**CS 121 Week 10 Worksheet - Friend Keyword and Operator Overloading**

**Notes and Syntax Examples:**

- The **friend** keyword is used to allow external functions/classes access to all of another class's members.

* + - Typically, the friend keyword is used when overloading the insertion/extraction operators, and when having classes that should share details but could not be made into a singular class. It doesn't matter if the friend is placed under the private or public section of the class.

- **Operator overloading** is used to define operators used in C++ to work within your class, which gives you freedom of either really helping users of your class out (e.g. making an assignment operator), or messing with them (e.g. making addition do subtraction instead).

* + - The main reason to overload operators is to make your class easier to use. For instance, imagine a vector class or string class without the assignment operator. You'd be stuck copying each position using a for loop (like a c-string)!
    - The following are some of the more commonly overloaded operators in C++:
      * `=`: Assignment operator (especially if you have pointers in your class)
      * The comparison operators ( `>`, `<`, `==`, etc.)
      * Insertion/Extraction operators (`>>` and `<<`)
      * Unary arithmetic operators (`++`, `--`); (In this case unary means one operand)
      * Arithmetic operators (`+`, `-`, `\*`, etc.)
    - Although there are other operators we can overload, chances are we won't be overloading them in this class (such as the subscript, `[]` and the pointer-to operator `->`).
    - Here are some examples of operator overloading. Note how the function headers look; <Credit of most to http://stackoverflow.com/questions/4421706/operator-overloading>:

*/\*\*\*\* EXAMPLE CLASS \*\*\*\*/*

*class SomeClass {*

*int var; //example member*

*public:*

*//...other methods...*

*SomeClass& operator=(SomeClass& other); //assignment*

*SomeClass& operator++(); //pre-increment*

*SomeClass operator++(int); //post-increment*

*SomeClass& operator+=(const SomeClass& rhs); //plus-equals*

*SomeClass& operator+(SomeClass lhs, const SomeClass& rhs); //plus*

*friend std::ostream& operator<<(std::ostream& os, const SomeClass& obj); //insertion*

*};*

*/\*\*\*\* ASSIGNMENT OPERATOR \*\*\*\*/*

*SomeClass& SomeClass::operator=(SomeClass& other)*

*{*

*//do all assignment stuff you would in a copy constructor to the this pointer*

*this->var = other.var; //example of the above comment*

*return \*this; //return itself*

*}*

*/\*\*\*\* INSERTION OPERATOR \*\*\*\*/*

*//You do not need the scope operator in front of 'operator<<' (as it's not a part of SomeClass)*

*std::ostream& operator<<(std::ostream& os, const SomeClass& obj)*

*{*

*os << obj.var; //example of writing obj to stream, just like using cout*

*return os; //required for this function overload*

*}*

*/\*\*\*\* PRE-INCREMENT OPERATOR \*\*\*\*/*

*SomeClass& SomeClass::operator++()*

*{*

*//do increment on member variable(s), for instance with this class:*

*var++;*

*return \*this;*

*}*

*/\*\*\*\* POST-INCREMENT OPERATOR \*\*\*\*/*

*SomeClass SomeClass::operator++(int)*

*{*

*SomeClass temp(\*this); //assuming you have a copy constructor made*

*operator++(); //reusing the previous operator (pre-increment) made*

*return temp; //note the order of operations here versus the other increment*

*}*

*/\*\*\*\* PLUS-EQUALS OPERATOR \*\*\*\*/*

*SomeClass& SomeClass::operator+=(const SomeClass& rhs)*

*{*

*//we're just adding `rhs` to `\*this`*

*`this->var = this->var + rhs.var`; // Alternative way is to do `var = var + rhs.var`*

*return \*this;*

*}*

*/\*\*\*\* ADDITION OPERATOR \*\*\*\*/*

*SomeClass& SomeClass::operator+(SomeClass lhs, const SomeClass& rhs)*

*{*

*//Keep in mind that the first parameter is NOT pass by reference*

*lhs.var = lhs.var + rhs.var; //Alternatively: `lhs+=rhs` (reuse of `+=` from before)*

*return lhs; //We could have also made a temp variable, like post-increment*

*}*

------------------------------------------------------------------------------------------------------------------------------------------

**Problems:**

1. Given the following overloaded operators (that are defined outside of their class), write the implementations for the `>`, `<=`, `>=`, and `!=` operators. Make sure to include the scope for each implementation. **HINT:** Reuse the given operators (e.g. `greater than` is the same as `not less than or equal to`). Your implementations should be very short for each newly-introduced operator and **DO NOT** require the dot operator.

*bool Dog::operator== (const Dog& lhs, const Dog& rhs)*

*{*

*return ((lhs.age == rhs.age) && (lhs.name == rhs.name));*

*}*

*bool Dog::operator<(const Dog& lhs, const Dog& rhs)*

*{*

*return (lhs.age < rhs.age);*

*}*

2. Using the given implementation for the insertion operator, write the implementation for the extraction operator on a class called Pokemon. It should read in integers for health, attack, defense, and speed. Make sure to include the implementation of the class, Pokemon, as well as your implementation below it (making sure it's outside of the class, as it's a `friend` and not an actual member).