In this problem you will complete the implementation of two new classes named Bulb and Lamp. We begin with an example of how these classes are used.

First, we create a new lamp that will hold 3 bulbs and make a note of the manufacturer’s recommended bulb: a 60 watt bulb with an estimated lifetime of 300 hours from Phillips. Note that initially this lamp has no bulbs installed.

We install one of the manufacturer’s recommended bulbs and use the lamp (turn it “on”) for a total of 50 hours.

```cpp
defloorlamp(Bulb(60,300,"Phillips"),3);
bool success;
success = floorlamp.install(); assert(success);
floorlamp.on(50);
assert(floorlamp.getTotalWattage() == 60);
```

Next, we attempt to install 3 bulbs: another of the manufacturer’s recommended bulbs, and then two other brands of bulbs. The installation of the 3rd bulb made by Sylvania fails because there are no available sockets in the lamp and no bulbs are burnt out and need replacement.

```cpp
success = floorlamp.install(); assert(success);
success = floorlamp.install(Bulb(40,120,"GE")); assert(success);
success = floorlamp.install(Bulb(120,500,"Sylvania")); assert(!success);
```

We then use the lamp for another 100 hours. Once the wattage drops (due to a burnt out bulb), we again try to install the Sylvania bulb and it is successful.

```cpp
floorlamp.on(100);
assert(floorlamp.getTotalWattage() == 160);
floorlamp.on(50);
assert(floorlamp.getTotalWattage() == 120);
success = floorlamp.install(Bulb(120,500,"Sylvania")); assert(success);
assert(floorlamp.getTotalWattage() == 240);
```

Finally, we create a duplicate lamp. Note that when we do this, we match the bulbs currently installed in the original lamp, but the bulbs installed in the new lamp are brand new (and unused).

```cpp
Lamp another(floorlamp);
assert(floorlamp.getTotalWattage() == another.getTotalWattage());
for (int i = 0; i < 10; i++) {
    floorlamp.on(50);
    another.on(50);
    std::cout << "compare " << floorlamp.getTotalWattage() " "
    << another.getTotalWattage() " <" std::endl;
}
```

Which results in this output:

```
compare 240 240
compare 240 240
compare 180 240
compare 120 240
compare 120 240
compare 120 120
compare 120 120
compare 120 120
```
1.1 Bulb Class Declaration [ / 4 ]

The Bulb class is missing only one function. *You will need to read the rest of the problem to determine what’s missing.* Fill in the missing function – implement the function right here, within the class declaration.

```cpp
class Bulb {
    public:
    // constructors
    Bulb(int w, int l, const std::string &b) :
        wattage(w), lifetime(l), hours_used(0), brand(b) {}
    Bulb(const Bulb& b) :
        wattage(b.wattage), lifetime(b.lifetime), hours_used(0), brand(b.brand) {} 

    // accessors
    int getWattage() const { return wattage; } 
    bool burntOut() const { return hours_used > lifetime; }
    const std::string& getBrand() const { return brand; }
    // modifier
    void on(int h) { hours_used += h; }

    private:
    // representation
    int wattage;
    int lifetime;
    int hours_used;
    std::string brand;
};
```

1.2 Lamp Class Declaration [ / 12 ]

The Lamp class has a few more missing pieces. *Read through the rest of the problem before attempting to fill this in.* Write the prototypes (not the implementation!) for the four missing functions. You will implement some of these missing functions later. Also, fill in the member variables for the Lamp representation. Important: You may not use STL vector on this problem.
class Lamp {
public:
    // constructors, assignment operator, destructor
    Lamp(const Bulb& b, int num);
    Lamp(const Lamp &l);
    const Lamp& operator=(const Lamp &l);
    ~Lamp();

    // accessor
    int getTotalWattage() const;

    // modifiers
    bool install(const Bulb &b = Bulb(0,0,""));
    void on(int h);

private:
    // representation
    Bulb recommended;
    Bulb** installed;
    int max_bulbs;
};

Lamp Class Implementation

Here's the implementation of one of the key member functions of the Lamp class.

bool Lamp::install(const Bulb &b) {
    // first, let's figure out where to install the bulb
    int which = -1;
    for (int i = 0; i < max_bulbs; i++) {
        // check for an empty socket
        if (installed[i] == NULL) {
            which = i;
            break;
        }
        // or a socket that contains a burnt out bulb
        if (installed[i]->burntOut()) {
            which = i;
            delete installed[i];
            break;
        }
    }
    // return false if we cannot install this bulb
    if (which == -1) return false;
    if (b.getWattage() == 0) {
        // install the manufacturer's recommended bulb type
        installed[which] = new Bulb(recommended);
    } else {
        // install the specified bulb
        installed[which] = new Bulb(b);
    }
    return true;
}

On the last two pages of this problem you will implement three important functions for the Lamp class, as they would appear outside of the class declaration (in the lamp.cpp file) because their implementations are > 1 line of code.
1.3 Lamp Constructor [ / 8 ]

Solution:
Lamp::Lamp(const Bulb& b, int num) : recommended(b) {
    max_bulbs = num;
    for (int i = 0; i < num; i++) {
        installed[i] = NULL;
    }
}

1.4 Lamp Destructor [ / 6 ]

Solution:
Lamp::~Lamp() {
    delete [] installed;
}

1.5 Lamp Copy Constructor [ / 12 ]

Hint: This function should use the Bulb class copy constructor.

Solution:
Lamp::Lamp(const Lamp &l) : recommended(l.recommended) {
    max_bulbs = l.max_bulbs;
    installed = new Bulb*[max_bulbs];
    for (int i = 0; i < max_bulbs; i++) {
        if (l.installed[i] != NULL) {
            installed[i] = new Bulb(*l.installed[i]);
        }
    }
}

2 Singly Linked List Subsequence Sum [ / 18 ]

Write a recursive function named FindSumStart that takes the head Node of a singly-linked list storing positive numbers. The function should return a pointer to the Node that begins a subsequence of numbers that ends in the sum of that subsequence. For example, given this sequence: 5 1 4 2 3 9 6 7 the function should return a pointer to the Node storing 4, because 4 + 2 + 3 = 9.

template <class T>
class Node {
public:
    Node(const T& v) : value(v), next(NULL) {}
    T value;
    Node* next;
};
Solution:
```
template <class T> Node<T>* FindSumStart(Node<T>* n) {
    if (n == NULL) {
        return NULL;
    }
    int total = 0;
    Node<T>* tmp = n;
    while (tmp != NULL) {
        if (total == tmp->value) {
            return n;
        }
        total += tmp->value;
        tmp = tmp->next;
    }
    return FindSumStart(n->next);
}
```

Assuming the sequence has \( n \) numbers, what is the order notation for the running time of your function?

Solution: \( O( n^2) \)

3 Reverse Splice [ / 20 ]

Write a function named `reverse_splice` that takes in 3 arguments: an STL list named `data` and two iterators `i` and `j`. The function should reverse the order of the data between those iterators. For example, if `data` initially stores this sequence: 1 2 3 4 5 6 7 8 9 and `i` refers to 3 and `j` refers to 7, then after the call `reverse_splice(data,i,j)`, `data` will contain: 1 2 7 6 5 4 3 8 9, `i` will refer to element 7, and `j` will refer to element 3. Your function should return `true` if the operation was successful, and `false` if the request is invalid. Note: Your function may only use a constant amount of additional memory.

Solution:
```
template <class T>
bool reverse_splice(std::list<T> &data, 
    typename std::list<T>::iterator &i, 
    typename std::list<T>::iterator &j) {
    // checking that i comes before j within data
    if (j == data.end()) {
        return false;
    }
    // slide the splice end iterator forward (off of the last element)
    j++;
    typename std::list<T>::iterator k;
    for (k = i; k != j; k++) {
        if (k == data.end()) return false;
    }
    // use a helper iterator to keep track as we walk between the
    // endpoints of the splice
    k = j;
    while (k != i) {
        // move one element
        data.insert(k,*i);
        k--;
        i = data.erase(i);
    }
    // back the splice end iterator back onto the last element
    j--;
    return true;
}
```
4    Doubly Linked Factorization [    / 17 ]

class Node {
public:
    Node(int v) :
        value(v),
        next(NULL),
        prev(NULL) {}
    int value;
    Node* next;
    Node* prev;
};

Write a recursive function named Factor that takes in two arguments, pointers to the head and tail Nodes of a doubly linked list. This function should look for a non-prime number in the linked list structure, break the Node into two Nodes storing two of its factors, and then return true. If all elements are prime the function returns false. For example, if we start with a 3 element list containing 35 30 28 and repeatedly call Factor:

PrintNodes(head);
while (Factor(head,tail)) { PrintNodes(head); }

This is the output:
35 30 28
5 7 30 28
5 7 2 15 28
5 7 2 3 5 28
5 7 2 3 5 2 14
5 7 2 3 5 2 7
5 7 2 3 5 2 7

You may write a helper function. You do not need to write the PrintNodes function.

Solution:
bool Factor(Node* &head, Node* &tail, Node* n) {
    // base case
    if (n == NULL) return false;
    // see if this element has any factors
    for (int i = 2; i < n->value; i++) {
        if (n->value % i == 0) {
            // create a new node in front of this one
            Node* tmp = new Node(i);
            // change all of the links
            tmp->prev = n->prev;
            if (n->prev != NULL) {
                tmp->prev->next = tmp;
            }
            tmp->next = n;
            n->prev = tmp;
            n->value = n->value / i;
            // handle the special case of the first node
            if (n == head) head = tmp;
            return true;
        }
    }
    // recurse if we couldn't split this element
    return Factor(head,tail,n->next);
}

// driver function
bool Factor(Node* &head, Node* &tail) {
    return Factor(head,tail,head);
}