CPE 111 – Programming with Data Structures
International Sections - January 2021
Laboratory Exercise 2

Objective

This lab is intended to give you practice analyzing the impact of using different data structures for the same problem.

Instructions

1. Download the demo program **stringBubble.c** from the course website. You will also need **timeFunctions.c** and **timeFunctions.h**. To make things simpler, download the **Makefile**, too. Also download the input file **dracula.txt**.

The easiest way to do this is to download the zip file **lecture2Demos.zip** which contains all the files in the directory. To unpack this archive, use this command:

```
unzip lecture2Demos.zip
```

2. Build the current version of **stringBubble** by typing

```
export OSTYPE=linux          (only necessary the first time – only on Linux)
make stringBubble            (or you can type in the gcc command directly).
```

3. Run **stringBubble** on the file **dracula.txt**. Record the total time and the time per item printed by the program on a piece of scrap paper.

4. Copy **stringBubble.c** to **newBubble.c** and modify the new version of the program to use an array of ***char* pointers***, instead of an array of ***char*** items. To see how this works, you may want to look at **treeSort.c**.

- Change the variable ***data*** to be a **char**** data type.
- Dynamically allocate **count** elements that are **sizeof(char*).**
- Remove the **getStringPointer** function and all references to it. Just use ***data*** as a normal array.
- As you read elements from the input file, use **strdup()** to allocate a new string with a copy of the contents read.
- Store the pointer returned from **strdup** in the current element of the ***data*** array.
- Change the **swapString** function so that instead of actually copying data from one array element to another, it simply reassigns the pointers. See sample code below.
- When you are done sorting, free the dynamically allocated information. You have to do this in two steps. First loop through the array freeing each character string. Then free the array itself. (hint: look at **treeSort.c**)

5. Try to compile **newBubble.c.** Fix any compile errors you find. You should add lines to the makefile to build **newBubble** in order to save you some typing.

6. Run **newBubble** on the **dracula.txt** file. Check the output file to make sure that the results are correct. Record the total time and the time per item printed by the program.

Is the new program faster than the old one? If so, why do you think this is true? (Think about this question. I may ask it in the quiz next week!)

7.Upload **newBubble.c** using the link on the CPE111 home page.

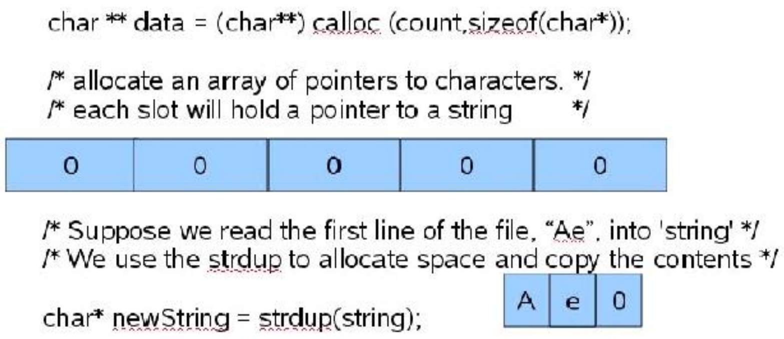
Sample code for swapString function

```
void swapString(char* array[],int i, int j)
{
        char* tempPtr = NULL;
        tempPtr = array[i];
        array[i] = array[j];
        array [j] = tempPtr;
}
```

Comparison of Data Structures in stringBubble and newBubble

The notes from Lecture 2 include a picture of the way that the data are stored in **stringBubble**. All the characters are dumped into one long array (dynamically allocated so that we can handle any number and size of strings to sort). We use pointer arithmetic to calculate the location of each successive string, so the array acts like a two dimensional array.

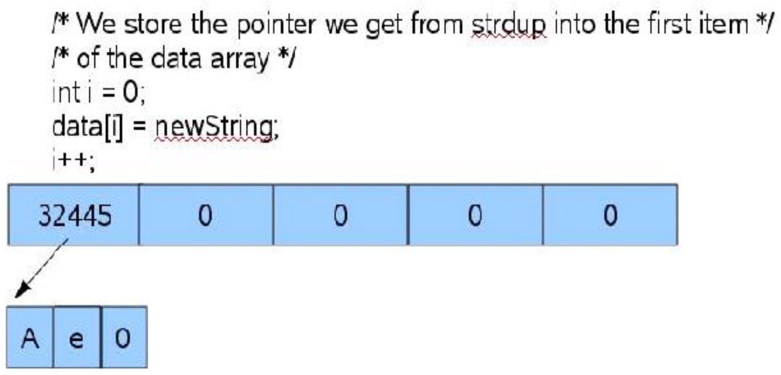
For **newBubble**, we have two levels of dynamic allocation. First we allocate space for the whole set of count strings.



```
char ** data = (char**) calloc (count,sizeof(char*));

/* allocate an array of pointers to characters. */
/* each slot will hold a pointer to a string       */
```

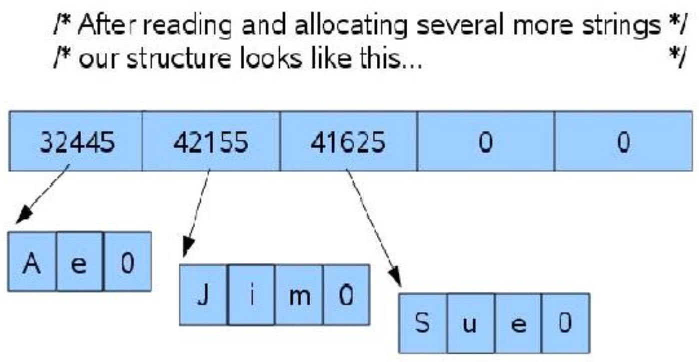
```
/* Suppose we read the first line of the file, "Ae", into 'string' */
/* We use the strdup to allocate space and copy the contents */

char* newString = strdup(string);
```

Then we read each line in turn and allocate just enough space for that string, using **strdup**. The **strdup** function also copies the contents of the string (including the terminating zero byte) to the newly allocated storage. Then we save resulting pointer in the next element of the data array.



```
/* We store the pointer we get from strdup into the first item */
/* of the data array */
int i = 0;
data[i] = newString;
i++;
```

We continue to read each string, duplicate it, and store its address, filling up the data array.



```
/* After reading and allocating several more strings */
/* our structure looks like this...                   */
```

We build the data structure in two steps. Thus we also need to free it in two steps. First we loop through the data array, freeing each string.

```
    for (i = 0; i < count; i++)
  {
     if (data[i] != NULL)
          free(data[i]);
     }
```

Then we can free the array itself.

```
   free(data);
```