1 Dynamically Allocated & Templated Stairs [ / 28 ]

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(2,1,'U');
s.set(3,1,'S');
```

1.1 Stairs Class Declaration [ / 14 ]

First, fill in the blanks in the class declaration:

```cpp
template <class T> class Stairs {
public:
   // constructor
   Stairs(int s, const T& val);

   // destructor
   ~Stairs();

   // prototypes of 2 other important functions related to the constructor & destructor
   Stairs(const Stairs& s);
   Stairs& operator=(const Stairs& s);

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

   /* NOTE: other Stair functions omitted */
private:
   // representation
   int size;
   T** data;
};
```

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).

```cpp
template <class T> Stairs<T>::Stairs(int s, const T& val) {
    size = s;
    data = new T*[s];
    for (int i = 0; i < s; i++) {
        data[i] = new T[i+1];
        for (int j = 0; j <= i; j++) {
            data[i][j] = val;
        }
    }
}
```
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).

Solution:
```cpp
template <class T> Stairs<T>::~Stairs() {
    for (int i = 0; i < size; i++) {
        delete [] data[i];
    }
    delete [] data;
}
```

2 Comparing Linked List Pointers w/ Recursion [ / 32 ]

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.

Solution:
```cpp
Node* search(Node* head, const std::string& word) {
    if (head == NULL) return NULL;
    if (head->word == word) {
        return head;
    }
    return search(head->next, word);
}
```

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

Solution: $O(n)$

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.

Solution:
```cpp
bool compare(Node *a, Node *b) {
```
if (a == NULL) return false;
if (b == NULL) return false;
if (a == b) return false;
if (a->next == b) return true;
return compare(a->next,b);
}

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

**Solution:** \( O(n) \)

### 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!

### 2.3 Implementing insert_between and Maintaining Fast Comparisons

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.

**Solution:**

```cpp
bool insert_between(Node *head, const std::string& before, const std::string& after, const std::string& word) {
    if (head == NULL) return false;
    if (head->next == NULL) return false;
    if (head->word == before && head->next->word == after) {
```
Node* tmp = new Node;
tmp->word = word;
tmp->prev = head;
tmp->next = head->next;
tmp->next->prev = tmp;
tmp->prev->next = tmp;
tmp->distance = (tmp->next->distance + tmp->prev->distance) / 2.0;
return true;
} else {
    return insert_between(head->next,before,after,word);
}
}

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 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 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

Solution:

template <class T>
int erase_middles(std::list<T>& data, const T& val) {
    bool found_first = false;
typename std::list<T>::iterator prev = data.end();
typename std::list<T>::iterator itr = data.begin();
int count = 0;
while (itr != data.end()) {
    if (*itr == val) {
        if (prev != data.end()) {
            data.erase(prev);
            count++;
        }
        if (found_first == false) {
            found_first = true;
        } else {
            prev = itr;
        }
    }
    itr++;
}
return count;
}

4 Debugging Skillz [ / 17 ]

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        E) examine different frames of the stack
B) add a breakpoint        F) reboot your computer
C) use step or next        G) use Dr Memory or Valgrind to locate the leak
D) add a watchpoint        H) examine variable values in gdb or lldb
Solution: E  A complex recursive function seems to be entering an infinite loop, despite what I think are perfect base cases.

Solution: G  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.

Solution: A  I’m unsure where the program is crashing.

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

Solution: D  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.

Solution: B) Once you’ve found the general area of the problem, it can be helpful to add a breakpoint shortly before the crash, so you can examine the situation more closely. C) Once you’ve decided the state of the program is reasonable, you can advance the program one line at a time using next or step into a helper function that may be causing problems. Rebooting your computer is unlikely to fix a bug in your own code.

5 Flipping & Sorting Words [* / 18 *]

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.

Solution:

```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) {
    std::list<std::string>::iterator current = words.begin();
    while (current!= words.end()) {
        std::string flip = reverse(*current);
        if (flip == *current) {
            current = words.erase(current);
        } else {
            std::list<std::string>::iterator tmp = words.begin();
            while (tmp != words.end() && flip > *tmp) {
                tmp++;
            }
            if (tmp == words.end() || flip != *tmp) {
                words.insert(tmp,flip);
            }
            current++;
        }
    }
}
```
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.

**Solution:**

```cpp
class Node {
public:
    Node* head;
    Node* tail;
    Node* next;
    Node* prev;
    int value;
};

void PushFront(Node* &head, Node* &tail, int v) {
    Node* tmp = new Node;
    tmp->value = v;
    if (head == NULL) {
        assert (tail == NULL);
        tmp->next = tmp->prev = NULL;
        tmp->head = tmp->tail = tmp;
        head = tail = tmp;
    } else {
        tmp->prev = NULL;
        tmp->next = head;
        tmp->prev = tail;
        head->prev = tmp;
        head = tmp;
        while (tmp != NULL) {
            tmp->head = head;
            tmp = tmp->next;
        }
    }
}
```
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);
    ~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 */
```

Solution:
```cpp
Student::Student() {
    num_terms = 8;
    courses_per_term = 4;
    initialize();
}

Student::Student(int courses_per_term_) {
    courses_per_term = courses_per_term_;
    num_terms = ceil(32 / float(courses_per_term_));
    initialize();
}

Student::Student(const Student& s) {
    copy(s);
}

const Student& Student::operator=(const Student& s) {
    if (this != &s) {
        destroy();
        copy(s);
    }
    return *this;
}

Student::~Student() {
    destroy();
}
```
```cpp
void Student::initialize() {
    data = new Course* [num_terms];
    for (int i = 0; i < num_terms; i++) {
        data[i] = new Course[courses_per_term];
    }
}

void Student::copy(const Student& s) {
    courses_per_term = s.courses_per_term;
    num_terms = s.num_terms;
    initialize();
    for (int i = 0; i < num_terms; i++) {
        for (int j = 0; j < courses_per_term; j++) {
            data[i][j] = s.data[i][j];
        }
    }
}

void Student::destroy() {
    for (int i = 0; i < num_terms; i++) {
        delete [] data[i];
    }
    delete [] data;
}

8 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.

Solution:
```template <class T>
std::vector<T> reverse(const std::list<T> &my_list) {
    std::vector<T> answer (my_list.size());
    int i = 0;
    typename std::list<T>::const_reverse_iterator itr = my_list.rbegin();
    while (itr != my_list.rend()) {
        answer[i] = *itr;
        i++;
        itr++;
    }
    return answer;
}
```

9 Order Notation [ / 5 ]

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

Solution: 1 \(O(8 \cdot s \cdot w \cdot h)\)  
Solution: 4 \(O((s \cdot w \cdot h)^8)\)  
Solution: 6 \(O((8 \cdot w \cdot h)^8)\)  
Solution: 5 \(O(w \cdot h \cdot 8^s)\)  
Solution: 2 or 3 \(O((s + w \cdot h)^8)\)  
Solution: 2 or 3 \(O(w \cdot h \cdot s^8)\)

NOTE: The ordering of the ‘2’ vs. ‘3’ depends on the relative size of the variables h, w, and s.

If \(w = h = s\) : \((w + w \cdot w)^8 = w^{16} > w \cdot w \cdot w^8 = w^{10}\).
If \(w = h \& s = w^2\) : \((w^2 + w \cdot w)^8 = w^{16} < w \cdot w \cdot (w^2)^8 = w^{18}\).

10 Dynamic Tetris Arrays [ /26]

10.1 HW3 Tetris Implementation Order Notation [ /6]

Grading Note: -1.5pts each unanswered or incorrect.
Match up the Tetris class member functions from HW3 with the appropriate order notation, where \( w \) is the width of the board and \( h \) is the maximum height of any column. Assume the solution is efficient, but uses only the 3 member variables specified in the original assignment (data, heights, and width).

Note: Some letters may be used more than once or not at all.

Solution:

\[
\begin{array}{ll}
\text{c) } & \text{void add_piece(char piece,int rotation,int position);} \\
\text{a) } & \text{int get_width();} \\
\text{e) } & \text{int remove_full_rows();} \\
\text{b) } & \text{int get_max_height();} \\
\text{e) } & \text{void destroy();}
\end{array}
\]

\begin{align*}
\text{a) } & O(1) \\
\text{b) } & O(w) \\
\text{c) } & O(h) \\
\text{d) } & O(w + h) \\
\text{e) } & O(w \cdot h)
\end{align*}

10.2 Tetris Representation Conversion [ /20]

Now let’s revisit the details of the dynamic memory representation for the game of Tetris. Your task is to convert a Tetris board from the column representation we used for HW3 to a row representation. In addition to the three member variables in our HW3 Tetris class: data, heights, and width, we add 2 additional member variables: widths and height. In the column representation we don’t need the heights variable, so it is set to NULL. Each time the board is modified to add Tetris pieces or score full rows the height variable is updated as necessary to store the maximum height of any column.

The diagram on the left shows an example Tetris board first in column representation and then in row representation — the “before” and “after” diagrams for a call to the new Tetris class member function convert_to_row_representation. Note that once in row representation the heights variable isn’t needed and we set it to NULL. The convert_to_row_representation function takes no arguments.

Now write the Tetris class member function convert_to_row_representation as it would appear in the tetris.cpp implementation file. You may assume that before the call the board is in the column representation and the member variables are all set correctly. Make sure your code properly allocates new memory as needed and does not have memory leaks.

Solution:

```cpp
void Tetris::convert_to_row_representation() {
    // allocate the top level arrays
    int* widths = new int[height];
    char** tmp = new char*[height];
    // for each row...
    for (int h = 0; h < height; h++) {
        // calculate the width of each row
        int w = widths[h] = 0;
        for (int w = 0; w < width; w++) {
            // allocate a row of the correct width in the tmp structure
            tmp[h] = new char[widths[h]];
            // fill in the row character data
            for (int w = 0; w < widths[h]; w++) {
                if (heights[w] > h && data[w][h] != ' ') tmp[h][w] = data[w][h];
                else tmp[h][w] = ' ';
            }
        }
    }
}
```
11 Mystery Recursion [ /9]

For each function or pair of functions below, choose the letter that best describes the program purpose or behavior.

A ) infinite loop       E ) function is not recursive
B ) factorial          F ) sum of the digits
C ) integer power       G ) syntax error
D ) the answer is 42     H ) modulo 2
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

Solution: K

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

Solution: J

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

Solution: F

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

Solution: I

```c
int mysteryFIVEa(int x, int y) {
    if (x == 0)
        return y;
    else
        return mysteryFIVEa_integer power

int mysteryTHREEa(int x);

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

Solution: H

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

Solution: B

```c
int mysterySIX(int x) {
    if (x == 0)
        return 1;
    else
        return x *
            mysterySIX(x-1);
}
```

12 Collecting Words [ /18 ]

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:

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

Solution:

```cpp
void collect(std::list<std::string> &threes, std::list<std::string> &candidates) {
    // start an iterator at the front of each list
    std::list<std::string>::iterator itr = threes.begin();
    std::list<std::string>::iterator itr2 = candidates.begin();
    // loop over all of candidate words
    while (itr2 != candidates.end()) {
        // if the candidate is length 3
        if ((*itr2).size() == 3) {
            // find the right spot for this word
            while (itr != threes.end() && *itr < *itr2) {
                itr++;
            }
            // modify the two lists
            threes.insert(itr,*itr2);
            itr2 = candidates.erase(itr2);
        } else {
            // only advance the pointer if the length is != 3
            itr2++;
        }
    }
}
```

13 Constantly Referencing `DSStudent` [ / 12 ]

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
1 class DSStudent {
2 public:
* 3 DSStudent(std::string n, int y)
 4 : name(n) {
* 5 int entryYear = y;
* 6 }
* 7 }
* 8 std::string& getName() const {
 9 return name;
10 }
*11 const std::string& getYear() {
12 if (entryYear == 2014) {
13 return "freshman";
14 } else if (entryYear == 2013) {
15 return "sophomore";
16 } else if (entryYear == 2012) {
17 return "junior";
18 } else {
19 return "senior";
20 }
21 }
*22 void incrLateDaysUsed() const {
23 days++;
24 }
*25 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;
}

Solution: DSStudent(const std::string &n, int y)
Solution: entryYear = y;
Solution: days = 0;
Solution: const std::string& getName() const {
Solution: std::string getYear() const {
Solution: void incrLateDaysUsed() {
Solution: int getLateDaysUsed() const {
Solution: std::string& FavoriteColor() {

14 Efficient Occurrences [ / 22 ]

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`.

Solution:

```cpp
// the recursive helper function
int occurrences(const std::vector<std::string> &data, const std::string &element, int s1, int s2, int e1, int e2) {
    // s1 & s2 are the current range for the start / first occurrence
    // e1 & e2 are the current range for the end / last occurrence (+1)
    assert (s1 <= s2 && e1 <= e2);
    if (s1 < s2) {
        // first use binary search to find the first occurrence of element
        int mid = (s1 + s2) / 2;
        if (data[mid] >= element)
            return occurrences(data,element,s1,mid,e1,e2);
        return occurrences(data,element,mid+1,s2,e1,e2);
    } else if (e1 < e2) {
        // then use binary search to find the last occurrence of element (+1)
        int mid = (e1 + e2) / 2;
        return occurrences(data,element,s1,e1,mid,e2);
    }
    // if they are equal, then the element is in the range
    return 1;
}
```
if (data[mid] > element)
    return occurrences(data, element, s1, s2, e1, mid);
return occurrences(data, element, s1, s2, mid + 1, e2);
} else {
    // the simply subtract these indices
    assert (s1 == s2 && e1 == e2 && e1 >= s1);
    return e1 - s1;
}

// "driver" function
int occurrences(const std::vector<std::string> &data, const std::string &element) {
    // use binary search twice to find the first & last occurrence of element
    return occurrences(data, element, 0, data.size(), 0, data.size());
}

15 Short Answer [ / 8 ]

15.1 What’s Wrong? [ / 4 ]

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

```
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;
```

Solution: Any iterators attached to an STL vector should be assumed to be invalid after a call to push_back (or erase or resize) because the internal dynamically allocated array may have been relocated in memory (or the data shifted). Dereferencing the pre-push_back iterators to print the data is dangerous since that memory may have been deleted/freed.

15.2 Fear of Recursion [ / 4 ]

Rewrite this function without recursion:

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

Solution:
```
void printer (Node* n) {
    int count = 0;
    while (n != NULL) {
        if (n->next != NULL) {
            std::cout << "(" << n->value << "+";
            count++;
        } else {
            std::cout << n->value;
        }
        n = n->next;
    }
    std::cout << std::string(count,')');
}
```
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 Vec {
public:
// conversion constructor
Vec(const dslist<T>& lst);
/* other functions omitted */
// representation
T* m_data;
unsigned int m_size;
unsigned int m_alloc;
};

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.)

### 16.1 Diagrams

First, draw the detailed internal memory representations for a Vec object and a dslist object, each storing the numbers: 1 2 3 4.

**Solution:**

```
m_data: ------ 1 2 3 4
m_alloc: 4
```

```
tail:
head:
size_: 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.

**Solution:**

```cpp
template <class T> Vec<T>::Vec(const dslist<T>& lst) {
    m_alloc = m_size = lst.size();
    if (m_alloc > 0)
        m_data = new T[m_alloc];
    else
```
m_data = NULL;
int i = 0;
Node<T> *tmp = lst.head_
while (tmp != NULL) {
    m_data[i] = tmp->value_
    tmp = tmp->next_
    i++;
}

template <class T> dslist<T>::dslist(const Vec<T>& v) {
    head_ = tail_ = NULL;
    size_ = v.size();
    Node<T> *tmp = NULL;
    for (int i = 0; i < size_; ++i) {
        tail_ = new Node<T>(v.m_data[i]);
        if (tmp != NULL) {
            tail_->prev_ = tmp;
            tmp->next_ = tail_;}
        if (i == 0) head_ = tail_;
        tmp = tail_;}
}