Brett Nicholas

ENGS 65 Homework 1: ADTs

In designing a 3-D Vector data type, I made the following overarching design choices:

1) Data Structure
   - Private instance variables for x,y,z components
     i. Alternative: public instance variables
     ii. This design choice was made in order to uphold the principles of information hiding. The user should be able to see vector dimensions, but not able to manipulate them once the vector is created. This allows any further programs that might depend on a vector to always be the same vector to remain stable.
   - All objects returned should never be references to the object itself, but should be dummy copies
     i. Alternative: passing a pointer to the object itself
     ii. This is for a similar reason as above. Information hiding and employing private types ensures that the user can never manipulate the actual data that represents the vector itself, once created.
2) Object Operations
   - Create member functions for vector operations
     i. Alternative: let the user create their own
     ii. I chose to implement a series of member functions in order to reinforce the encapsulation of the 3D vector class. The alternative would require compromising on the principles of encapsulation and information hiding, as it would require user access to private data, and would limit the reusability of the code for complex vector operations. Instead, I included a number of common vector operations such as the dot product, cross product, magnitude calculation, and the other operations included in the code, so that the class would ensure code portability and stability.
   - Override +/- operators
     i. Alternative: let user do it manually
     ii. This design choice also serves to hide information, and encapsulate all relevant manipulations for the data.
   - Return vector dimensions in array form
     i. Alternative: Have user access each dimension individually
     ii. This was chosen because returning the vector components in an array facilitates access to every element at once, and more importantly, allows for the user to represent a vector object as a column vector in a matrix. This is a common convention in linear algebra, and allows for easy manipulation of data, as

well as integration with MATLAB and other matrix-oriented programs.

3) Implementation in C++
    - See attached C++ pseudocode for a rough conceptual implementation of the above ideas. **NOTE: Assume all arguments to functions are passed by reference, and that all function and get/set return values point to copies, and not actual data.**

SEE ATTACHED CODE

```
/*
 * 3D Vector Class in PseudoCode C++
 * ENGS 65 Homework1
 * Author: Brett Nicholas
 * 1/12/2015
 */
class Vector {
    private:
        double x, y, z;
    public:
        /* constructors */
        Vector(); /*init to 0*/
        Vector(double x, double y, double z);
        ~Vector();

        /* Get/setters  */
        double get_components() {return new double[3] = {this.x, this.y, this.z}; //
 returns array holding components

        /* class methods */
        float magnitude() { return sqrt(this.x^2 + this.y^2 + this.z^2) }; // return
s magnitude of vector
        double dot(Vector v); // returns dot product scalar
        Vector cross(Vector v); // returns new cross product vector
        Vector proj(Vector v); // returns vector projection onto Vector v

        /* overloaded operators */
        Vector& operator + (Vector LHS, Vector RHS); // redefine addition for vector
s
        Vector& operator - (Vector LHS, Vector RHS); // redefine subtraction for vec
tors
};

/* default constructor */
Vector::Vector() { /* default constructor*/
    this->components = {0,0,0};
};

/* Constructor to initialize components */
Vector::Vector(double x, double y, double z) {
    this.x = x;
    this.y = y;
    this.z = z;
};

/* calculates and returns dot product scalar */
float Vector::dot(Vector v) {
    return ( (this.x * v.x) + (this.y * v.y) + (this.z * v.z) );
};

/* calculates and returns cross product vector */
Vector Vector::cross(Vector v) {
    /*
     * compute cross product here
     */
    return new_vector;
};

Vector operator + (vector LHS, vector RHS) {
    /*
     * redefine addition to define vector addition
     */
}

Vector operator - (vector LHS, vector RHS) {
    /*
     * overload subtraction to define vector subtraction
     */
};

}; // end class Vector
```