COMP2401—Tutorial 5
Pointers and Arrays

Learning Objectives

After this tutorial, you will be able to:

- Manipulate pointers
- Use pointers to structures
- Manipulate arrays using pointers

Submit your tutorial in a tar file t5.tar at the end of the tutorial.

Tutorial

Download the tar file t5.tar and extract the files.

1 Problem 1: String comparison using recursion

Purpose: gaining experience with string pointers and taking advantage of "call by value"

In this part of the tutorial you will code a recursive function, `myStrCmp()`, for comparing two strings. Create two files mystr.c and mystr.h, which will contain the code and the function prototype respectively. Use the file str_cmp_main.c to test your code.

Input:

Input consists of two strings s1 and s2 (given as address to the memory location)

Output:

None

Return:

-1 if string s1 should appear before string s2 in lexicographic order.

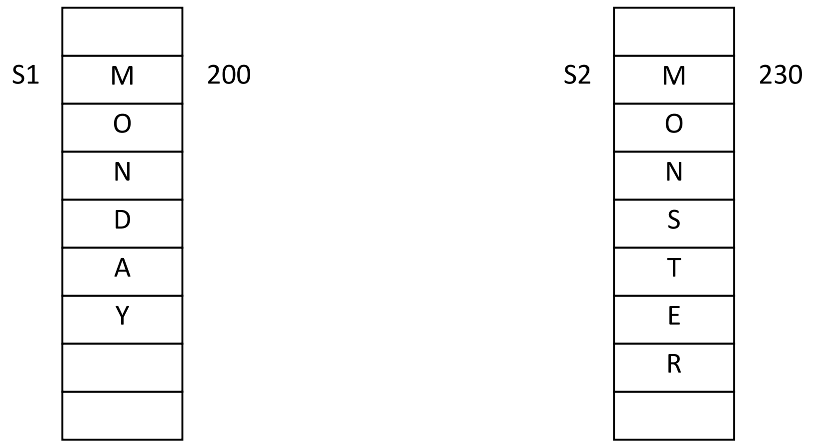
0 if string s1 is the same as s2

1 is s2 appears before s1 in lexicographic order.

Assumption:

The value of s1 and/or s2 is not NULL

For example: if s1 = "Monday" and s2 = "Monster" then a call to `myStrCmp(s1, s2)` would return -1 because Monday should appear before Monster in a lexicographic order.



Algorithm layout

Compare the characters of the two strings one at a time until either one of the strings is empty (all characters were exhausted), or the two characters are different.

Compare the last two characters that were tested and return the result of the comparison.

In this function we take advantage of the fact that all parameters passed to the function are "called by value". Namely, the parameters are a local copy. This means that we can change the value of the pointer (namely, the address stored in the pointer without external impact).

Comparing the current characters pointed to by s1 and s2 is done by using the '*' operator. For example, the statement if (*s1 == *s2) {"do something"} compares the characters for equality. In the example above *s1 is the first character at position s1[0] which is 'M'.

Checking whether s1 is an empty string is carried out by comparing the character pointed to by s1 to the character '\0' (the sentinel). For example if (*s1 != '\0') {"do something" } or if (*s1 != 0) {"do something"}

In order to advance the addresses stored in the pointers to point to the next character one can use the '++' operator (e.g., advancing the pointer s1 using s1++ will change the content of s1 to 201 and it will point to the letter 'O').

Function prototype

```
int myStrCmp(char *s1, char *s2);
```

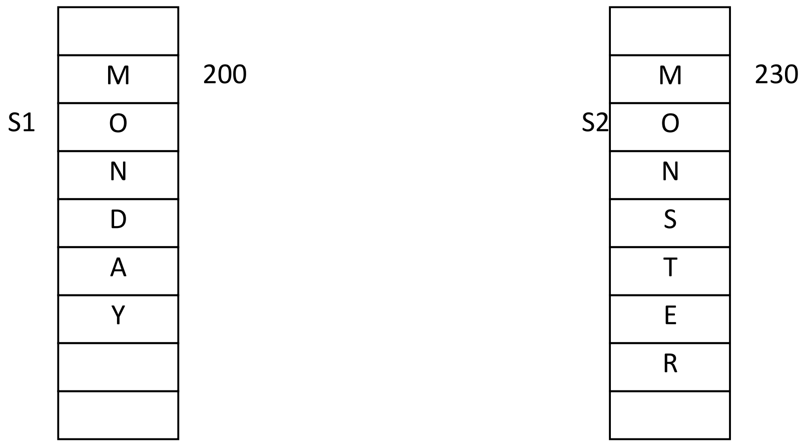
Pseudo code + code

```
Int myStrCmp(char *s1, char *s2)
{
        // recursion condition
        // if (s1 is not empty and s2 is not empty and the characters pointed to by s1 and s2 are the
        // same) then recurse by advancing s1 and s2 to point to the next character
        if (*s1 != '\0' && *s2 != '\0' && *s1 == *s2) {
                return(myStrCmp(s1++, s2++));
        }
        // one of the strings may be empty or they contain different characters
        // return -1 if *s1 precedes *s2 in the lexicographic order
        if (*s1 < *s2) return(-1);
        else return (*s1 > *s2);
}
```

Code the function and test it. Most likely you will get a segmentation violation. Try to understand why you receive segmentation violation and fix the function. Take 5 minutes to work on the problem.

Did you identify the problem?

The problem is that the recursive call to `myStrCmp()` with the parameters s1++, and s2++ is not correct. The postfix operator ++ advances the pointers s1 and s2 only after the call to myStrCmp is complete. Here, however, we want the pointers s1 and s2 to advance to the next location before the recursive call to `myStrCmp()` as shown in the figure below.



Solution:

There are two options of achieving it:

a. Advance each pointer independently and then call `myStrCmp`
```
s1++;
s2++;
myStrCmp(s1, s2);
```
b. Use the prefix operator ++ to do so. Namely call `myStrCmp(++s1, ++s2);`

2 Problem 2: Initializing and searching an array

Purpose: To become familiar with: 1) passing structures as pointers, 2) accessing structure fields using "->" operator, 3) using pointer arithmetic and 4) taking advantage of "call by value"

To do:

1. The file find_struct.c contains a declaration of a struct emp, a main() function and a function that populates the emp struct. Review the code and make sure that you understand it.

2. The `main()` function allocates two arrays of size `MAX_EMPLOYEES: empArr` contains `emp structs` and `empPtrArr` contains pointers to `employee structs`.

3. Initialize (populate) the array records with employee data by calling the provided function `populateEmployee(…)`. Pass to the function the address of current record to be initialized. Review the code of `populateEmployee(…)`.

4. Code a function that compares a single employee record against a given key (in this case it is a family name). The function specifications are:

   Prototype:

   ```
   int cmpEmployee(struct emp *p, char *familyName)
   ```

   input:
   ```
   familyName - family name of employee to be searched
   p - a pointer to an employee record
   ```

   Output:
   None

   Return:
   ```
   0 - if family name of employee in the provided record does not match the familyName
   1 - if family name of employee in the provided record matches the familyName
   ```

   Note 1: use the function myStrCmp() from Section 2 to compare the keys
   Note 2: function prototype is already in the C file
   Note 3: use the operator -> to access the fields inside struct emp

5. Code a function that searches the array emp for an employee by family name.

   The function specifications are:

Prototype:

```
struct emp * findEmployee(struct emp **arr, int arraySize, char
*familyName);
```

input:
arr – an array of pointers to employees
arraySize – the number of elements in the array
familyName – familyName to be used as a key

Output:
None

Return:
NULL – if no matching record was found
a pointer to a struct in the array that matches the family name

Pseudo Code

```
// iteratively traverse the array using pointer arithmetic.  Namely by augmenting the value of
// arr by one  at every iteration.  Note that here we can take advantage of the fact that the pointer is
// a call by value and we can use pointer arithmetic.
// Also note that you will have to take care of the precedence order between the "*" and the "->"
operators

    // compare the family name of the record with the key that was given.

    // if a record with a matching name is found then print the record (see below)
```

Record printing

```
firstName   familyName
 salary=    years of service =
```

E.g.,

```
Dina Door
salary= 28500.00  years of service = 9.00
```

6. Call the function from main()

Submit your tutorial work in a tar file t5.tar!