CS 32 Worksheet 4

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.
Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

Concepts

Templates, STL

1. You are given an STL set< list<int>* >. In other words, you have a set of pointers, and each pointer points to a list of ints. Write a function that removes the lists with odd sums from the set. The lists with odd sums should be deleted from memory and their pointers should be removed the set. This function should also return the number of lists that are removed from the set. If a list is empty, treat its sum as zero. You may assume that none of the pointers is null.

```
int deleteOddSumLists(set<list<int>*>& s);
```

```
int deleteOddSumLists(set<list<int>*>& s) {
   int numDeleted = 0;

   // iterate over the set
   set<list<int>*>::iterator set_it = s.begin();
   while (set_it != s.end())
   {
      // iterate over each list and get the sum
      int sum = 0;
      list<int>::iterator list_it = (*set_it)->begin();
      list<int>::iterator list_end = (*set_it)->end();
      while (list_it != list_end)
      {
         sum += *list_it;
         list_it++;
      }

      // delete list and remove from set if sum is odd
```

```
        // otherwise, proceed to check the next list
        if (sum % 2 == 1)
        {
            delete *set_it;
            set_it = s.erase(set_it);
            numDeleted++;
        }
        else
            set_it++;
    }

    return numDeleted;
}
```

2. The following code has 3 errors that cause either runtime or compile time errors. Find all of the errors.

```
class Potato {
public:
  Potato(int in_size) : size(in_size) { };
  int getSize() const {
    return size;
  };
private:
  int size;
};

int main() {
  set<Potato> potatoes;  // 1
  Potato p1(3);
  Potato p2(4);
  Potato p3(5);
  potatoes.insert(p1);
  potatoes.insert(p2);
  potatoes.insert(p3);

  set<Potato>::iterator it = potatoes.begin();
  while (it != potatoes.end()) {
    potatoes.erase(it);  // 2
    it++;
  }

  for (it = potatoes.begin(); it != potatoes.end(); it++) {
```

```
    cout << it.getSize() << endl;  // 3
  }
}
```

```
1: The type set<Potato> requires that Potato objects can be
compared with operator<.  Here's an example of how to define <:
bool operator<(const Potato& a, const Potato& b) {
  return a.getSize() < b.getSize();
}

2: After calling erase with the iterator it, it is invalidated.
Instead of incrementing it, the return value of
potatoes.erase(it) should be assigned to it.

3: Iterators use pointer syntax, so the last for loop should
use it->getSize() instead of it.getSize().
```

3. Create a function that takes a container of integers and removes all zeros while preserving the ordering of all the elements. Do the operation in place, which means do not create a new container.
   a. Implement this function taking STL list

```
void removeAllZeroes(list<int>& x){
      //Implement me
}
```

```
void removeAllZeroes(list<int>& x) {
    list<int>::iterator it = x.begin();
    while (it != x.end()) {
      if (*it == 0)
          it = x.erase(it);
      else
          it++;
    }
}
```

   b. Implement the function using STL vectors(JF)

```
void removeAllZeroes(vector<int>& x){
      //Implement me
}
```

```
void removeAllZeroes(vector<int>& x) {
    vector<int>::iterator it = x.begin();
    while (it != x.end()) {
```

```
            if (*it == 0)
                it = x.erase(it);
            else
                it++;
        }
    }
```

4. Implement a stack class *Stack* that can be used with any data type using templates. This class should use a linked list (not an STL `list`) to store the stack and implement the functions *push(), pop(), top(), isEmpty(),* a default constructor, and a destructor that deletes the linked list nodes.

```
template<typename Item>
class Stack {
 public:
  Stack() : m_head(nullptr) {}

  bool isEmpty() const {
    return m_head == nullptr;
  }

  Item top() const {
    // We'll return a default-valued Item if the Stack is
empty,
    // because you should always check if it's empty before
    // calling top().
    if (m_head != nullptr) {
      return m_head->val;
    else
      return Item();
  }

  void push(Item item) {
    Node* new_node = new Node;
    new_node->val = item;
    new_node->next = m_head;
    m_head = new_node;
  }

  void pop() {
    // We'll simply do nothing if the Stack is already empty,
    // because you should always check if it's empty while
    // popping.
```

```
        if (m_head == nullptr) {
          return;
        }
        Node* temp = m_head;
        m_head = m_head->next;
        delete temp;
      }

      ~Stack() {
        while (m_head != nullptr) {
          Node* temp = m_head;
          m_head = m_head->next;
          delete temp;
        }
      }

     private:
      struct Node {
        Item val;
        Node* next;
      };
      Node* m_head;
    };
```

5. What is the output of this program?

```
template <class T>
void foo(T input) {
      cout << "Inside the main template foo(): " << input <<
endl;
}

template<>
void foo(int input) {
      cout << "Specialized template for int: " << input << endl;
}

int main() {
      foo<char>('A');
      foo<int>(19);
      foo<double>(19.97);
}
```

```
Inside the main template foo(): A
Specialized template for int: 19
Inside the main template foo(): 19.97
```

6. Implement the vector class *Vector*. Focus purely on the following functionality:
    a. Implementing *push_back*
    b. Allow the vector to hold any type of data (i.e. use templates)

```
#include<iostream>
using namespace std;

template <typename T>
class Vector {
  public:
    Vector();
    ~Vector();
    void push_back(const T& item);
  private:
    // Total capacity of the vector -- doubles each time
    int m_capacity;
    // The number of elements in the array
    int m_size;
    // Underlying dynamic array
    T* m_buffer;
};

template <typename T>
Vector<T>::Vector()
: m_capacity(0), m_size(0), m_buffer(nullptr)
{}

template <typename T>
Vector<T>::~Vector() {
     delete[] m_buffer;
}

template <typename T>
void Vector<T>::push_back(const T& item) {
  // if space is full, allocate more capacity
  if (m_size == m_capacity)
  {
    // double capacity; special case for capacity 0
```

```
    if (m_capacity == 0)
      m_capacity = 1;
    else
      m_capacity *= 2;

    // allocate an array of the new capacity
    T* newBuffer = new T[m_capacity];

    // copy old items into new array
    for(i = 0; i < m_size; i++)
      newBuffer[i] = m_buffer[i];

     // delete original array (harmless if m_buffer is null)
     delete [] m_buffer;

     // install new array
     m_buffer = newBuffer;
  }

  // add item to the array, update m_size
  m_buffer[m_size] = item;
  m_size++;
}
```

7. What is the output of the following code?

```
template <typename T>
T max(T x, T y)
{
      return (x > y) ? x : y;
}

int main()
{
      std::cout << max(3, 7) << endl;
      std::cout << max(3.0, 7.0) << endl;
      std::cout << max(3, 7.0) << endl;
}
```

On Xcode, it gives the following error messages:

```
int main()
{
    std::cout << max(3, 7) << std::endl;
    std::cout << max(3.0, 7.0) << std::endl;
    std::cout << max(3, 7.0) << std::endl;
    return 0;
}
```

No matching function for call to 'max'

For max, the compiler expects two arguments that are of the same type, as indicated in the template declaration T. In the third call, 3 is an integer and 7.0 is a double, so there is no matching function call for this instance.
Notice that since std namespace has its own implementation of the max function, for the purpose of testing this code, namespace std should not be used.