Section Solutions 2: C basics

Problem 1 : String warmup

```
/*
 * Function: containsDouble
 * ------------------------
 * Returns TRUE if the string passed in contains a double letter
 * (like book, grass, bottle).  The two letters must be consecutive
 * to count as a double letter.
 */
bool containsDouble(string str)
{
    int i;

    for(i = 1; i < StringLength(str); i++) { // Note that we start
from 1!
        if (IthChar(str, i-1) == IthChar(str, i)) {
           return TRUE;
        }
    }
    return FALSE;
}
```

Problem 2 : Mode.

As with many algorithmic problems, there are many ways to accomplish the task o finding a mode. Probably the most straightforward mechanism is to simply count the times each element appears:

```
/* Helper function declarations */
static int CountElement(int value, int array[], int size);


/*
 * Function: Mode
 * --------------
 * This implementation just goes through the array and counts the
number of
 * times each element appears, keeping track of the current maximum
value.
 */
static int Mode(int array[], int size)
{
    int mode, max, count, i;

    max = 0;
    for (i = 0; i < size; i++) {
        count = CountElement(array[i], array, size);
        if (count > max) {
            max = count;
            mode = array[i];
```

```
        }
    }
    return (mode);
}


/*
 * Function: CountElement
 * ----------------------
 * Returns the number of times the first argument appears in the
array.
 */
static int CountElement(int value, int array[], int size)
{
    int count, i;

    count = 0;
    for (i = 0; i < size; i++)
        if (value == array[i]) count++;
    return (count);
}
```

Alternatively, one could sort the array first and then just check the length of each run of consecutive elements. This is slightly more complicated, but certainly more efficient, as shown in this version of `Mode` (which assumes that we've already defined a standard `Sort` function):

```
static int Mode(int array[], int size)
{
    int start, count, i, max, mode;

    Sort(array, size);
    start = max = 0;
    count = 1;
    for (i = 1; i < size; i++) {
        if (array[i] == array[start]) {
            count++;
        } else {
            if (count > max) {
                mode = array[start];
                max = count;
            }
            start = i;
            count = 1;
        }
    }
    if (count > max) mode = array[start]; // Takes care of the last
run.
    return (mode);
}
```

Problem 3 : Quick Pointer Questions.

a) What are the types of the two variables in the following declaration?

```
double *p1, p2;
```

`p1` *is a pointer to a double, but* `p2` *is an actual double—remember that the * applies to the variable, not the type.*

b) What happens if you try to dereference a `NULL` pointer?

Well, nothing good, that's for sure. What you're actually doing is trying to read or write what is in address 0, which probably isn't what you want to do. Writing to this address can make strange errors happen, and could even crash your machine.

c) For any variable `x`, is the expression `*&x` basically a synonym for `x`?

Yes.

d) For any variable `x`, is the expression `&*x` basically a synonym for x?

No. If x is not of a type which is a pointer to another type, the compiler will flag that as an error, giving a message such as "Pointer Required," since you're trying to dereference it.

Problem 4 : General pointer problems

a)

i)

```
int a[10];
*(a+4) = 5;
```

ii)

```
int a[5][10];
*(a+(3*10)+4) = 15
```

iii)

```
int **a;
InitArray(&a);
*((*(a+3))+4) = 15;
```

b) This function does not allocate memory for a new string like the current Concat does - instead it merely appends all characters from the second string to the end of the first string. This means that there's a good possibility of a memory error should the string s1

not have enough memory allocated to it to hold both its own characters and the characters from s2. In addition, even if there was enough memory allocated, normal Concat does not affect the two strings passed into it in any way, whereas this version destroys the string s1 when used.

If this function was used in place of the currently implemented Concat, then logical errors could occur due to the latter bug, ie code that would expect the string s1 to be unchanged after returning from the function call would no longer correctly work. In addition, the memory error would garbage memory, causing a computer crash or at least unexpected results.