Queens College, CUNY, Department of Computer Science
Object-Oriented Programming in C++
CSCI 211/611
Summer 2018
Instructor: Dr. Sateesh Mane

due date Friday, August 3, 2018, 11.59 pm

Homework: Templates & STL

- **Experience with other classes has demonstrated that in many cases the source of difficulty is not the mathematics or the programming.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

- **Consult your lab instructor for assistance.**
- **You may also contact me directly, but I cannot promise a prompt response.**
- Please submit your inquiry via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS211 or CS611):

     `StudentId_first_last_CS211_hw_templates.zip`
     `StudentId_first_last_CS611_hw_templates.zip`
  2. The archive should contain one "text file" named "hw_templates.[txt/docx/pdf]" (if required) and cpp files named "Q1.cpp" and"Q2.cpp" etc.
  3. Note that a text file is not always required for every homework assignment.
  4. Note that not all questions may require a cpp file.

General information

- You should include the following header files, to run the programs below.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <string>
#include <cmath>
```

- If you require additional header files to do your work, feel free to include them.
- **Include the list of all header files you use, in your solution for each question.**
- The questions below do not require complicated mathematical calculations.
- If for any reason you require help with mathematical calculations, **ask the lab instructor or the lecturer.**

Q1 Class `Vec_double`

- **Write a class `Vec_double` which is the same as `Vec_int` but supports the data type `double`.**
- Given that you have written `Vec_int` (for homework) and `Vec_Message1` (for Project 2), it should be obvious what to do.

Q2 Templated `Vec` class

- It should be obvious that all of these "Vec" classes can be implemented as special cases of a templated class.
- **Write a templated class `Vec`.**
- **You can substitute `<class T>` in place of `<typename T>`.**

```
template<typename T> class Vec {
public:
  Vec();
  Vec(int n);
  Vec(int n, const T &a);

  Vec(const Vec &orig);
  Vec& operator= (const Vec &rhs);
  ~Vec();

  int capacity() const { return _capacity; }       // inline
  int size() const     { return _size; }           // inline

  T front() const;
  T back() const;

  void clear();
  void pop_back();
  void push_back(const T &a);

  T& at(int n);
  T& operator[] (int n);
  const T& operator[] (int n) const;

private:
  void allocate();
  void release();

  int _capacity;
  int _size;
  T * _vec;
};
```

Q2.1 Sample code

- The following are written for you, to give you an idea how to write non-inline templated function definitions.

```
template<typename T>
Vec<T>::Vec() : _capacity(0), _size(0), _vec(NULL) {}

template<typename T>
void Vec<T>::release()
{
  if (_vec != NULL) {
    delete [] _vec;
  }
  _vec = NULL;
}

template<typename T>
void Vec<T>::allocate()
{
  if (_capacity > 0)
    _vec = new T[_capacity];
  else
    _vec = NULL;
}
```

- The signature of `operator=` is a bit tricky.
- The "this" operator exists for a templated class, and will be correctly implemented in an actual instantiation.

```
template<typename T>
Vec<T>& Vec<T>::operator= (const Vec<T> &rhs)
{
  if (this == &rhs) return *this;
  // etc
}
```

Q2.2 Functions and overloaded operators

- Do not write any functions or overloaded operators for the templated `Vec` class.
- The STL overloads `operator+` to concatenate two strings, but it does not overload a templated `operator+` for `vector`.
- Similarly, there is no templated function "`print`" for `vector`.

Q2.3 Main program

- **Write a main program to test your code.**

  1. The templated class `Vec<int>` should match `Vec_int` exactly.
  2. The templated class `Vec<double>` should match `Vec_double` exactly.
  3. The templated class `Vec<Message1>` should match `Vec_Message1` exactly.

- ***Do you understand how special cases are written first and the templated class is written and tested later?***

Q3 STL #1

- We are given a vector v with some data.
- Suppose v is a vector of strings.
- **We shall employ a set to determine if v contains duplicate data (two or more elements have the same value).**
- You must include the header `<set>`.
- **We shall also employ iterators exclusively to traverse all the containers (vectors and sets).**
- **Declare a set s and traverse the vector using a `const` iterator.**
- **Insert the data into s.**

```
vector<string> v;
// populate v with data

set<string> s;
vector<string>::const_iterator cit;      // const iterator
for (cit = ...; cit != ...; ++cit) {     // figure out the range of iteration
  // insert *cit into s
}
```

- **Because the set contains only unique elements, if the size of s is not equal to the size of v, then v contains duplicates.**

```
if (s.size() != v.size()) {
  cout << "vector contains duplicates" << endl;
}
else {
  cout << "no duplicates" << endl;
}
```

- **Use an iterator `sit` to traverse the set s and print the data in s.**
- You should observe that the data in s is sorted, even if v is not sorted.

Q4 STL #2

- The previous Question Q3 has the weak feature that we traverse the whole vector v before we perform a test for duplicates.
- In fact, as we insert data into the set s, as soon as we detect that the size of s is less than the number of terms we have traversed in v, we know that v contains a duplicate, and we can break out of the loop.
- However, we wish to perform all traversals using iterators exclusively.
- **Declare a set s and a temporary vector `tmp`.**
- **Populate both s and `tmp` as we traverse v.**

```
vector<string> v;
// populate v with data

set<string> s;
vector<string> tmp;                        // tempoary vector
vector<string>::const_iterator cit;        // const iterator
for (cit = ...; cit != ...; ++cit) {       // figure out the range of iteration
  // insert *cit into s
  // push back *cit onto tmp
  if (s.size() != tmp.size()) // break out of loop
}
```

- **After the loop, perform the following test.**

```
if (s.size() != tmp.size()) {
  // there are duplicates
}
else {
  cout << "no duplicates" << endl;
}
```

- If you think about it, the first duplicated value is in `tmp.back()` and its location in v is `tmp.size()-1`.
- **If there are duplicates, print the values of `tmp.back()` and `tmp.size()-1`.**

```
if (s.size() != tmp.size()) {
  // print value of tmp.back() and tmp.size()-1
}
else {
  cout << "no duplicates" << endl;
}
```

Q5 STL #3

- We shall sort the vector v.
- You must include the header `<algorithm>`.
- **Write two comparison functions as follows.**

```
bool comp1(const string &s1, const string &s2)
{
  return (s1 < s2);
}

bool comp2(const string &s1, const string &s2)
{
  return (s1 > s2);
}
```

- **Populate v with data. Then sort as follows.**

```
std::sort(v.begin(), v.end(), comp1);     // first comparison function
```

- **Print the data in v using an `iterator`.**
- **Print the data in v using a `reverse_iterator`.**
- **Populate v with more data. Then sort as follows.**

```
std::sort(v.begin(), v.end(), comp2);     // second comparison function
```

- **Print the data in v using a `const_iterator`.**
- **Print the data in v using a `const_reverse_iterator`.**

Q6 STL #4

- *This is really very naughty, and I should not be teaching you bad habits like this, but we shall play the fool and misuse some operators.*
- We shall sort the vector v.
- You must include the header `<algorithm>`.
- Instead of functions "comp1" and "comp2" we shall overload some operators.
- However, the `string` class already supports comparisons `s1 < s2` and `s1 > s2` so we must do something different.
- **Overload two operators as follows.**

```
bool operator^(const string &s1, const string &s2)
{
  return (s1 < s2);
}

bool operator%(const string &s1, const string &s2)
{
  return (s1 > s2);
}
```

- Why these two operators?
  1. Because . . . *they're both binary operators and they're available for overload.*
  2. This is really terrible. This is operator misuse, not operator overloading.
- **Populate v with data. Then sort as follows.**

```
std::sort(v.begin(), v.end(), operator^);     // comparison function is operator^
```

- **Print the data in v using an `iterator`.**
- **Print the data in v using a `reverse_iterator`.**
- **Populate v with more data. Then sort as follows.**

```
std::sort(v.begin(), v.end(), operator%);     // comparison function is operator%
```

- **Print the data in v using a `const_iterator`.**
- **Print the data in v using a `const_reverse_iterator`.**