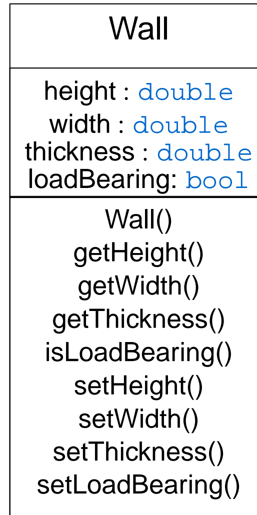
Below is a partial definition of the Rational class that we discussed in lecture this week.

```
class Rational
{
public:
  Rational(int n=0, int d=1);
  Rational add(const Rational& rhs) const;
  Rational subtract(const Rational& rhs) const;
  void setValue(int n, int d=1);
  ...
private:
  int num;
  int den;
};
```

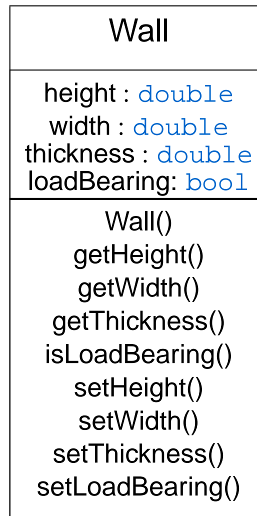
Implement the setValue() member function for the Rational class.

The function should handle data members in a way that is consistent with the member functions developed in lecture.

Write the class definition (but not implementation) for the class described below in UML.



Write the copy constructor for the class described below in UML.



Recall from lecture that the function,

```
ostream& operator<<(ostream& lhs, const Rational& rhs);
```

was not a member of the Rational class. Explain why we defined it outside of the Rational class.

Write a function called percentOdd that will accept a list of `int`s and return the percentage of odd numbers in the list.

Write a function, called removeElem, that accepts a list of doubles, lst, and an unsigned int, n. If there are at least n elements in lst, the function should remove the nth element in lst and return true. Otherwise, the function should return false.

Recall the following VecInt class discussed in lecture this week:

```
class VecInt
{
public:
  VecInt();
  VecInt(unsigned int sz, int val=0);
  VecInt(const VecInt& rhs);
  ~VecInt();
  VecInt& operator=(const VecInt& rhs);
  void push_back(int val);
  unsigned int size() const;
  bool empty() const;
  void clear();
...
private:
  unsigned int size;
  unsigned int capacity;
  int* array;
};
```

Assume, as we did in lecture, that the default constructor allocates space for an array of 25 integers. Write the clear member function.

Define the term **memory leak**, and write a short code segment which creates a memory leak. Be sure to explain your code. You may find it useful to draw a picture.

Suppose that a Point class similar to the one discussed in lecture has already been defined. Write the class definintion for a ColorPoint class which has all of the same functionality as the Point class with the following exceptions:

- It should keep track of the color of the point. You may use a Pixel object to represent its color.
- The draw() member function from the ColorPoint should draw using the appropriate color.
- There should be a setColor() member function that allows the color of the point to change.

Note: You are not required to implement any member functions.

Explain the role of **composition** in object oriented design. How does it differ from **inheritance**?

Recall the List<T> and Link<T> classes developed in lecture this week.

```
template <class T>
class Link {
public:
  friend class List<T>;
  friend class LstItr<T>;
  Link();
  ...
private:
  T value;
  Link<T>* next;
  Link<T>* prev;
};

template <class T>
class List {
public:
  List();
  List(const List<T>& rhs);
  ~List();
  List<T>& operator=(const List<T>& rhs);
  void push_back(const T& val);
  ...
private:
  unsigned int numEl;
  Link<T>* start;
  Link<T>* end;
};
```

Write the push_back member function for the List<T> class. You may only make use of List and Link member functions that are explicitly listed above.

Suppose that a Point class similar to the one discussed in lecture has already been defined. Write the class definintion for a ColorPoint class which has all of the same functionality as the Point class with the following exceptions:

- It should keep track of the color of the point. You may use a Pixel object to represent its color.
- The draw() member function from the ColorPoint should draw using the appropriate color.
- There should be a setColor() member function that allows the color of the point to change.

Note: You are not required to implement any member functions.

Explain what keyword `virtual` is used for. Give a simple example of when it is needed.

Recall the List<T> and Link<T> classes developed in lecture.

```
template <class T>
class Link {
public:
  friend class List<T>;
  friend class LstItr<T>;
  Link();
  ...
private:
  T data;
  Link<T>* next;
  Link<T>* prev;
};

template <class T>
class List {
public:
  List();
  List(const List<T>& rhs);
  ~List();
  List<T>& operator=(const List<T>& rhs);
  void push_back(const T& val);
  ...
private:
  unsigned int numEl;
  Link<T>* start;
  Link<T>* end;
};
```

Write the push_back member function for the List<T> class. You may only make use of List and Link member functions that are explicitly listed above.