1 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++;
}
}
```

2 “Smart” List Nodes [ / 18 ]

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

3 Dynamically Allocated Student Schedules

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.

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 */

Solution:
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();
}

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;
}

4 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:

```cpp
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;
}
```
5 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 \( w = h \land s = w^2 \) : \( (w^2 + w \cdot w)^8 = w^{16} < w \cdot w \cdot (w^2)^8 = w^{18} \).

6 iCliked or iCoulda Been a ClickerQ [ / 16 ]

Grading Note: -1.5pts each iclicker unanswered or incorrect.

6.1 What is the average case order notation for these three STL vector operations?
   (A) erase: \( O(n) \)  
   (B) erase: \( O(n) \)  
   (C) erase: \( O(n) \)  
   (D) erase: \( O(1) \)  
   (E) erase: \( O(n) \)  
   Solution: C

6.2 Which of the following is false about STL vector iterators?
   (A) The vector iterator syntax is bulkier and more annoying than the \[ \] subscripting syntax and it is ok to avoid using it most of the time.
   (B) To use the erase member function of an STL vector, you must specify an iterator pointing at the element, not just the index.
   (C) It is possible to rewrite a program using vectors to remove the subscripting \[ \] syntax for vectors.
   (D) The .begin() and .end() syntax for sorting vectors are actually iterators!
   (E) vector iterators are more efficient than the \[ \] subscripting syntax.
   Solution: E

6.3 What is the purpose of the typename keyword?
   (A) It’s the same thing as typedef.
   (B) It’s the same thing as class.
   (C) Sometimes when using templates you need to add it to unconfuse the compiler.
   (D) It makes the memory debugger more efficient.
   (E) It’s not even legal C++ code. (It’s only valid in C.)
   Solution: C

6.4 Which of the following statements is not a reason that a vector iterator value should be assumed to be invalid after a push_back, erase, or resize function call on that vector?
   (A) The allocation size of the internal array has changed.
   (B) The data may be shifted within the allocated array.
   (C) The iterator/pointer is attached to an array that has been deleted.
   (D) These functions are expensive linear-time operations.
   (E) The iterator/pointer is no longer pointing at the same data – it is still attached to the array slot that was the old location of the data.
   Solution: D

6.5 Which of the following statements is false about returning values by reference from a function.
   (A) If memory is abundant and we are not worried about speed of execution, there is no reason to return data from functions by reference.
   (B) If you store the result of a function that returns a const reference in a regular (non-alias/non-reference) variable, you may edit that variable because the data was copied when storing it in the variable.
   (C) If a function returns a local variable, the variable must be returned by value (a.k.a. return by copy, not by reference).
   (D) If a member function is const, then to return a member variable by reference, it must be returned by const reference.
   (E) The non-const operator[] for the STL vector and Vec classes returns a “slot” in the array by (non-const) reference. Thus, we may edit that slot by using that return value as an l-value in an assignment operation.
   Solution: A
6.6 Why are the `operator<` and `operator>` comparison operators not available for iterators of the STL list or our `dslist` class?

(A) These operators can only be implemented for data values of pointer type.
(B) There is never a need for these operators since we have `operator==` and `operator!=` defined for list iterators.
(C) These operators cannot be implemented efficiently because list memory is not arranged contiguously like it is for vectors.
(D) The meaning of these operators would be unclear or ambiguous to the users of these classes.
(E) All of the above.

Solution: C

6.7 Which of the following statements is false about templated classes?

(A) The `typedef` syntax allows us to define shorthand for bulky templated (and non-templated) types.
(B) A templated class allows multiple different types to be stored within a single container simultaneously.
(C) Unlike regular non-templated C++ classes, which should generally be split into separate `.h` (declaration) and `.cpp` (implementation) files, both the declaration and implementation of all member functions of templated classes should (generally) be placed in a single `.h` file.
(D) We can make our own custom templated classes by adding the keyword `template` and one (or more!) placeholder types (we often use `T`) inside of angle brackets at the front of the C++ class declaration.
(E) Having our own version of the templated STL `vector` class (named `Vec`) enables us to make custom extensions to the datatype.

Solution: B

6.8 Why does the `dslist` class have a `size` member variable?

(A) If we didn’t store the size, we would be unable to implement the `.size()` member function.
(B) It serves the same purpose as the `m_alloc` variable for our `Vec` class: extra space for anticipated future `push_backs`.
(C) For efficient use of memory.
(D) For efficient runtime speed.
(E) All of the above.

Solution: D

6.9 Which of the following `dslist` or `dslist::iterator` member functions is the most expensive in terms of order notation for running time?

(A) `size()`
(B) `operator++()`
(C) `push_back()`
(D) `erase()`
(E) `copy()`

Solution: E

6.10 Which of the following statements is true?

(A) $100n^2 + n^3 - n + 2 = O(n^2)$
(B) $5n^3 - 2n^2 + 3 = O(n^3)$
(C) $5n + n(n + 1) = O(n)$
(D) $5n^3 - 1000n^{200} - 2^n + 1 = O(n^{200})$
(E) None of the above.

Solution: B

6.11 Which of the following is *not* a common mistake when implementing a linked list data structure?

(A) Not considering special cases of adding / removing at the beginning or the end of the linked list.
(B) Trying to use STL iterators to visit elements of a “homemade” linked list chain of nodes.
(C) Forgetting to implement the subscript operator.
(D) Not correctly setting the `next` pointer from the last node to NULL.
(E) Pointer manipulations that are out of order; e.g., deleting a node before it is appropriately disconnected from the list.

Solution: C

6.12 What is the order notation of the running time and space/memory for the merge sort function we implemented in lecture?

(A) running time: $O(n \log n)$ space/memory: $O(n \log n)$
(B) running time: $O(n)$ space/memory: $O(\log n)$
(C) running time: $O(n \log n)$ space/memory: $O(\log n)$
(D) running time: $O(n)$ space/memory: $O(n \log n)$
(E) running time: $O(n \log n)$ space/memory: $O(n \log n)$

Solution: C

6.13 Which of the previous questions was not actually used during lecture?

(A) Question 6.3
(B) Question 6.6
(C) Question 6.9
(D) Question 6.12
(E) None of the above.

Solution: A