# Test 1: CPS 100

Owen Astrachan

February 12, 1996

Name: ____________________________________________

Honor code acknowledgement (signature) ______________________________________

| | value | grade |
|---|---|---|
| Problem 1 | 8 pts. | |
| Problem 2 | 18 pts. | |
| Problem 3 | 13 pts. | |
| Problem 4 | 16 pts. | |
| Extra | 6 pts. | |
| TOTAL: | 57 pts. | |

This test has 8 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 50 minutes.

**PROBLEM 1 :** (*Vocabulary:* 8 points)

For each of the words/phrases below, circle the definition that is the best description as it pertains in the context of computer science, programming, and C/C++.

1. *copy constructor*
   (a) A member function used to construct an object (an instance of a class) from another object.
   (b) A member function called when one object is is assigned to another using `operator =`.
   (c) A member function that serves as a template for all other constructors to eliminate duplicated code.
2. *this*
   (a) A pointer used in member functions to prevent local variables in the function from aliasing data members of the class.
   (b) A pointer used in member functions to refer to the object that is being affected by the member functions, e.g., the object on the left of the `.` when a member function is invoked on an object.
   (c) A pointer used in the destructor of a class to ensure that all private data members are deleted appropriately.
3. *friend*
   (a) A way of circumventing the use of `const` for constant objects when a member function isn't declared as a `const` function.
   (b) A way of circumventing the use of `private` for specific functions and classes to have access to the private section of another class.
   (c) A way of circumventing the use of templates so that a non-templated class can be implemented as a templated class.
4. *QuickSort*
   (a) A **worst-case** $O(n \log n)$ sort with good average case performance.
   (b) An **average-case** $O(n \log n)$ sort whose worst-case is $O(n^2)$, but with good performance in practice.
   (c) A theoretically optimal sort whose performance in practice is not as good as selection sort for sorting large amounts of data

**PROBLEM 2 :** (*Linkin' Logs:* 18 points)

The following declaration for a `Node` is given, linked lists are implemented **without** dummy-header nodes.

```
struct Node
{
    string info;                        // stores data in node
    Node * next;                        // points to next node
    Node(string val, Node * ptr = 0)    // constructor
       : info(val), next(ptr)
    {}
};
```

**Part A (4 points)** The function *CountWord* is intended to return the number of occurrences of *word* in the linked-list pointed to by *first*. The function always returns 1 more than the correct answer. Describe how to fix *CountWord* so that it works as intended.

```
int CountWord(Node * first, const string & word)
// postcondition: returns # of occurrences of word in list pointed
//                to by first
{
    int count = 0;
    if (first == 0)
        return 1;
    else
    {
        if (first->info == word)
             count++;
        return count + CountWord(first->next);
    }
}
```

**Part B (6 points)** Write the function *Copy* whose header is given below; *Copy* returns a pointer to a newly constructed linked list that is a copy of the list pointed to by the parameter *first*. (hint: it's easier to do this recursively than iteratively)

```
Node * Copy(Node * first)
// postcondition: returns a copy of the list pointed to by first
{






}
```

**Part C (8 points)** In this problem assume that lists **are implemented using dummy/header nodes**. Write the function *MoveToFront* that finds the **first** node containing *word* and unlinks this node from the list, making it the first node of the list. For example, if *list* represents `("bear","ape","cat","dog")`, the call `MoveToFront(list,"cat")` makes *list* represent `("cat","bear","ape","dog")`.

```
void MoveToFront(Node * list, const string & word)
// precondition: list has a dummy/header node
// postcondition: first occurrence of word in list has been moved
//                to the front of list as the first node
{









}
```

**PROBLEM 3 :** (*Jerry's house in Philadelphia:* 13 points)

**Part A (3 points)** What is the value of the postfix expression: `13 8 6 2 * + 7 - +`

**Part B (8 points)** A `Stack` class can be implemented using a vector, a linked list, or using other ways of storing the data in the stack. Using only the member functions *IsEmpty()*, *MakeEmpty()*, *Top()*, *Pop()*, and *Push()*, write the assignment operator whose header is given below (the header file for the stack class in the book is provided at the end of the test). The general method you should use for `s = rhs` is:

- make the stack being assigned to empty
- make a reversed-duplicate of the stack `rhs` by popping all the elements of `rhs` onto a temporary stack
- pop the elements from the temporary stack to the stack being assigned to

**You must also complete the while loop of the function ReverseStack so that it works as intended.**

```
template <class Etype>
ReverseStack(const Stack<Etype> & s, Stack<Etype> & copy)
// precondition: s is a1, a2, ..., an (a1 on top of stack)
// postcondition: copy is an, ..., a2, a1 (an on top of stack)
{
    Stack<Etype> temp(s);      // copy of s to pop
    copy.MakeEmpty();          // make sure copy is empty

    // loop until temp is empty, popping all elements onto copy









}
template <class Etype>
Stack<Etype>::operator = (const Stack<Etype> & rhs)
{
    Stack<Etype> temp;                  // temporary stack
    MakeEmpty();                        // make myself empty

    // copy elements from rhs onto temp (reversed), then back to myself
    // so that I'm a copy of rhs [note: use ReverseStack]







}
```

**Part C (2 points)** What is the complexity (using big-Oh) of the assignment operator described in Part B for assigning a stack of $n$ elements, briefly justify your answer.

**PROBLEM 4 :** (*Sequences:* 16 points)

A member function *Length* for the sequence class could be implemented simply as

```
int Sequence::Length()
// postcondition: returns number of elements in sequence
{
    return mySize;
}
```

However, if you don't have access to the class you cannot add a new member function.

**Part A (5 points)** Write a **non-member** function *Length* that returns the number of elements in the sequence $s$. The header files for the *Sequence* and *SequenceIterator* classes are provided at the end of the test.

```
int Length(const Sequence & s)
// postcondition: returns number of elements in s
{



}
```

**Part B (8 points)** For two sequences $s$ and $t$ by definition $s < t$ if the sequences have the same length and each element of $s$ is less than the corresponding element of $t$. For example

| s | t | $s < t$ |
|---|---|---|
| (1,3,5) | (2,4,7,8) | false, not the same length |
| (1,3,5) | (2,4,7) | true |
| (1,3,5) | (2,4,5) | false, 5 isn't less than 5 |

(continued)

Write the function `operator <` whose header is given below. In writing the function assume that *Length* works as specified, regardless of what you wrote in part A.

```
bool operator < (const Sequence & lhs, const Sequence & rhs)
// postcondition: returns true if lhs < rhs, otherwise returns false
{









}
```

**Part C (3 points)** Write the function that implements `operator >`, defined to be consistent with the definition of `operator <`. The body of `operator >` **MUST** be only one statement. In writing the function, assume that `operator <` works as specified, regardless of what you wrote in Part B.

```
bool operator > (const Sequence & lhs, const Sequence & rhs)
// postcondition: returns true if lhs > rhs, otherwise returns false
{


}
```

**PROBLEM 5 :** (*Extra Credit:* 6 points)

A linked list can be sorted using quicksort using the code below. Write the function *Partition* so that all nodes in the list pointed to by the first parameter are moved into two other lists (the second and third parameters). If the first node of the first parameter list has info field value X, then all nodes whose info fields are less than or equal to X should be moved to the list pointed to by the second parameter and all nodes whose info fields are greater than X should be moved to the list pointed to by the third parameter. When *Partition* is done, the first parameter should be an empty list. (Nodes are implemented as they are in Problem 2, the info field is a string.)

```
void QuickSort(Node * & list)
// postcondition: list is sorted
{
    if (list != 0)
    {
        Node * first  = 0;
        Node * second = 0;
        Partition(list,first,second);          // partition into two lists
        QuickSort(first);                      // sort each sublist
        QuickSort(second);
        Node * temp = first;                   // find end of first sublist
        if (temp != 0)                         // and attach it to second
        {
            while (temp->next != 0)
                temp = temp->next;
            temp->next = second;               // attach to second
            list = first;
        }
        else
        {
            list = second;
        }
    }
}
```