1.1 Stairs Class Declaration [ / 14 ]

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
};
1.2 Stairs Constructor [ / 9 ]

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

1.3 Stairs Destructor [ / 5 ]

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

2.1 Searching for a Word [ / 7 ]

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;
};
```

If the Node chain contains \( n \) elements, what is the running time of the search function?

2.2 Comparing Positions within the Node Chain [ / 8 ]

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.

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

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

```cpp
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.
3  Erase Middles [   / 20 ]

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 3 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.
5 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: 1 line(s) of code
        } else {
            // sample solution: ≤3 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) {
        // sample solution: 4 line(s) of code
    } else {
        // sample solution: 9 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 */
Students::Students() {
}
Students::Students(int courses_per_term_) {
}
```

Sample solution: 3 line(s) of code

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
}
8 Reverse Iterators

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

9 Order Notation

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

- $O(8 \cdot s \cdot w \cdot h)$
- $O((s \cdot w \cdot h)^8)$
- $O((8 \cdot w \cdot h)^8)$
- $O(w \cdot h \cdot 8^8)$
- $O((s + w \cdot h)^8)$
- $O(w \cdot h \cdot s^8)$
For each function or pair of functions below, choose the letter that best describes the program purpose or behavior.

A) infinite loop  
B) factorial  
C) integer power  
D) the answer is 42  
E) function is not recursive  
F) sum of the digits  
G) syntax error  
H) modulo 2  
I) reverse the digits  
J) multiplication  
K) greatest common divisor  
L) other

int mysteryONE(int x, int y) {
    if(y == 0)
        return x;
    else
        return mysteryONE(y, x % y);
}

int mysteryTWO(int x) {
    if (x == 0)
        return 0;
    else
        return mysteryTWO(x/10) + x%10;
}

int mysteryTHREEa(int x);
int mysteryTHREEb(int x) {
    if (x == 0)
        return 1;
    else
        return mysteryTHREEa(x-1);
}

int mysteryFOUR(int x, int y) {
    if (x == 0)
        return 0;
    else
        return y + mysteryFOUR(x-1,y);
}

int mysteryFIVEa(int x, int y) {
    if (x == 0)
        return y;
    else
        return mysteryFIVEa(x/10, y*10 + x%10);
}

int mysteryFIVEb(int x) {
    return mysteryFIVEa(x,0);
}

int mysterySIX(int x) {
    if (x == 0)
        return 1;
    else
        return x * mysterySIX(x-1);
}
11 Collecting Words

Write a function named `Collect` that takes in two *alphabetically sorted* STL lists of STL strings named `threes` and `candidates`. The function searches through the second list and removes all three letter words and places them in the first list in alphabetical order. For example, given these lists as input:

```plaintext
threes: cup dog fox map
candidates: ant banana egg goat horse ice jar key lion net
```

After the call to `Collect(threes, candidates)` the lists will contain:

```plaintext
threes: ant cup dog egg fox ice jar key map net
candidates: banana goat horse lion
```

If there are $n$ and $m$ words in the input lists, the order notation of your solution should be $O(n + m)$.

*sample solution: 15 line(s) of code*
The expected output of the program below is:

chris is a sophomore, his/her favorite color is blue, and he/she has used 1 late day(s).

However, there are a number of small but problematic errors in the DSStudent class code. Hint: This problem’s title is relevant! Only one completely new line may be added (line 6), and the 7 other lines require one or more small changes. These lines are tagged with an asterisk, *. Your task is to rewrite each incorrect or missing line in the appropriately numbered box. Please write the entire new line in the box.

```cpp
class DSStudent {
public:
    DSStudent(std::string n, int y)
        : name(n) {
        int entryYear = y;
    }

    std::string& getName() const {
        return name;
    }

    const std::string& getYear() {
        if (entryYear == 2014) {
            return "freshman";
        } else if (entryYear == 2013) {
            return "sophomore";
        } else if (entryYear == 2012) {
            return "junior";
        } else {
            return "senior";
        }
    }

    void incrLateDaysUsed() const {
        days++;
    }

    int& getLateDaysUsed() const {
        return days;
    }

    std::string FavoriteColor() {
        return color;
    }

private:
    std::string name;
    std::string color;
    int entryYear;
    int days;
};

int main() {
    DSStudent s("chris",2013);
    s.FavoriteColor() = "blue";
    s.incrLateDaysUsed();
    std::cout << s.getName() << " is a " << s.getYear() << ", his/her favorite color is " << s.FavoriteColor() << ", and he/she has used " << s.getLateDaysUsed() << " late day(s)." << std::endl;
}
```
Write a recursive function named `Occurrences` that takes in a sorted STL vector of STL strings named `data`, and an STL string named `element`. The function returns an integer, the number of times that `element` appears in `data`. Your function should have order notation $O(\log n)$, where $n$ is the size of `data`.

Sample solution: 21 line(s) of code
14 Short Answer [ / 8 ]

14.1 What’s Wrong? [ / 4 ]

Write 1-2 complete and concise sentences describing the problem with this code fragment:

```cpp
std::vector<std::string> people;
people.push_back("sally");
people.push_back("brian");
people.push_back("monica");
people.push_back("fred");
std::vector<std::string>::iterator mom = people.begin() + 2;
std::vector<std::string>::iterator dad = people.begin() + 1;
people.push_back("paula");
std::cout << "My parents are " << *mom << " and " << *dad << std::endl;
```

14.2 Fear of Recursion [ / 4 ]

Rewrite this function without recursion:

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

void printer (Node* n) {
    if (n->next == NULL) {
        std::cout << n->value;
    } else {
        std::cout << "(" << n->value << "+";
        printer (n->next);
        std::cout << ")";
    }
}
```

Sample solution: 13 line(s) of code
Ben Bitdiddle is working on a project that stores data with two different data structures: our Vec and dslist classes. Occasionally he needs to convert data from one format to the other format. Alyssa P. Hacker suggests that he write a copy-constructor-like function for each class that takes in a single argument, the original format of the data. For example, here's how to convert data in Vec format to dslist format:

```cpp
// create a Vec object with 4 numbers
Vec<int> v; v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4);
// create a dslist object that initially stores the same data as the Vec object
dslist<int> my_lst(v);
```

Here are the relevant portions of the two class declarations (and the Node helper class):

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

```cpp
template <class T> class Vec {
public:
    // conversion constructor
    Vec(const dslist<T>& lst); 
    /* other functions omitted */
    // representation
    T* m_data;
    unsigned int m_size;
    unsigned int m_alloc;
};
```

```cpp
template <class T> class dslist {
public:
    // conversion constructor
    dslist(const Vec<T>& vec);
    /* other functions omitted */
    // representation
    Node<T>* head_; 
    Node<T>* tail_; 
    unsigned int size_; 
};
```

Ben asks about access to the private member variables of one class from a member function of the other. Alyssa says he can write the functions assuming he has full access to the private member variables. (She promises to teach him how to use the friend keyword to make that work after Test 2.)

### 15.1 Diagrams

First, draw the detailed internal memory representations for a Vec object and a dslist object, each storing the numbers: 1 2 3 4.
Now write the two conversion constructors. You may not use `push_back`, `push_front`, `insert` or iterators in your answer. Instead, demonstrate that you know how to construct and manipulate the low level memory representation.

```cpp
// Sample solution: 13 lines of code

template <class T> Vec<T>::Vec(const dslist<T>& lst) {

// Implementation of Vec constructor

}

// Sample solution: 13 lines of code

template <class T> dslist<T>::dslist(const Vec<T>& v) {

// Implementation of dslist constructor

}
```