1. (15 points)

(a) Rewrite the `MergeArrays` function so that it does all of the work inside the single `while` loop, eliminating the two `for` loops at the end. Perhaps surprisingly, this will only involve minor changes to the while loop condition and the two if conditions.

**Solution:**

```c
void MergeArrays( double a[], int m, double b[], int n, double result[] )
{
    int i=0, j=0;  // indices into the two sorted arrays
    int k=0;      // index into the result array

    // Each time through the following loop, the next smallest value
    // the two arrays is added to the result array. If one of the
    // arrays is exhausted, then the values are automatically taken
    // from the other. The loop ends when both arrays has been
    // exhausted.

    while ( i<m || j<n ) {
        if ( j==n || (i<m && a[i] < b[j]) ) {
            result[k] = a[i];
            ++ i;
        }
        else {
            result[k] = b[j];
            ++ j;
        }
        ++ k;
    }
}
```
(b) Starting from this rewrite, rewrite MergeArrays again to merge three sorted arrays simultaneously.

**Solution**

```c
void
ThreeWayMerge( double a[], int num_a,
               double b[], int num_b,
               double c[], int num_c,
               double result[] )
{
    int i=0, j=0, k=0; // indices into the three sorted arrays
    int n=0;

    while ( i<num_a || j<num_b || k<num_c ) {

        if ( ( j==num_b && k==num_c ) || // both other arrays exhausted
             ( i<num_a && // a is not exhausted and
               ( j==num_b || a[i] < b[j] ) && // b is exhausted or a’s
               ( k==num_c || a[i] < c[k] ))) // c is exhausted or a’s
        {
            result[n] = a[i];
            ++ i;
        }
        else if ( ( i==num_a && k==num_c) || // similar to above conditions
                  ( j<num_b &&
                    ( i==num_a || b[j] < a[i] ) &&
                    ( k==num_c || b[j] < c[k] )))
        {
            result[n] = b[j];
            ++ j;
        }
        else
        {
            result[n] = c[k];
            ++ k;
        }
    }
}
```
2. **(15 points)** Carrano & Prichard, page 321, number 6. You only need to write a main program, similar to the one given in class.

**Solution**

```cpp
int main()
{
    char c;
    StackOfChar paren_stack;

    while ( cin >> c )
    {
        if ( c == '{' || c == '[' || c == '(' )
        {
            paren_stack.push( c );
        }
        else if ( c == '}' || c == ']' || c == ')' )
        {
            if ( paren_stack.isEmpty() )
            {
                cout << "Too many right " << c << " characters\n";
                return 0;
            }
            else
            {
                char c2;
                paren_stack.pop( c2 );
                if ( !( ( c == '(' && c2 == ')) ||
                      ( c == '[' && c2 == ']' ) ||
                      ( c == '{' && c2 == '}' ) ) )
                {
                    cout << "Mismatched " << c << " and " << c2 << "\n";
                    return 0;
                }
            }
        }
    }

    if ( !paren_stack.isEmpty() )
    {
        cout << "Unmatched opening character\n";
        return 0;
    }
}
```
{ cout << "Too many left parens\n"; }
else {
    cout << "Everything is balanced.\n";
    return 0;
}

3. (15 points) How can you use one or more stacks to simulate a queue? Answer by outlining how the enqueue, dequeue, and getFront operations of the queue must occur. You may only use the public stack and queue interfaces.

Solution: Maintain items in a stack, s, where the top of the stack is the front of the queue. The getFront and dequeue then are just like getTop and pop. enqueue requires a separate stack. Call it temp. To do the enqueue operation, pop all items off s, pushing them onto temp. Then, push the new item onto the empty stack s. Finally, pop each item off of temp and push it back onto s.