Test 2: CPS 100E

Owen Astrachan          Susan Rodger

November 16, 1995

Name: __________________________________________

Honor code acknowledgement (signature) ________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Value</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>14 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>16 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>52 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 12 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.
Declarations for the stack and queue class

template <class Etype>
class Stack // : public AbsStack<Etype>
{
public:
    Stack(); // construct empty stack
    ~Stack(); // destructor

    const Stack & operator=( const Stack & Rhs ); // assign Rhs to self

    void Print() const; // print stack to cout

    void Push( const Etype & X ); // push X onto top of stack
    void Pop(); // pop top element
    const Etype & Top( ) const; // return top element (NO pop)
    bool IsEmpty( ) const; // return true if empty, else false
    bool IsFull( ) const; // return true if full, else false
    void MakeEmpty(); // empty stack (no elements)

private:
};

template <class Etype>
class Queue // : public AbsQueue<Etype>
{
public:
    Queue(); // construct empty queue
    ~Queue() {} // destruct (nothing now)

    const Queue & operator=( const Queue & Rhs ); // assign

    void Enqueue( const Etype & X ); // insert X (at rear)
    void Dequeue(); // remove first element
    void MakeEmpty(); // clear queue to 0 elements

    const Etype & GetFront( ) const; // return front (still there)
    bool IsEmpty( ) const; // true if empty else false
    bool IsFull( ) const; // true if full else false
    void Print() const; // queue printed to cout
    int NumEnqueues() const; // return # calls of Enqueue

private:
};
PROBLEM 1:  \(\text{(MaxStacks (10 points))}\)

Write the function \(\text{MergeStacks}\) whose header is given below. \(\text{MergeStacks}\) creates a new stack, \(\text{result}\). The top of \(\text{result}\) should be the top of \(\text{first}\), followed by the top of \(\text{second}\), and then alternating between \(\text{first}\) and \(\text{second}\). If \(\text{second}\) empties before all items are popped from both stacks, the remaining items are popped from \(\text{first}\). For example, stacks of ints are shown below with the result of merging \(\text{first}\) and \(\text{second}\).

\[
\begin{array}{|c|c|c|}
\hline
\text{first} & \text{second} & \text{result} \\
\hline
50 & 22 & 50 \\
45 & 45 & 45 \\
35 & 32 & 32 \\
100 & 42 & 100 \\
200 & 52 & 200 \\
\hline
\end{array}
\]

Because stacks \(\text{first}\) and \(\text{second}\) are passed as const reference parameters, local copies must be made that can be popped. Note that the \(\text{Stack}\) class has an assignment operator (you can assume it has a copy constructor as well).

```cpp
void MergeStacks(const Stack<int>& first, const Stack<int>& second, 
                 Stack<int>& result)
```

// precondition: \# elements in first >= \# elements in second, result is empty
// postcondition: result = merge of first and second (starting with first)
Consider writing a class to manipulate mathematical tuples, or ordered collections of numbers. For example, (2,3) is a pair, or 2-tuple, and (1,3,-5) is a 3-tuple (tuples are sometimes called vectors in mathematics, but with a Vector class this is confusing so we’ll use tuple).

Tuples contain double values; operations on tuples include adding two tuples (corresponding entries are added) and multiplying a tuple by a double (each entry in the tuple is multiplied by the double).

\[(1.0, 2.3, -3.6) + (2.5, -1.0, 4.4) = (3.5, 1.3, 0.8)\]

\[2.0 \times (3.5, -2.1, 10.3, 11) = (7.0, -4.2, 20.6, 22.0)\]

\[(3.5, -2.1, 10.3, 11) \times 2.0 = (7.0, -4.2, 20.6, 22.0)\]

The class declaration below (from tuple.h) uses a Vector of double values to implement tuples.

class Tuple
{
public:
    Tuple(int size); // # elements = size, all elements = 0
    Tuple(const Tuple & t); // copy constructor

    Tuple & operator += (const Tuple & t); // add a tuple
    Tuple & operator *= (double value); // multiply by scalar double

    int Size() const; // returns size

private:
    int mySize; // # elements in myList
    Vector<double> myList; // the elements in a tuple
};

Tuple::Tuple(const Tuple & t) : mySize(t.mySize), myList(t.myList) // postcondition: I’m a copy of t (object constructed is copy)
{
}

int
Tuple::Size() const // postcondition: returns size of tuple
{
    return mySize;
}
You will implement several member functions and reason about alternate implementations of the Tuple class.

**Part A (5 points)**
Implement the member function for `operator +=`. If the tuples being added are NOT the same size, then nothing should be added, e.g., `(2,3) += (3,4,5)` does NOT alter `(2,3)`.

```cpp
Tuple & Tuple::operator += (const Tuple & t)
// precondition: my tuple value = a, t.Size() == Size()
// postcondition: my tuple value = a + t, the value a+t is returned
```

**Part B (3 points)**
Implement the non-member function for `operator +`. Assume that `+=` works as specified, regardless of what you wrote in Part A. Tuples that are NOT the same size should NOT be added (as for `+=`).

```cpp
Tuple operator + (const Tuple & t1, const Tuple & t2)
// precondition: t1.Size() == t2.Size()
// postcondition: returns t1 + t2
```
Part C (3 points)
In the current implementation (with NO operator []) there is no way to assign a value to a tuple other than all zeros (using one of the constructors). Describe how to make it possible to assign a Vector of doubles to a Tuple, i.e., the code below would make t represent (6.0,6.0,6.0,6.0,6.0). Be brief

```cpp
Vector<double> fiveTwos(5,2.0);
Tuple t(5);
t = fiveTwos;
t *= 3;
```

Part D (3 points)
Someone re-implements the class Tuple using linked-lists rather than vectors. All client programs still work, and the performance of the arithmetic operators +, *, etc. remains very good. However, a new operator is added to the class, the header is given below.

```cpp
double & Tuple::operator [](int index)
// precondition: 0 <= index and index < mySize
// postcondition: returns the index-th entry of the tuple
// i.e., if t = (4,3,7), then t[0] = 4,
```

Code using this newly added operator is drastically slowed when the Tuple class is re-implemented using linked-lists. Briefly, why is the performance bad for operator [] when linked lists are used rather than vectors.
**PROBLEM 3**: *(The club of clubs (16 points))*

Consider the following struct declarations used to represent a linked list of clubs, each club containing a linked list of members.

```c
struct Member
{
    string name;
    Member * next;

    Member(string newname, Member * newnext = 0) // constructor
    {
        name = newname;
        next = newnext;
    }
};

struct Club
{
    string clubName;
    Member * list;
    Club * next;

    Club(string newname, Member * newlist = 0, Club * newnext = 0) // constructor
    {
        clubName = newname;
        list = newlist;
        next = newnext;
    }
};
```

- duke
- tennis
- cps
- debate
- juggling
- dhatt
- sharp
- yoho
- slepack
- roh
- tye
- magee
- sharp
- bloom
- howell
Part A: 4 points

Write the function `IsMember` whose header is given below. `IsMember` returns true if a person with the given name is in the list. For example, `IsMember(duke->list,"roh")` returns true because "roh" is in the tennis club (the club pointed to be `duke->list`) and `IsMember(duke->next->list,"magee")` returns false because "magee" is NOT in the cps club (the club pointed to by `duke->next->list`).

```c
bool IsMember(Member * list, string name)
// postcondition: returns true if name is in list
// otherwise returns false
{
}
```

Part B: 4 points

Complete the function `FindClubList` whose header is given below. `FindClubList` returns a pointer to the first student member in a specified club. For example on the previous page, `FindClubList(duke,"juggling")` returns a pointer to the node with "slepack", the first member in the "juggling" club list.

```c
Member * FindClubList(Club * clubList, string clubname)
// postcondition: returns a pointer to the list of members in the
// club clubname (returns 0 if no club named clubname)
{
}
```
Part C: 8 points

Complete the function CloneClubs whose header is given below. CloneClubs creates a new club. The new club is formed by combining the members of two other clubs. For example using the diagram on page 7, CloneClubs("duke", "cps", "juggling", "music") creates a new club called music with the members from the cps and juggling clubs. The members of the music club are: sharp, tye, howell, and slepack (note that sharp is a member of both clubs, but should just appear once in the new club). The cps and juggling clubs should still remain, do not destroy them.

In writing CloneClubs you may call the function IsMember and the function FindMember you wrote in Parts A and B; assume these functions works as specified regardless of what you wrote in Parts A and B.

```c
void CloneClubs(Club * & clubList, string club1, string club2, string newName)
// precondition: no club named newName is in clubList
// postcondition: a club named newName is added to clubList, the
//                members of the new club are all members from
//                clubs club1 and club2, with no member listed twice
```

```
PROBLEM 4:  (Lots of Q’s 12 points)

Consider writing a MultiQueue class that might be used to process class registrations (if ACES goes down, for example). Students submit requests and the requests are processed in order: first come first served. However all seniors are processed before junior, juniors before sophomores, and sophomores before freshmen. The MultiQueue class (declaration and implementation shown at the end of the test) is implemented using four queues, one per class. The declarations and implementations of some member functions are given on the next page. You must do two things

Part A: (6 points)

Write the function Dequeue that removes the first student from the multiqueue. The first student is the first senior; if there are no seniors, then the first junior, etc.

```cpp
void MultiQueue::Dequeue()
// postcondition: "first" student is removed from queue
// EXCEPTION raised if queue is empty
{
    EXCEPTION( isEmpty( ), "Queue is empty" );
}
```
Part B: (6 points)
Write a new function `Withdraw` (the header is given below) that removes student `s` from the queue. All other students should be left in the same relative order after student `s` is removed. Students are identified by social security number (the `ssnum` field).

```cpp
void MultiQueue::Withdraw(const Student & s)
```