CS 104 Midterm Exam **Summer 2016**

Name: Solutions

USC email: ________________________________

In your solutions, you should try hard to write mostly correct C++ syntax. Minor syntax errors will (at most) lose small numbers of points, so don't fret too much about those; but you should also demonstrate a mastery of most C++ syntax. Unless otherwise noted, do not worry about `#include` directives or `using namespace std`.

Your solutions should be as efficient as possible [i.e. if the best known algorithm to achieve a task is O(f(n)) then your algorithm should also be O(f(n))]

Unless otherwise specified you may not use STL data structures.

Question	1	2	3	4	5	6	7	Total
Possible	14	15	12	13	8	13	25	**100**
Earned								

1) Short answer – **Please use at most 2 reasonable length sentences or examples to answer the question.**

a) When overloading the ostream operator '<<' we return an ostream&. Briefly explain why?

To enable chaining multiple << operators
Ex. cout << obja << " hello "; // cout.operator<<(obja).operator<<(" hello ");

b) Briefly explain why the ostream operator '<<' cannot be a member function of the class we wish to output but instead must be a friend function?

The left hand side is not an instance of the class but is instead an ostream

cout << obja; // cout.operator<<(obja) which would look for a member of ostream and not of the type of obja

c) For each of the descriptions indicate the most specific ADT appropriate (don't say LIST if a QUEUE is applicable) to store the following information and show what types would be used for the template arguments (e.g. map<string, int> or stack<double> or something like that). You may provide a brief explanation if you feel it will help us understand your intentions.

i. All the TV station call signs (e.g. KNBC, WGN, KABC, etc.) and allow for quick checks if a given code is being used.

set<string> // the strings would be the call signs. Sets allow log(n) lookup/find

ii. Teams in the NCAA top 25 football rankings. It should be easy to access a team given its ranking (i.e. 1 = USC, 2 = Alabama, etc.)

list<string> (or list< ?Team? > (list preserves order and allows random access (i-th)

iii. The files are stored in a specific folder/directory on a disk. Given the folder name we want to efficiently retrieve the files in that folder.

Map<string, list<string/file> (key is the folder name, value could be a set as well)

iv. Orders that need to be shipped by Amazon (assume an 'Order' class is already defined). Workers will want to retrieve an order from the data structure and then will go process it.

queue<Order> (should be processed in FIFO order)

2) Runtime

a) Analyze the runtime of the function, f1. Derive an expression, T(n,m) and then solve to find as tight a bound as you can using big-O/Ω/Θ notation.

```
void f2(int k)
{
  for(int i=0; i < k; i++){
    for(int j=0; j < i; j++){
      /* do something that take O(1) time */
    }
  }
}

void f1(int n, int m)  // Find the total runtime of this function
{
  if(n < 1) return;
  /* do something that take O(1) time */
  else {
    /* do something else that takes O(1) time */
    f2(m);
    f1(n/2, m);
  }
}
```

Runtime of f2:

$$f2(k) = \sum_{i=0}^{k-1}\sum_{j=0}^{i-1}\theta(1) = \sum_{i=0}^{k-1}\theta(i) = \theta(k^2)$$

Runtime of f1 (will execute log(n) times since we divide n by 2 each recursion)

$$T(n,m) = \theta(m^2) + T\left(\frac{n}{2}, m\right) = \theta(m^2) + \theta(m^2) + T\left(\frac{n}{4}, m\right) = \cdots$$
$$= \theta(m^2 * \log(n))$$

b) What is the amortized run-time of a call for ObjB::f1(). Determine an expression in big-O notation in terms of **n,** where n represents the size of the internal vector<int> data member.

```
class ObjB {
public:
  // Constructor
  ObjB(const vector<int>& other, int n)
  { this->n = n;  m = other; }

  void f1(int v){
    m.push_back(v);
    if(m.size() == n){
      f2();
    }
  }
private:
  void f2(){
    set<int> s;
    vector<int>::iterator it;
    for(it=m.begin(); it != m.end(); it++){
      s.insert(*it);
    }
    cout << m.size() - s.size() << endl;
    m.reserve(2*n);  // resizes the array to be of capacity/size=2*n
    n = 2*n;
  }
  // Data members
  vector<int> m;
  int n;

};
```

Cost of f1 when m.size() < n: O(1)
Cost of f1 when m.size() == n is O(1) + T_{f2}(n):

$$T_{f2}(n) = [total\ set\ insertion\ time] + [resize\ time] = \left[\sum_{i=1}^{n} \theta(\log(i))\right] + [\theta(n)]$$

$$= \left[\sum_{i=1}^{n} O(\log(n))\right] + [\theta(n)] = [O(n * (\log(n))] + [\theta(n)] = O(n * \log(n))$$

Amortized time = Total time of n calls / n

When m.size() == n, we need to call f2() and pay O(n*log(n)).
After resizing we can make n-1 calls and not need to invoke f2 and thus only pay O(1) for each call which totals to O(n) for n-1 calls.

Total cost = O(n*log(n) + n)
Amortized over n calls yields an amortized cost of (O(log(n)).

3) Examine the code below and show what will be printed by the program in the provided box. Look over the code carefully so you do not miss any **cout** statements which are in the member functions.

a) Show what will be printed by the code in main() shown in the boxed inset to the right.

```
class A {
public:
  int n; // data member
  A()
  { cout << "A()" << endl;
    n = 2;
  }
  void f1() {
    cout << "A::f1" << endl;
    for(int i=1; i <= n; i++){
      f2(i);
    }
    cout << endl;
  }
  virtual void f2(int x)
  { cout << x * 2 << " "; }
};

class B : public A {
public:
  B() {  }
  void f1() {
    cout << "B::f1" << endl;
    for(int i=n; i >= 1; i--){
      f2(i);
    }
    cout << endl;
  }
  void f2(int x) { cout << -x << " "; }
  void f3(int i)
  { cout << "B3" << endl; }
};

class C : public B {
public:
  C() { }
  void f2(int x) { cout << x*3 << " "; }
  virtual void f3(int i)
  { cout << "C3" << endl; }
};
```

```
int main()
{
  A* p1 = new C();
  p1->f1();

  B* p2 = new B();
  p2->f3(3); p2->f1();

  B* p3 = new C();
  p3->f3(2); p3->f1();

  A* p4 = p2;
  p4->f1();
  return 0;
}
```

Program Output:
(Note all the couts…ones in constructors, f1(), and print().

A()
A::f1
3 6
A()
B3
B::f1
-2 -1
A()
B3
B::f1
6 3
A::f1
-1 -2

b) Which class(es) above is/are abstract (if any)?
None (no classes have pure virtual functions)

4) Recursion and Linked Lists

Billy Bruin wanted to use a linked list to implement a set where no duplicates are allowed. Unfortunately, he didn't know how. Tommy Trojan said he could do it and even implement the **insert** function recursively. Show how Tommy Trojan could achieve this by writing a recursive **`set_insert`** function that inserts a value into a linked list only if the value is not there already, and then returns true. If the value is already in the list, then do not insert it and return false.

Your function must run in O(n). You may define helper functions should that be desirable…prototype them below and implement them further down.

```
struct Item
{ // Constructor
  Item(int v, Item* n) : val(v), next(n) { }
  int val;
  Item* next;
};
/* prototype any helper functions here..then write them below */
bool set_insert_helper(Item*& head, int value);

bool set_insert(Item*& head, int value)
{
  if(head == NULL){
    head = new Item(value, NULL);
    return true;
  }
  else {
    return set_insert_helper(head, value);
  }
}

bool set_insert_helper(Item*& head, int value)
{
  if(head->val == value){
    return false;
  }
  else if(head->next == NULL){
    head = new Item(value, NULL);
    return true;
  }
  else {
    return set_insert_helper(head->next, value);
  }
}
```

```
// Alternate solution without helper
bool set_insert(Item*& head, int value)
{
  if(head == NULL){
    head = new Item(value, NULL);
    return true;
  }
  else if(head->val == value){
    return false;
  }
  else {
    return set_insert_helper(head->next, value);
  }
}
```

5) **Sorting**

a.) Given the array below show the result [contents of the array] of an **insertion** sort after each iteration/pass of the outer for loop (assume i is the counter for the outer loop/pass). The results after iteration 0 have been filled in for you and are complete.

index	0	1	2	3	4
data	15	11	10	20	13

Data after iteration					
i=0	15	11	10	20	13
i=1	11	15	10	20	13
i=2	10	11	15	20	13
i=3	10	11	15	20	13
i=4	10	11	13	15	20

6) A jagged matrix is one where the number of columns in each row is not uniform (i.e. the 1st row could have 3 columns and the 2nd row could have 10 columns). We could simply use a vector<vector<int> > for this structure. However, imagine a spreadsheet like application where we may delete a whole row of data. Rather than copy data from the following rows up one spot which could require memory allocations, copying, etc. it would be more efficient if each row were dynamically allocated and we simply kept a pointer to each row in our data structure so that we only move/copy the pointers when a row is deleted. This is what we've done for class JaggedMatrix shown below

a.) Prototype the copy constructor and then implement it on the next page. Also implement the `operator-=` member function (on the next page) which follows the specification in the comment line above its prototype.

```
class JaggedMatrix {
public:
  JaggedMatrix() {}
  // Add a copy constructor prototype here:
  JaggedMatrix(const JaggedMatrix& other);

  // Complete...adds a row to the bottom of the matrix with numcols
  //  columns. Values will be initialized to 0
  void addRow(int numcols);
  // deletes the row with index rownum, shifting subsequent rows up
  JaggedMatrix& operator-=(int rownum);
private:
  vector<vector<int> *> data;
};
```

```
void JaggedMatrix::addRow(int numcols)
{
  data.push_back(new vector<int>);
  int r = data.size()-1;
  for(int i=0; i < numcols; i++){
    data[r]->push_back(0);
  }
}

// Write your copy constructor here

JaggedMatrix(const JaggedMatrix& other)
{
  // implementations may vary…a deep copy is needed
  for(int i=0; i < other.data.size(); i++){
    data.push_back(new vector<int>(*(other.data[i])));
  }
}

JaggedMatrix& JaggedMatrix::operator-=(int rownum){
  if(rownum < data.size()){
    delete data[rownum];
    // option 1:  use vector's member function
    // data.erase(data.begin() + rownum);

    // option 2:  do it yourself
    for(int i=rownum+1; i < data.size(); i++){
      data[i-1] = data[i];
    }
    data.pop_back();
  }
  return *this;
}
```

b.) Does this class need a destructor? (**Yes** / **No**)

c.) Does this class need to implement an operator=? (**Yes** / **No**)

7) **[Background]** Suppose we are writing a simple text editor where the entire contents are a single string and the user performs **insert** and **erase** actions on the string. You are given an **`Action`** struct that contains information about "insert" and "erase" actions and a **`Str`** class that maintains the internal string contents and **applies** actions on that string. **Assume this code is provided and works**.

```
struct Action {
  int insert_erase;      // 0 = insert / 1 = erase
  int pos;               // position at which to insert/erase
  string str_to_insert;  // Used only for insert
  int numchars_to_erase; // Used only for erase
};

class Str {  // Complete class and may not be modified
public:
  Str() { curr = ""; }  // Default constructor
  // Correctly applies the provided action to update 'curr'
  void apply(const Action& a);
protected:
  string const & get() { return curr; }
  string curr;
};
```

[Your task and Example] On the next page you will develop a class named: **`VStr`**. This class should acts as a "Versioned String". A versioned string should work like git/github where a user can **add/remove** a sequence of Actions but not have them be applied to the internal string until a **commit** is performed. At that point all actions since the last commit are applied in the order they were added and the resulting string is saved and given a version number. Then, the user can **revert** the string to a prior version if desired.

To write the class described on the next page you may use:

- primitive types (string, int, double, etc.)
- Str object(s)
- Either or both of the two classes shown below implementing a templated Queue and Stack class. Assume these work and can be freely used.

```
template <typename T>
class Queue {
 public:
  Queue();
  void push(const T& item);
  void pop();
  T const& front() const;
  bool empty() const;
  int size() const;
};
```

```
template <typename T>
class Stack {
 public:
  Stack();
  void push(const T& item);
  void pop();
  T const& top() const;
  bool empty() const;
  int size() const;
};
```

Below is an example execution of the desired behavior. Assume the following example code uses two helper functions are provided to create actions easily:

```
Action make_insert(int pos, string str_to_insert); // returns an insert action
Action make_erase(int pos, int numchars_to_erase); // returns an erase action
```

```
  VStr s1;
  s1.addAction( make_insert(0, "aa") );
  s1.commit();  // Applies actions and saves Version 1 = "aa"
  s1.addAction( make_insert(0, "b") );   // if committed: "baa"
  s1.addAction( make_insert(3, "b") );   // if committed: "baab"
  s1.addAction( make_erase(2, 1) );      // if committed: "bab"
  s1.commit();  // Applies actions and saves Version 2 = "bab"
  s1.addAction( make_insert(0, "a"));    // if committed: "abab"
  s1.removeFirstAction();                // removes previous insert
  s1.commit();  // No actions to apply but saves Version 3 = "bab"
  s1.addAction( make_insert(0, "a"));
  s1.addAction( make_insert(0, "a"));
  s1.revert(0); // Removes actions added since last commit
  s1.revert(1); // Reset the string to Version 1 = "aa"
```

Recall you may use the `Str`, `Queue` and `Stack` classes shown on the previous page. Complete the class header (data members, etc.) here. The requirements of the class are shown in the comments above the various member functions. You will write the member functions on the next page:

```
class VStr _: private/protected Str__  // Complete if needed
{public:
  VStr(); // Default constructor

  // Adds an action to be applied on commit
  void addAction(const Action& a);

  // Removes the first action added.  Has no effect if no actions have
  //  been added since the last commit (or initialization)
  void removeFirstAction();

  // Performs the added actions and maintains a saved version of the
  //  result string in case of future revert operations. Also returns the
  //  committed string
  string commit();

  // Reverts and returns the current string (assume no negative version number)
  // If version is 0, leaves current string as is (i.e. same revision)
  //   but removes all added actions since the last commit
  // If version is positive (i.e. +n) reverts to revision n
  //      setting curr to that revision.
  string revert(int version);
 private:  // Add data members here

  Stack<string> versions; // stack of versions
  Queue<Action> actions; // queue of actions

};
```

```
VStr::VStr()
{

// can be blank for default constructable data members

}

void VStr::addAction(const Action& a)
{
  actions.push(a);
}



void VStr::removeFirstAction()
{
  actions.pop();
}


string VStr::commit()
{
  while(!actions.empty()){
    apply(actions.front());
    actions.pop();
  }
  versions.push(curr);
  return curr;
}


string VStr::revert(int version)
{
  while(!actions.empty()) actions.pop();
  if(version > 0){
    while(versions.size() > version){
      versions.pop();
    }
  }
  curr = versions.top();
  return curr;
}
```