Name: Sample Solutions Student #: _____

Engineering 4892 — Data Structures
Quiz 1

Dr. D. K. Peters

June 4, 2003

Instructions: Answer all questions. Write your answers on this paper. This is a closed book test, no textbooks, notes, calculators or other aides are permitted.
Total points: 50

---

1. [4 points] Consider a class `Foo`, written by Jeff, that is to be used by a function `bar`, written by Akbar.

   a) What is the term for the set of assumptions that Akbar can make about `Foo`?

   Interface

   b) If a pre-condition for a method in `Foo` is false, whose fault is it?

   Akbar

   c) If a post-condition for a method in `Foo` is false, whose fault is it?

   Jeff

   d) Who needs to know the secrets of `Foo`.

   Jeff

2. [6 points] Give a brief definition for each of the following.

   a) Procedural abstraction

   Describing subroutines (procedures, functions) in terms of what they do rather than how they do it.

   b) Data abstraction

   Describing objects (classes, types) in terms of the externally observable behaviour rather than concrete implementation.

   c) Class invariant

   A predicate that is true whenever no member function is executing (i.e., before and after each function executes).

3. [10 points] The following is a partial declaration of a template for a stack class. (The question is on the next page.)

```
template <class T>
class Stack
{
  public:
  // A stack (LIFO list) of type T.
  // Modeled by:
  //   S sequence of T
  //
  // INV: none

  // Local types
    enum Status { Ok, Overflow, Fail };
    // It might also be useful to have a 'Underflow' status, but the
    // pre-condition for pop allows us to ignore those cases.

  // Constructors
    Stack(int _size = DEFAULT_SIZE);
    // Post: S = _
  // ...


  // Accessors
    T top() const;
    // Pre: |S| > 0
    // Post: Result = last element of S

    bool empty() const;
    // Post: Result = (|S| = 0)

    Status getStatus() const { return err; }
    // Post: Result = status of last access

  // Mutators
    Status push(T x);
    // Post: |S| < size -> (S' = Sx /\ Result = Ok) /\
    //       |S| = size -> (S' = S /\ Result = Overflow)

    void pop();
    // Pre: |S| > 0
    // Post: S' = S with last element removed

    Stack<T>& operator=(const Stack<T>& r);
    // Assignment operator
    // Post: this is a copy of r

  private:
    // Representation:
    // topIndex >= 0 -> S = { s[0], s[1], ... s[topIndex] } /\
    // topIndex = -1 -> S = _

    // INV: -1 <= topIndex < size

    static const int DEFAULT_SIZE; // Default for stack size
    T *s;         // Pointer to begining of the stack.
    int size;     // Maximum |S|
    int topIndex; // Index of top item in the stack.
    Status err;   // Status of the last call.
};
```

Give an implementation of the push method for this class, as follows:

```
/*****************************************************************
 * push -- put an element on the top of the stack. Set err to Ok if
 *         successful, Overflow if the stack is full.
 *
 * Post: topIndex < size-1 -> (s'[topIndex'] = x /\ Result = err' = Ok
 *                            /\ topIndex' = topIndex + 1)
 *       /\ topIndex = size-1 -> Result = err' = Overflow
 *****************************************************************/
template <class T>
Stack<T>::Status
Stack<T>::push(T x)
{
  if (topIndex < size-1) {
    topIndex++;
    s[topIndex] = x;
    err = Ok;
  } else {
    err = Overflow;
  }
  return err;
}
```

4. [10 points] In assignment 2 we saw how a univariate polynomial can be represented by a list of its coefficients, so, for example, the polynomial $5.6x^2 + 3.4x + 1.2$ is represented by the list $\{1.2, 3.4, 5.6\}$. The following is a partial declaration of a `Polynomial` class that uses this representation. (The question is on the next page.)

```
class Polynomial {
  // A class for representing univariate polynomials of the form
  //        d         d-1         d-2
  //    c  x  + c    x    + c    x    + ... c  x + c
  //     d       d-1         d-2             1      0
  //
  //
  // Representation:
  //    c -- a sequence of Reals which are the coefficients
  //
  public:

  // Constructors
    Polynomial() { /* You don't need to implement this one */ }
    // no-argument constructor.
    // Post: c = _

    Polynomial(int d, double *coefs);
    // constructor.
    // Parameters: d - the degree of the polynomial
    //             coefs - pointer to an array containing the coefficients
    //                     in ascending order (i.e., coefs[0] is c_0). Note
    //                     that the number of coefficients should be one
    //                     more than the degree.
    //
    // Pre: d >= 0
    // Post: d = 0 -> c = _ /\ d != 0 -> c = coefficients from coefs

  // Accessors
    double operator() (double x) const;
    // Pre: this.degree() >= 0 (i.e., the polynomial is defined)
    // Post: Result = the polynomial evaluated at x

    int degree() const;
    // Post: Result = the degree of the polynomial (i.e., the exponent for
    //                the highest order coefficient)
    //                -1 if the Polynomial is undefined

  // Mutators
  // ...

  private:
  // Representation:
  //   coef contains the coefficients, starting with the lowest order
  //   coefficient (i.e., c_0).

    std::list<double> coef;
};
```

Give an implementation for the evaluation operator as specified below.

```
/**********************************
 * compute the value of the polynomial at x
 *
 * Pre: this.degree() >= 0 (i.e., the polynomial is defined)
 * Post: Result = the polynomial evaluated at x
 */
double
Polynomial::operator() (double x) const
{
  double result = 0.0;
  double xToTheI = 1.0; // The ith power of x
  list<double>::const_iterator i = coef.begin();
  while (i != coef.end()) {
    result +=  *i * xToTheI;
    i++;
    xToTheI *= x;
  }
  return result;
}
```

5. [20 points] A *reverse polish notation expression* (rpn) is an expression in which the operators follow the operands, e.g., "3 7 +". This form is used by some calculators (most notably those made by Hewlett Packard) and has the advantage that it eliminates the need for brackets and precedence. For example the infix expression "$(1 + 2) * (3 + 4)$" is written in rpn "1 2 + 3 4 + $*$". Give an implementation of a function, as specified below, that uses one or more (STL) `stack`s or `list`s to evaluate an rpn expression. You may assume that the expression contains only **single digit** integers (i.e., 0–9), the operators + - * / and space. The result is to be computed using integer math.

```
#include <stack>
#include <cstdlib>
#include <string>
using namespace std;
/*****************************************************************
 * rpnEvaluate -- evaluate an rpn (postfix) expression.
 *
 * Pre: exp contains an expression in RPN form containing only single digit
 *      integer operands, the operators + - * /, and space.
 * Post: Result = the value of the expression computed using integer math.
 *****************************************************************/
int
rpnEvaluate(const string& exp)
{
  stack<int> operands;

  for (unsigned int i = 0; i < exp.length(); i++) {
    if (isdigit(exp[i])) {
      operands.push(exp[i]-'0');
    } else if (!isspace(exp[i])) {
      int op2 = operands.top();
      operands.pop();
      int op1 = operands.top();
      operands.pop();
      switch (exp[i]) {
        case '+':
          operands.push(op1 + op2);
          break;

        case '-':
          operands.push(op1 - op2);
          break;

        case '*':
          operands.push(op1 * op2);
          break;

        case '/':
          operands.push(op1 / op2);
          break;
      }
    }

  }
  return operands.top();
}
```