiable, clear and consistent.
- Avoid memory problems (leaks, aliasing, double delete)
- The code should be const correct, use const where needed and not where it shouldn't be, same for sending receiving values by value, reference, const reference & rvalue reference.

You may be required to present your code to an instructor and/or tester, in which case you will present your code as a project in your development environment, and you are encouraged to create additional files/code to show your code is working correctly or to test its functionality.

Task 1 - Quadratic Equation

(Note: this task is intentionally non-realistic to test resource handling. a realistic implementation of Equation would probably not use dynamic allocation for solutions array and would have less limitations)

Create a class/struct called Equation. It will represent a Quadratic Equation of the form:

$$A * X^2 + B * X + C = 0$$

It will have the following private fields:

a - double representing the A part.

> **A should never be zero!** If there's an attempt to construct an Equation with a=0 or set its value later to zero, the code should throw std::invalid_argument (from the <stdexcept > library). (be careful of constructor memory leaking if the exception is caught)

b - double representing the B part
c - double representing the C part

solutions - double pointer - points to an array of two doubles (dynamically allocated)
solutions_size - std::size_t - represents the size of solutions (0,1,2)

It will have the following public methods:

accessors **`get_*`** and **`set_*`** for a,b & c (i.e. **`get_a()`** which returns the value of a, **`set_a(double)`** which sets the value of a etc..)

when setting a,b or c, the values of **solutions** and **soultions_size** needs to be updated accordingly (more details below. hint: you can make a private update_solutions method and call it when required)

`std::size_t get_solutions_size();` - returns **solutions_size**.
`double const * get_solutions();` - returns **solutions**.

It will have the following constructor:
Equation(double a, double b, double c);

It will follow the rule of Three & the rule of Five.
(You may employ the idioms teached in class to simplify the process)

It will support the following operations (need to be const-correct!):

Allow "pushing" an equation object to an output stream with the << operator. the format will be the same **as though** you called the following printf format:
`"%.1f X^2 + %.1f X + %.1f = 0"`

the first **`"%.1f"`** specifier will show the value of a
the second **`"%.1f"`** specifier will show the value of b
the third **`"%.1f"`** specifier will show the value of c

Do not actually use printf functions to write to a stream. You can utilize the **`std::fixed`** and **`std::setprecision(1)`** manipulators instead. Restoring the old stream state post-printing is not required

(Note: no need to worry about changing addition-sign with subtraction-sign for negative B and C)
(Note: no need to remove " + BX" if B is zero nor will you need to remove "+ C" if C is zero)

Usage example:

```
std::cout << Equation{1,2,3};
```

Output:

```
1.0 X^2 + 2.0 X + 3.0 = 0
```

addition of two equations:

```
Equation(1,2,3) + Equation(40,50,60)
//result: temporary Equation object with a=41, b=52,c=63
```

addition of an Equation and a double. the operation will modify the c part of the result:

```
Equation(1,2,3) + 40.0
//result: temporary Equation object with a=1,b=2,c=43
```

Addition of a double and an Equation (double is the left operand):

```
40.0 + Equation(1,2,3)
//result: temporary Equation object with a=1,b=2,c=43
```

(**NOTE:** if an addition will cause an equation to have a with a value of zero it should throw std::invalid_argument as was explained above, it will be up to the user-code to avoid adding two equations that will cause this or to catch the exception)

How to calculate solutions:

You can do so by checking the result of the **discriminant**:

$$B^2 -4 * A * C$$

If greater than zero: there are two solutions (solution_size = 2;). solutions array will contain the following solutions ("√(...)" is the square root function):

index 0 will contain the result of:

$$(-B + \sqrt{(B^2 - 4AC)})/ 2A$$

index 1 will contain the result of:

$$(-B - \sqrt{(B^2 - 4AC)})/ 2A$$

If equals zero: there is a single solution. (solution_size = 1;). Index 0 of solutions will contain the result of:

$$-B / 2A$$

If smaller than zero: there are no solutions (solution_size = 0;)

Below is an example of a program to test your answers for Task1:

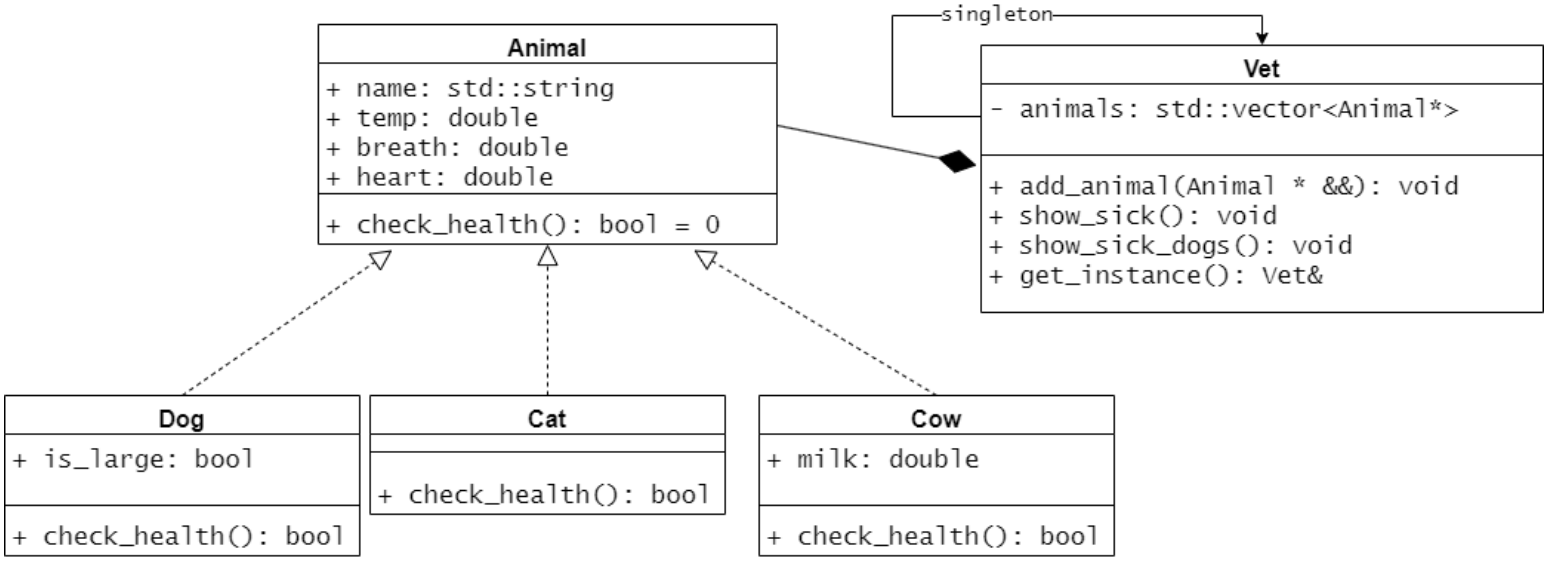
```
#include <iostream>

//...

int main() {
    using namespace std;

    Equation eq(10,20,30);

    cout << eq << endl;
    //should print: 10X^2 + 20X + 30 = 0

    cout << eq.get_solutions_size() << endl;
    //should print 0

    eq = -20 + eq;
```

```
    cout << eq << endl;
    //should print: 10X^2 + 20X + 10 = 0

    cout << eq.get_solutions_size() << endl;
    //should print 1

    cout << eq.get_solutions[0] << endl;
    //should print -1


    cout << Equation(1,5,3) + Equation(2,6,4) << endl;
    //should print: 3X^2 + 11X + 7 = 0

    eq = Equation(1,3,-4);

    cout << eq << endl;
    //should print 1X^2 + 3X + -4 = 0

    cout << eq.get_solutions_size() << endl;
    //should print 2

    cout << "X1 = " << eq.get_solutions()[0] << endl;
    //should print: X1 = 1

    cout << "X2 = " << eq.get_solutions()[1] << endl;
    //should print: X2 = -4

}
```

(End of task 1 description)

Task 2 - Vet program

Build a program to manage the health of animals based on the following UML diagram:



the Animal class

constructor: Animal(string name, double temp, double breath, double heart)

The animal class is an abstract class.
its **check_health** method returns true if the animal is healthy and returns false if it is not.

Animal has four fields:
name (string) - the name of the animal
temp (double) - the temperature of the animal in celsius
breath (double) - respiration rate - the amount of breaths per minute
heart (double) - heart rate - the amount of beats per minute

The Dog class

constructor: Dog(string name, double temp, double breath, double heart, bool is_large)

inherits from Animal, adds an **is_large** field (bool). if true the dog is of a large breed, if false the dog is of a small breed. implements **check_health** (details below)

The Cat class

constructor: Cat(string name, double temp, double breath, double heart)

inherits from Animal. implements **check_health** (details below)

The Cow class

constructor: Cow(string name, double temp, double breath, double heart, double liters)

inherits from Animal, adds **milk** field (double). represents the amount of liters per day the cow produces. implements **check_health** (details below)

check_health()
The check_health override for each class will be based on the following chart. If an animal has any field outside the mentioned ranges, it is considered unhealthy (check health will return false) the ranges are inclusive (i.e 10-35 means from 10 to 35 including 10 and 35):

Type	Body Temperature	Respiratory Rate (breaths per minute)	Heart Rate (beats per minute)	Milk Production
Dog	38-39.2	10-35	60-100 if large 100-140 if small	---
Cat	38-39.2	16-40	120-140	---
Cow	38.5-39.5	26-50	48-84	30 or more liters per day

For example: to check if a Cat is healthy we can use the following condition:

```
(38 <= temp && temp <= 39.2)
&& (16 <= breath && breath <= 40)
&& (120 <= heart && heart <= 140)
```

The Vet class

Will represent a collection of animal pointers inside a vector. The lifetime of the animal objects will depend on the lifetime of the vet class (when the Vet class is destroyed, its destructor will first delete all the animal objects)

You may choose to use std::unique_ptr to handle the animal pointers for this task in which case the animals vector will be of type:

```
std::vector< std::unique_ptr<Animal> > animals;
```

However: the signature for add_animal will remain the same:

```
void add_animal(Animal * &&);
```

The Vet class will be a singleton based on a static-local instance created inside the **get_instance** method. Make sure to prevent-access for the default constructor and destructor. Additionally make sure to prevent access or preferably delete, the rule of 5 constructors and operators.

Vet Methods:

get_instance - a static method with a static local instance of Vet. will return a reference to that instance.

add_animal - the method add_animal will receive an rvalue of a pointer of a dynamically allocated animal (dog/cat/cow) and will add it to the vector.

show_sick - will print to cout the name of each animal that is sick (meaning its check_health returned false)

show_sick_dogs - will print to cout the name of each Dog instance that is sick (ignoring other animal instances)

Below is an example of a program to test your Vet class:

```
#include <iostream>

int main() {
    using namespace std;
    cout << "program start" << endl;
    Vet& vet = Vet::get_instance();
```

```
    //healthy dog
    vet.add_animal(new Dog("dog1", 38.5, 20, 80, true));

    //not healthy dog (temp too high)
    vet.add_animal(new Dog("dog2", 40.0, 20, 80, true));

    //not healthy dog (heart rate too slow for a small breed)
    vet.add_animal(new Dog("dog3", 38.5, 20, 80, false));

    //healthy cat
    vet.add_animal(new Cat("cat1", 38.7, 30, 130));

    //not healthy cat (breath is too high)
    vet.add_animal(new Cat("cat2", 38.7, 42, 130));

    //healthy cow
    vet.add_animal(new Cow("cow1", 39.0, 37, 70, 32);

    //not healthy cow (milk production too low)
    vet.add_animal(new Cow("cow2", 39.0, 37, 70, 20);

    vet.show_sick(); //should print: dog2 dog3 cat2 cow2
    vet.show_sick_dogs(); //should print: dog2 dog3
}
```

(End of task 2 description)