**Question 1:**
Below are two different accessor functions implemented for a four-dimensional vector class. They are used to get the floating point number stored at a particular index in the vector. Note the different return types of these functions as well as their const-ness.

In **two sentences**, give an example of one situation in which the first operator[] *must* be used rather than the second, and one situation in which the second operator[] *must* be used rather than the first.

```
float vec4::operator[](unsigned int index) const
{
  assert(index <  4);
  return data[index];
}

float& vec4::operator[](unsigned int index)
{
  assert(index <  4);
  return data[index];
}
```

1. The first operator must be used when using [] on a const variable since the function is const
2. The second operator must be used when modifying an element of a variable, since it returns a reference to the element

**Question 2:**
Which of these function headers is the best way to pass two three-dimensional vectors into a function that computes their cross product without directly modifying them?

```
A. vec3 Cross ( vec3 v1, vec3 v2 )
B. vec3 Cross ( vec3* v1, vec3* v2 )
C. vec3 Cross ( vec3& v1, vec3& v2 )
D. vec3 Cross ( const vec3 v1, const vec3 v2 )
E. vec3 Cross ( const vec3* v1, const vec3* v2 )
F. vec3 Cross ( const vec3& v1, const vec3& v2 )
```

In **one sentence**, explain why the implementation you chose is the best one.

Answer: F
Const ensures we won’t modify the input vec3s, and making them references saves stack space.

**Question 3:**

```
class Rational {..};

      Rational operator*(const Rational& lhs, const Rational& rhs);
      const Rational operator*(const Rational& lhs, const Rational& rhs);
```

Consider the above two implementations of the operator* function. Generally speaking, is the **non-const** or **const** version more appropriate for common use? Explain with **one example**.

The const version is more appropriate. Non-const allows the statement “a*b = c” to modify the return value of a*b, which is undesirable.

**Question 4:**

```
class Node
{
private:
      std::string name;
public:
      std::string& getName() const;
      void setName(const std::string& newName);
};
```

The code above has a subtle mistake in it that allows the user to *unintentionally* change the name of a Node. In **one sentence**, explain how to change the code in order to prevent this.

Change getName so it returns a const std::string&:

```
      const std::string& getName() const;
```