CSCI-1200 Data Structures - Fall 2023
Test 2 – Thursday, October 19th 6:00pm

- This exam has 4 problems worth a total of 100 points (including the cover sheet).
- This packet contains 8 pages of problems numbered 1-8. Please count the pages of your exam and raise your hand if you are missing a page.
- **DO NOT REMOVE THE STAPLE OR SEPARATE THE PAGES OF YOUR EXAM. DOING SO WILL RESULT IN A -10 POINT PENALTY!**
- You may have pencils, eraser, pen, tissues, water, and your RPI ID card on your desk. Place everything else on the floor under your chair. Electronic equipment, including computers, cell phones, calculators, music players, smart watches, cameras, etc. is not permitted and must be turned "off" (not just vibrate).
- Raise your hand if you need to ask a proctor something that is NOT related to one of the questions on the test.
- Please state clearly any assumptions that you made in interpreting a question. Unless otherwise stated you may use any technique that we have discussed in lecture, lab, or on the homework.
- Please write neatly. If we can't read your solution, we can't give you full credit for your work.
- You do not need to write `#include` statements for STL libraries. Writing `std::` is optional. The keyword `auto` is not allowed.

Problem 1: Office Hours Queue. [/35]

Your friend Hazel wanted to simulate the functionalities of the Submitty Office Hours Queue using a linked list with the following `OHNode` class:

```
class OHNode{
  public:
  OHNode(const string& n, const string& q):
    name(n), queue(q), next(NULL) {}
  string name, queue;
  OHNode* next;
};
```

She has already wrote the following code to test the functionalities of her Office Hours queue:

```
1  int main(){
2    // setup
3    OHNode* OHqueue = NULL;
4    cout << "Initializing queue with 4 students" << endl;
5    add_student(OHqueue, "Anna", "debug");
6    add_student(OHqueue, "Chloe", "help");
7    add_student(OHqueue, "Ellie", "debug");
8    add_student(OHqueue, "Sophia", "check");
9    print_queue(OHqueue);
10
11   // remove everyone from the debug queue
12   // this removes Anna and Ellie
13   int result = empty_queue(OHqueue, "debug");
14   cout << "Removed " << result << " student(s)" << endl;
15   print_queue(OHqueue);
16   assert(result == 2);
17
18   // remove everyone from the debug queue again
19   result = empty_queue(OHqueue, "debug");
20   cout << "Removed " << result << " student(s)" << endl;
21   print_queue(OHqueue);
22   assert(result == 0);
23
24   // remove everyone from ANY queue
25   result = empty_queue(OHqueue, "");
26   cout << "Removed " << result << " student(s)" << endl;
27   print_queue(OHqueue);
28   assert(OHqueue == NULL);
29   assert(result == 2);
30 }
```

What appears in line 16 is called an assertion statement. It has the format of **assert(condition);**. And what such a statement will do is, if **condition** is not true, this statement will trigger a crash. Thus we oftentimes use assertion statements to test our programs: when we use **assert(condition);**, we expect **condition** to be true; and if it is not true, we want the program to crash and we get notified that something is wrong. (line 22, line 28, line 29 are also assertion statements.)

After running the code, the following should be printed to STDOUT:

```
Initializing queue with 4 students
       +-------+    +-------+    +-------+    +--------+
Name:  |  Anna |    | Chloe |    | Ellie |    | Sophia |
Queue: | debug | => |  help | => | debug | => |  check |
       +-------+    +-------+    +-------+    +--------+
Removed 2 student(s)
       +-------+    +--------+
Name:  | Chloe |    | Sophia |
Queue: |  help | => |  check |
       +-------+    +--------+
Removed 0 student(s)
       +-------+    +--------+
Name:  | Chloe |    | Sophia |
Queue: |  help | => |  check |
       +-------+    +--------+
Removed 2 student(s)
Queue is currently empty!
```

Since Hazel spent too much time implementing the `print_queue` function (and even more time binge-watching TV shows on Netflix), she wasn't able to implement the `add_student` and `empty_queue`. Please help Hazel finish implementing the remaining two functions.

Part 1: Add Student
Write a **recursive** function called `add_student`, which takes in the first node of the `OHNode` list, a `string` that represents the student's name, and a `string` that represents the queue the student is entering. This function would add an `OHNode` representing the student at the end of the list.

// Implement your add_student function here:

```
void add_student(OHNode*& head, const string& name, const string& queue){
  if(!head){
    head = new OHNode(name, queue);
    return;
  }
  add_student(head->next, name, queue);
}
```

Part 2: Empty Queue
Write an **iterative** function called `empty_queue`, which takes in the first node of the `OHNode` list and a `string` that represents the queue to be emptied. The function will remove and **deallocate** all nodes with the passed in queue name. If the empty string is passed in, it will deallocate all the nodes in the queue. Do NOT write this function recursively.

// Implement your empty_queue function here:

```
// remove all students whose queue name is same as q_name
// if q_name is "", remove all students
int empty_queue(OHNode*& head, const string& q_name){
  int removed = 0;
  while(head && (q_name == "" || head->queue == q_name)){
    OHNode* tmp = head->next;
    delete head;
    removed++;
    head = tmp;
  }
  OHNode* cur = head;
  while(cur){
    while(cur->next && (q_name == "" || cur->next->queue == q_name)){
      OHNode* tmp = cur->next;
      cur->next = tmp->next;
      delete tmp;
      removed++;
    }
    cur = cur->next;
  }
  return removed;
}
```

Part 3: Program Analysis

Assuming that there are n students in the queue and q unique queue names in the queue. What is the Big O Notation for the runtime for the two functions you've just written? Write 2 to 3 concise sentences to justify your answer.

// Write your analysis here:

`add_student`: $O(n)$ time since we need to travel to the end of the list to add the new student in.

`empty_queue`: $O(n)$ time since we need to visit all nodes of the list to see if it needs to be deleted. Since deleting takes $O(1)$ time, the total time complexity would be $O(n)$.

Problem 2: Short Recursion Questions. [/30]

For the following two questions, your functions must be **recursive**. You can write driver/helper functions. Do not declare any additional STL containers or arrays. Do not use any loops. The only library included for these two questions are `<iostream>` and `<list>`.

Part 1: Collatz Conjecture

Consider the following operation for a positive integer x:

- If x is even, then we set $x = x/2$
- If x is odd, then we set $x = 3x + 1$

The Collatz Conjecture, also known as the 3n + 1 problem, claims that for any positive integer, if we apply the above operation enough times, x will reach 1. Write a function called `collatz` which takes in a positive integer x and find the minimum amount of times we need to apply the operation to x in order to get 1. For example, suppose $x = 10$, applying the operation repeatedly, we'll get $10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$. Therefore, the minimum amount of operations we need to apply is 6. *You may assume that the value of x would not exceed the range of* ***int*** Here are a few usage cases for `collatz`:

```
collatz(1); // returns 0
collatz(6); // returns 8
collatz(10); // returns 6
collatz(12); // returns 9
collatz(21); // returns 7
```

// **Implement your collatz function here:**

`int collatz(int x){`
`  if(x == 1) return 0;`
`  if(x % 2 == 1) return 1 + collatz(3 * x + 1);`
`  return 1 + collatz(x / 2);`
`}`

Part 2: Recursive Reverse

Write a function called `reverse` that takes in two `std::list<int>` iterators, first and last. The function will reverse the range, where first is included in the range while last is excluded from the range. Since `<algorithm>` is not included, You may not use `std::reverse` or `std::swap` in this question. As a reminder, you are not allowed to use any loops, create new STL containers or arrays in this question. Here are a few usage cases for `reverse`:

```
std::list<int> l({1, 2, 3, 4});
reverse(l.begin(), l.end());
// l is now {4, 3, 2, 1}
reverse(l.begin(), --l.end());
```

```
// l is now {2, 3, 4, 1}
reverse(l.end(), l.end());
// l is now {2, 3, 4, 1}
reverse(++l.begin(), --l.end());
// l is now {2, 4, 3, 1}
```

// Implement your reverse function here:

```
void reverse(std::list<int>::iterator first,
             std::list<int>::iterator last){
  // no elements in range
  if(first == last) return;
  last--;

  // only one element in range
  if(first == last) return;

  // swap first and last element
  int tmp = *first;
  *first = *last;
  *last = tmp;

  // increment first
  first++;
  reverse(first, last);
```

Problem 3: **++i vs i++.** [/15]

```
#include <iostream>

int main() {
    int a = 5;
    int b, c;

    b = ++a;

    a = 5;

    c = a++;

    std::cout << "Value of a is: " << a << std::endl;
    std::cout << "Value of b is: " << b << std::endl;
    std::cout << "Value of c is: " << c << std::endl;

    return 0;
}
```

This program will print 3 messages to STDOUT, please complete these 3 messages:

```
    Value of a is ____________________


    Value of b is ____________________


    Value of c is ____________________
```

Solution:

Value of a is 6.

Value of b is 6.

Value of c is 5.

Problem 4: **Operators**. [/20]

Part 1: The homework 4 spec lists this as one learning objective: Practice overloading operator<<, and understand why it is a bad idea to make it a member function. And in class we discussed why it is a bad idea. Now let's use the following program to demonstrate why it is a bad idea. [/5]

Complete the missing line in this program, so that the program will print the message "No, I can't keep up with the Kardashians" to STDOUT. Please write one line only, and this one line should call the overloaded operator<<, which is a member function of MyClass.

```
#include <iostream>
#include <ostream>

class MyClass {
public:
    // Overload the output operator as a member function
    std::ostream& operator<<(std::ostream& os) {
        // Define how to output the object
        os << "No, I can't keep up with the Kardashians";
        return os;
    }
};

int main() {
    MyClass obj;


    ____________________________________ // This will call your overloaded operator<<


    return 0;
}
```

Solution (both of the following are correct):

```
obj << std::cout;
```

or

```
obj.operator<<(std::cout);
```

Part 2: Your friend Kylie is trying to concatenate two linked lists, each of these two linked lists is represented by an std::list<int> object. Kylie comes up with this unusual idea, which is overloading the addition operator (+) as a non-member function of std::list. And if the addition operator is overloaded, then she can implement the lists concatenation function like this:

```
// example1: if list1 is {1, 2, 3}, and list2 is {4, 5, 6, 7, 8}, and then this function
    will return a linked list containing {1, 2, 3, 4, 5, 6, 7, 8}
// example2: if list1 is {1, 2, 3, 4}, and list2 is {5, 6}, and then this function will
    return a linked list containing {1, 2, 3, 4, 5, 6}
std::list<int> concatenateLists(std::list<int>& list1, std::list<int>& list2) {
        return list1 + list2;
}
```

In order for this `concatenateLists` function to work, she needs to overload the addition operator. She actually knows what should go inside the function body of the overloaded addition operator, but she is just not sure what should be the return type and what parameters the overloaded addition operator takes. Thus her implementation of this overloaded addition operator is missing one line, please complete this missing line. (This line is also known as the function signature, which defines its name, return type, and parameters.) [/15]

```
--------------------------------------------------------------------------------

{
    std::list<int> result = list1; // copy the first list
    result.insert(result.end(), list2.begin(), list2.end()); // insert elements from
        list2 into result, and thus concatenate the second list
    return result;
}
```

Solution:

```
std::list<int> operator+(const std::list<int>& list1, const std::list<int>& list2)
```

or just

```
std::list<int> operator+(std::list<int>& list1, std::list<int>& list2)
```