1. How many DS students to change a lightbulb? [ / 38 ]

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

```
Lamp floorlamp(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 slots in the lamp and no bulbs are burnt out and need replacement.

```
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.

```
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).

```
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 120 240
compare 120 120
compare 120 120
compare 120 120
```
1.1  Bulb Class Declaration [ / 14 ]

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.

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

    // 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 [ / 14 ]

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 Class Implementation

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

```cpp
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 [ / 9 ]

sample solution: 7 line(s) of code

1.4 Lamp Destructor [ / 5 ]

sample solution: 8 line(s) of code
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.

```cpp
template <class T>
class Node {
public:
    Node(const T& v)
        : value(v),
          next(NULL) {}  
    T value;
    Node* next;
};
```

Assuming the sequence has $n$ numbers, what is the order notation for the running time of your function?
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.

sample solution: 21 line(s) of code
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` `Node`s of a doubly linked list. This function should look for a non-prime number in the linked list structure, break the `Node` into two `Node`s 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 2 7
5 7 2 3 5 2 2 7

You may write a helper function. You do not need to write the `PrintNodes` function.
5 Dynamically Allocated & Templated Stairs

In this problem you will write a simple class to build a staircase-shaped storage shelf. Here’s an example usage of the class, which constructs the diagram on the right.

```cpp
Stairs<char> s(4,'_');
s.set(0,0,'A');
s.set(1,1,'B');
s.set(2,2,'C');
s.set(3,3,'D');
s.set(2,1,'U');
s.set(3,1,'S');
```

5.1 Stairs Class Declaration

First, fill in the blanks in the class declaration:

```cpp
class Stairs {
    public:
        // constructor
        sample solution: 1 line(s) of code

        // destructor
        sample solution: 1 line(s) of code

        // prototypes of 2 other important functions related to the constructor & destructor
        sample solution: 2 line(s) of code

        // modifier
        void set(int i, int j, val) { data[i][j] = val; }

        /* NOTE: other Stair functions omitted */
    private:
        // representation
        sample solution: 2 line(s) of code
};
```
5.2  Stairs Constructor [ / 9 ]

Now write the constructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

sample solution: 10 line(s) of code

5.3  Stairs Destructor [ / 5 ]

Now write the destructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

sample solution: 6 line(s) of code
6 Comparing Linked List Pointers w/ Recursion

Ben Bitdiddle is working on a software project for essay writing using a doubly-linked chain of nodes. His initial `Node` class is on the right.

One of the features of his software allows a user to compare the location of two words within the document and say which word appears first. Ben plans to implement this using two helper functions: `search` and `compare`.

### 6.1 Searching for a Word

First, let's write the `search` function, which takes in two arguments: a pointer to the first `Node` in the document (word chain) and the specific word we’re looking to find. The function returns a pointer to the first `Node` containing that word. Use recursion to implement this function.

```cpp
class Node {
    public:
        std::string word;
        Node* next;
        Node* prev;
};
```

**sample solution:** 7 line(s) of code

If the `Node` chain contains $n$ elements, what is the running time of the `search` function?

### 6.2 Comparing Positions within the Node Chain

Next, let’s implement the `compare` function. This function takes in two `Node` pointers and returns true if the first argument appears closer to the front of the list than the second argument. For example, let’s say a chain of word `Nodes` named `sentence` contains:

```
the quick brown fox jumps over the lazy dog
```

Here’s an example using the `search` and `compare` functions:

```cpp
Node* over = search(sentence,"over");
Node* quick = search(sentence,"quick");
Node* lazy = search(sentence,"lazy");

assert (compare(quick,over) == true);
assert (compare(over,quick) == false);
assert (compare(quick,lazy) == true);
assert (compare(lazy,over) == false);
```
Again using recursion, implement the compare function.

*sample solution: 7 line(s) of code*

If the Node chain contains $n$ elements, what is the running time of the compare function?

**Improving Word Position Comparison Performance**

Alyssa P. Hacker stops by to help, and suggests that Ben switch to a different data structure if he is frequently comparing word positions within a long essay.

But Ben’s a stubborn guy. Instead of switching to a different data structure, he has a plan to augment his list structure to improve the running time of compare. Ben explains that the new distance member variable in each node will indicate how far away the node is from the front of the list.

Here’s Ben’s new compare function:

```cpp
bool compare_fast(Node *a, Node *b) {
    return a->distance < b->distance;
}
```

Ben reassures Alyssa that he’ll add some error checking to this code.

*SIDE NOTE: Hopefully your implementation of the original compare function has some error checking!*

But Alyssa is more concerned about how this addition to the data structure will impact performance when the essay or sentence is edited. She says he can’t afford to change the distance in all or many Nodes in the data structure any time a small edit is made to the document.

Ben explains that the push_back function will assign the distance of the new Node to be the distance of the last Node in the chain plus 10.0. And similarly, push_front will assign the new Node to be the distance from the first Node minus 10.0. *BTW, negative distance values are ok.* Finally, Ben says the insert_between function (on the next page) can similarly be implemented without editing the distance value in any existing Node!
6.3 Implementing insert_between and Maintaining Fast Comparisons

Continuing with the previous example, here's a quick demonstration of how this function works:

```c
bool success = insert_between(sentence,"the","lazy","VERY");
assert (success);
Node* VERY = search(sentence,"VERY");
assert (compare(VERY,lazy) == true);
assert (compare(quick,VERY) == true);
assert (compare_fast(VERY,lazy) == true);
assert (compare_fast(quick,VERY) == true);
success = insert_between(sentence,"quick","fox","RED");
assert (!success);
```

And here's the contents of the `sentence` variable after the above fragment of code:

```
the quick brown fox jumps over the VERY lazy dog
```

Implement `insert_between`. And yes, use recursion.

Sample solution: 16 line(s) of code
Write a function named `erase_middles` that takes in 2 arguments: an STL list named `data` and a `value`. The function should remove all instances of `value` from `data`, except the first and the last instances. The function returns the number of removed elements. For example, if `data` initially contains:

```
5 2 5 2 3 4 3 2 5 2 3 2 3 4 2 5
```

A call to `erase_middles(data, 5)` will return 2 and now `data` contains:

```
5 2 2 3 4 3 2 2 3 2 4 2 5
```

And then a call to `erase_middles(data, 2)` will return 4 and `data` contains:

```
5 2 3 4 3 3 3 4 2 5
```

*sample solution: 22 line(s) of code*
For each program bug description below, write the letter of the most appropriate debugging skill to use to solve the problem. Each letter should be used at most once.

- A) get a backtrace
- B) add a breakpoint
- C) use step or next
- D) add a watchpoint
- E) examine different frames of the stack
- F) reboot your computer
- G) use Dr Memory or Valgrind to locate the leak
- H) examine variable values in gdb or lldb

A complex recursive function seems to be entering an infinite loop, despite what I think are perfect base cases.

The program always gets the right answer, but when I test it with a complex input dataset that takes a long time to process, my whole computer slows down.

I’m unsure where the program is crashing.

I’ve got some tricky math formulas and I suspect I’ve got an order-of-operations error or a divide-by-zero error.

I’m implementing software for a bank, and the value of a customer’s bank account is changing in the middle of the month. Interest is only supposed to be added at the end of the month.

Select one of the letters you did not use above, and write a concise and well-written 3-4 sentence description of a specific situation where this debugging skill would be useful.
Let’s build a histogram of students organized by their grade on Homework 1 into buckets of size 5. An example of the data is to the right and the expected output is on the far right.

First, write a fragment of code (as it would appear in `main.cpp`) to open the file `hw1_scores.txt` and read the data into an STL `vector` of STL `vector`s of STL `string`s. The vector should be resized as necessary to adapt to the maximum score present in the file.

Next, write code to output the data stored in the vector to `std::cout` as shown above.
Flipping & Sorting Words

Finish the implementation of the function `FlipWords` that takes in an alphabetically sorted STL list of STL strings named `words` and modifies the list. The function should remove all palindromes (words that are the same forwards & backwards). The function should insert the flipped (reversed) version of all other words into the list, in sorted order. For example this input list:

```
bard civic diva flow pots racecar stop warts
```

Should be changed to contain:

```
avid bard diva drab flow pots stop straw warts wolf
```

You may not use STL sort. You may assume the input list does not contain any duplicates. And after calling the `FlipWords` function the list should not contain any duplicates.

```cpp
std::string reverse(std::string &word) {
    std::string answer(word.size(),' ');
    for (int i = 0; i < word.size(); i++) { answer[i] = word[word.size()-1-i]; }
    return answer;
}
void FlipWords(std::list<std::string> &words) {
    while (current != words.end()) {
        std::string flip = reverse(*current);
        if (flip == *current) {
            sample solution: ≤3 line(s) of code
        } else {
            sample solution: ≤8 line(s) of code
        }
    }
}
```
Ben Bitdiddle thinks he has stumbled on a brilliant idea to make each Node of a doubly linked list “smart” and store global information about the list. Each Node will have a pointer to the head and tail Nodes of the overall list.

Help him by finishing the implementation of `PushFront` to add a new element to the list. *Note: You should not change the value inside of any existing Nodes.*

```cpp
void PushFront(Node* head, Node* tail, int v) {
    Node* tmp = new Node;
    tmp->value = v;
    if (head == NULL) {
    } else {
        ...
    }
}
```

*sample solution: 4 line(s) of code*
Alyssa P. Hacker has joined the Rensselaer Center for Open Source Software and is working on a program to help students manage their schedules over their time at RPI. She will use a two dimensional array to store courses taken each term. The declaration for two key classes is shown on the right:

Alyssa’s program assumes that all undergraduate RPI degree programs require students to take 32 4-credit courses. She also assumes that each specific student takes the same number of courses per term throughout their time at RPI.

Your task is to implement the critical functions for this class with dynamically allocated memory, as they would appear in the Student.cpp file. Make sure to use the private helper functions as appropriate so your code is concise.

A few examples of usage are shown below.

```cpp
class Course {
public:
    Course(const std::string &p="XXXX", int n=1000) : prefix(p), num(n) {} /* member functions omitted */
private:
    std::string prefix;
    int num;
};

class Student {
public:
    Student();
    Student(int courses_per_term_);
    Student(const Student& s);
    const Student& operator=(const Student& s);
    Student();
    int numTerms() const { return num_terms; }
    const Course& getCourse(int t, int c) const { return data[t][c]; }
    /* additional member functions omitted */
private:
    void initialize();
    void copy(const Student& s);
    void destroy();
    int num_terms;
    int courses_per_term;
    Course** data;
};

// a typical student takes 4 courses per term for 8 terms
Student regular; assert (regular.numTerms() == 8);
// if a student takes 5 courses per term, they can finish in 3.5 years
Student overachiever(5); assert (overachiever.numTerms() == 7);
// students who take 3 courses per term will require 5.5 years
Student supersenior(3); assert (supersenior.numTerms() == 11);
/* details of how courses are scheduled omitted */

Student::Student() {
    sample solution: 3 line(s) of code
}

Student::Student(int courses_per_term_) {
    sample solution: 3 line(s) of code
}
Student::Student(const Student& s) {
    sample solution: 1 line(s) of code
}

const Student& Student::operator=(const Student& s) {
    sample solution: 5 line(s) of code
}

Student::~Student() {
    sample solution: 1 line(s) of code
}

void Student::initialize() {
    sample solution: 4 line(s) of code
}

void Student::copy(const Student& s) {
    sample solution: 8 line(s) of code
}

void Student::destroy() {
    sample solution: 4 line(s) of code
}
13 Reverse Iterators [ / 10 ]

Complete the function below named reverse that takes in an STL list as its only argument and returns an STL vector that contains the same list except in reverse order. You should use a reverse iterator and you may not use push_back.

```cpp
reverse(my_list) {
    sample solution: 3 line(s) of code
    while (itr != my_list.rend()) {
        sample solution: 3 line(s) of code
    }
    return answer;
}
```

14 Order Notation [ / 5 ]

Rank these 6 order notation formula from fastest(1) to slowest(6).

```
O(8 \cdot s \cdot w \cdot h)

O((s \cdot w \cdot h)^8)

O((8 \cdot w \cdot h)^8)

O(w \cdot h \cdot 8^s)

O((s + w \cdot h)^8)

O(w \cdot h \cdot s^8)
```