# Introduction to Computers and Programming LAB-12 2015/12/22

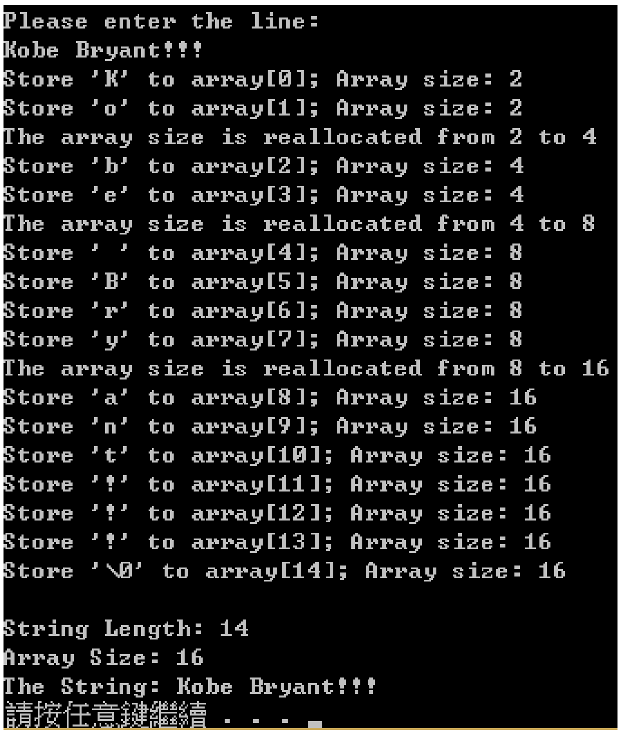
- ✧ Your input/output must follow our format in examples
- ✧ Please sign in the computer with your own account of CS Computer Center if you have it, and save your codes in your own space in case of something bad happens on your computer.
- ✧ <u>If you cannot finish it in time, you should demo your lab work at next lab time.</u>
  <u>(The deadline is 21:30).</u>

## 1. Unlimited String

In real life applications, sometimes we may have to implement a variable-length string, so that it can hold any number of characters entered by users. This kind of "unlimited string" is often implemented as follows:

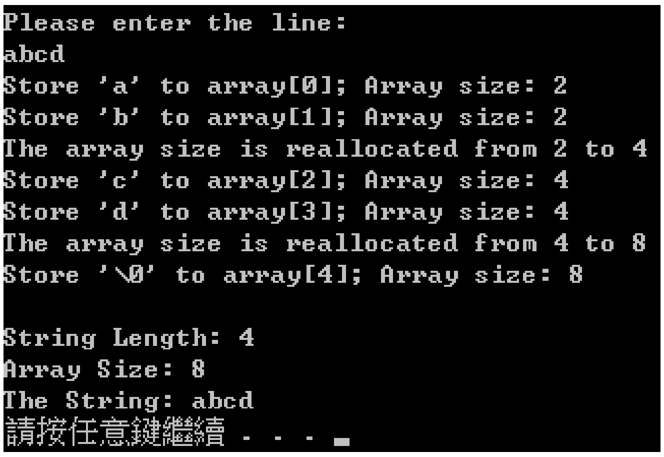
(1) Declare a pointer to char. Allocate a block of memory of an array of n char to it. Record the size of the array as n.
(2) Read the input char by char. Store them in the corresponding members of the array.
(3) If the number of stored characters reaches n, allocate a new block of 2n char for the array. Record the size of the array as 2n. Go back to (2).
(4) If the end of the input is met, add a null character (\0) to the end of the string.

You are asked to implement an unlimted string as mentioned above. The initial size is 2 char. You have to prompt the user to enter a line of characters, and then store the line (not include \n) to the string. You have to keep trace of the length of the string and the size of the array, print messages when **storing elements** and **array reallocation**, and print out **the length, the size and the string in the end**. The example output is shown below.
Note: Be sure to check whether the returned memory is NULL. If it's NULL, just print "*String length out of memory!*", and terminate the program.



```
Please enter the line:
Kobe Bryant!!!
Store 'K' to array[0]; Array size: 2
Store 'o' to array[1]; Array size: 2
The array size is reallocated from 2 to 4
Store 'b' to array[2]; Array size: 4
Store 'e' to array[3]; Array size: 4
The array size is reallocated from 4 to 8
Store ' ' to array[4]; Array size: 8
Store 'B' to array[5]; Array size: 8
Store 'r' to array[6]; Array size: 8
Store 'y' to array[7]; Array size: 8
The array size is reallocated from 8 to 16
Store 'a' to array[8]; Array size: 16
Store 'n' to array[9]; Array size: 16
Store 't' to array[10]; Array size: 16
Store '!' to array[11]; Array size: 16
Store '!' to array[12]; Array size: 16
Store '!' to array[13]; Array size: 16
Store '\0' to array[14]; Array size: 16

String Length: 14
Array Size: 16
The String: Kobe Bryant!!!
請按任意鍵繼續 . . .
```



```
Please enter the line:
abcd
Store 'a' to array[0]; Array size: 2
Store 'b' to array[1]; Array size: 2
The array size is reallocated from 2 to 4
Store 'c' to array[2]; Array size: 4
Store 'd' to array[3]; Array size: 4
The array size is reallocated from 4 to 8
Store '\0' to array[4]; Array size: 8

String Length: 4
Array Size: 8
The String: abcd
請按任意鍵繼續 . . .
```

## 2. Dynamic Integer Array

Dynamic memory allocation allows us to allocate memory at the execution time. And because the allocated memory is contiguous, we can use it as an array. However, dealing with dynamic allocation of memory is difficult and dangerous. If you don't do it right, you may crash the program or cause memory leak.
So we may want to encapsulate the action of dynamic memory allocation.
In order to do that, we build the Structure of Dynamic Integer Array and supposed functions of it.
**Your program should not crash in any situation !!**
About the Structure of Dynamic Integer Array (DynaIntArr)

```
    capacity: the maximum number of elements that we can store
    size: the actual number of elements that are stored
    data: the pointer to the dynamic allocated memory spaces
```

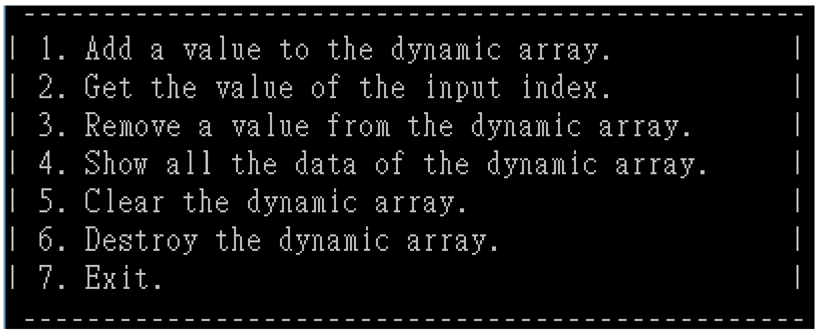
About the behaves of supposed functions

```
int addOne(struct DynaIntArr *arr, int value) {
    if capacity is not enough
        Allocate size + 1 units of int.
        Copy the original size ints into the newly allocated spaces.
    Put value at the last position.
    Increase size by one.
    return 1 if it allocates memory successfully; otherwise, return 0.
}
int * getOne(struct DynaIntArr *arr, int index) {
    return "the pointer" to the target value
    (Will return null pointer when the index is invalid)
}
int removeOne(struct DynaIntArr *arr, int index) {
    Remove the value of the index from the array
    Move forward all the values behind the index
    return 1 if remove successfully; otherwise, return 0.
    (Assuming that removeOne () is not called frequently, we do not deallocate the memory.)
}
void clearArr(struct DynaIntArr *arr){
    remove all data but do not give back the memory space
    (After call clearArr(targetArr),  getOne(targetArr, anyIndex) should always give null pointer.)
}
void destroyArr(struct DynaIntArr *arr){
    remove all data and give back the memory space
    (After call destroyArr (targetArr), getOne(targetArr, anyIndex) should always give null pointer.)
    (We should call destroyArr (targetArr) many times without crash the program)
    (After you call destroyArr, you can still call addOne to add new elements)
}
```

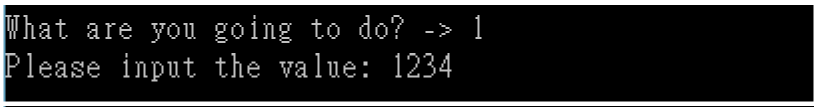
Note:

You have to follow TA's template and finish these five functions to practice how to use Dynamic memory allocation effectively and avoid Memory Leak.

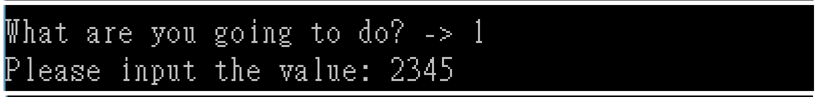
Examples:



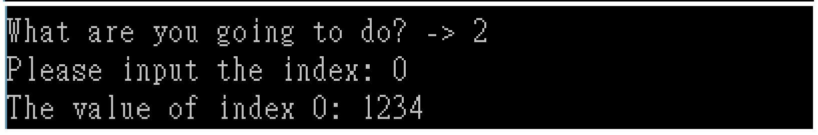
```
 ---------------------------------------------
| 1. Add a value to the dynamic array.        |
| 2. Get the value of the input index.        |
| 3. Remove a value from the dynamic array.   |
| 4. Show all the data of the dynamic array.  |
| 5. Clear the dynamic array.                 |
| 6. Destroy the dynamic array.               |
| 7. Exit.                                    |
 ---------------------------------------------
```



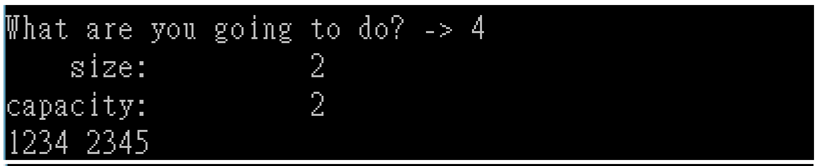
```
What are you going to do? -> 1
Please input the value: 1234
```



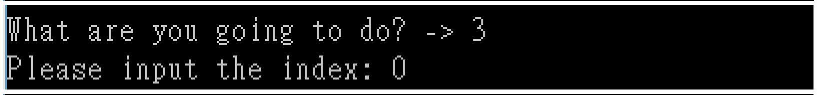
```
What are you going to do? -> 1
Please input the value: 2345
```



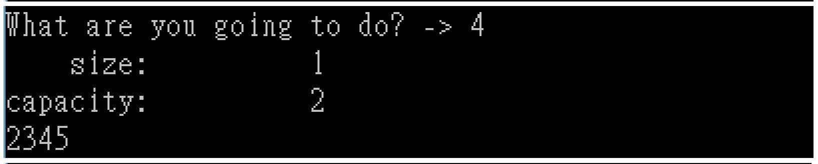
```
What are you going to do? -> 2
Please input the index: 0
The value of index 0: 1234
```



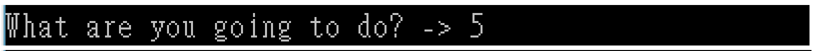
```
What are you going to do? -> 4
    size:         2
capacity:         2
1234 2345
```



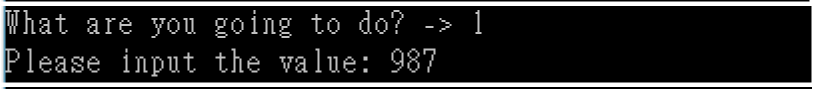
```
What are you going to do? -> 3
Please input the index: 0
```



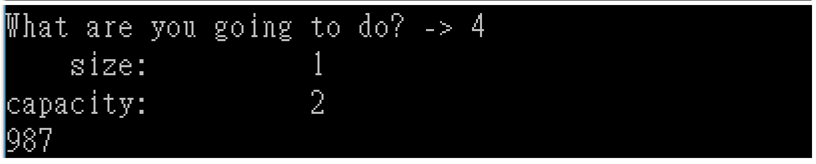
```
What are you going to do? -> 4
    size:         1
capacity:         2
2345
```



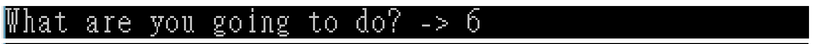
```
What are you going to do? -> 5
```



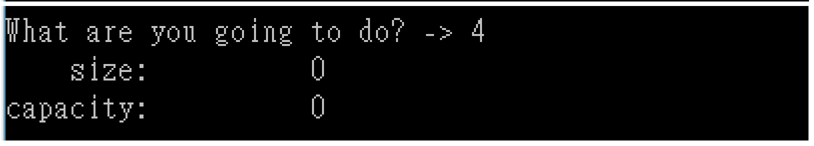
```
What are you going to do? -> 1
Please input the value: 987
```



```
What are you going to do? -> 4
    size:         1
capacity:         2
987
```



```
What are you going to do? -> 6
```



```
What are you going to do? -> 4
    size:         0
capacity:         0
```

## 3. Paragraph Compression

In this program, you are going to implement a simple paragraph compression tool. Given **a paragraph only consists of alphabetic words** (used to represent a word), and **other symbolic words** (used to separate two words) like comma, dot, space character, …, etc. **There will be no number in a paragraph except the last line**. Here are some rules to generate such a compressed paragraph and a list:

(1) Once it reads a word that never occurs, **arrange that word in the front of the list** (It could be either a structure array or a linked list. Depend on your choice), and output the original word.

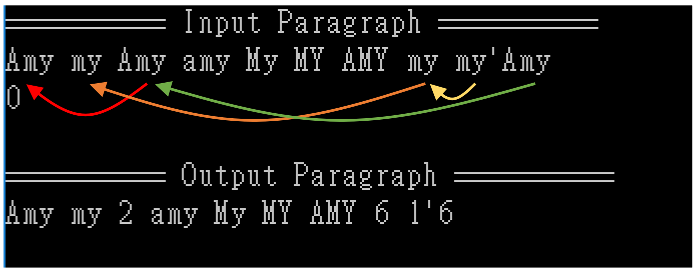
(2) If a word that has already been added into the list, then **replace and output that word with the index number** (index starts from 1) in the list, **and rearrange that word to the front of the list agai**n.

(3) Always output the symbolic in the original position.

(4) Words "Abc" and "abc" represent different words. "X-ray" represents words "X" and "ray", which are separate by symbol "-".

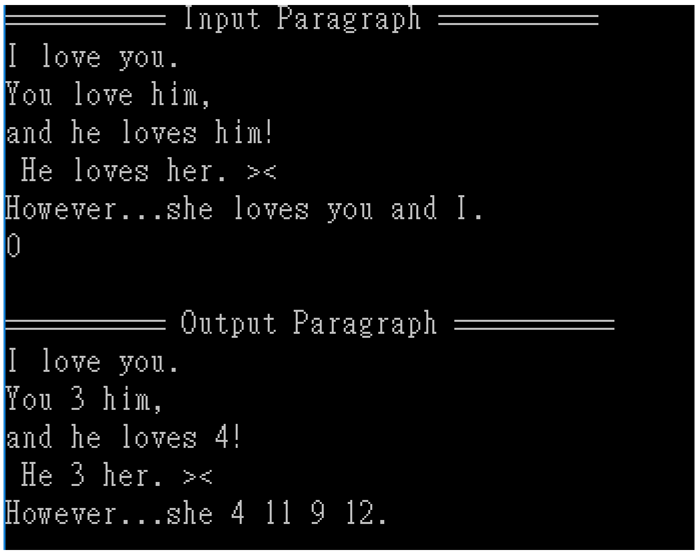
(5) The last line of the input paragraph should be "0". Then output the compressed paragraph.

(6) The maximum of input line is **10**. The maximum of length of a word is **15**. The number of characters in a line won't exceed **80**.



```
========== Input Paragraph ==========
Amy my Amy amy My MY AMY my my'Amy
0

========== Output Paragraph ==========
Amy my 2 amy My MY AMY 6 1'6
```



```
========== Input Paragraph ==========
I love you.
You love him,
and he loves him!
 He loves her. ><
However...she loves you and I.
0

========== Output Paragraph ==========
I love you.
You 3 him,
and he loves 4!
 He 3 her. ><
However...she 4 11 9 12.
```

Hint:

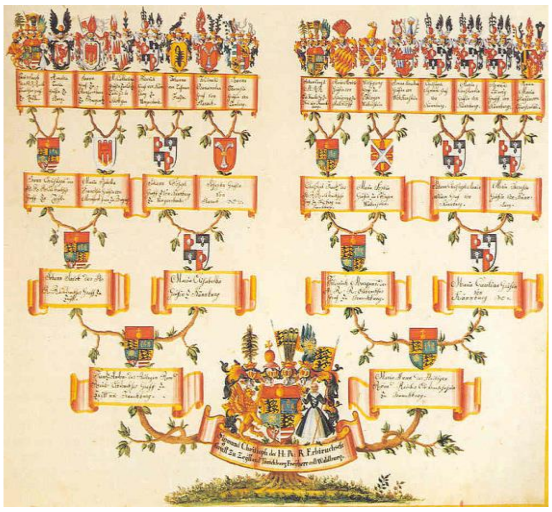
(1) **strcmp(str1, str2)** returns 0 if two strings are the same

(2) **sprintf(str, "%d", index)** could easily convert a number to a string

(3) **strcat(str1, str2)** could combine str2 to the back of str1

## 4. Family Tree:

A family tree, or pedigree chart, is a chart representing family relationships in a conventional tree structure. The more detailed family trees used in medicine and social work are known as genograms.



Genealogical data can be represented in several formats, for example as a pedigree or **ancestry chart**. Family trees are often presented with the oldest generations at the top and the newer generations at the bottom. An ancestry chart, which is a tree showing the ancestors of an individual, will more closely resemble a tree in shape, being wider at the top than the bottom. In some ancestry charts, an individual appears on the left and his or her ancestors appear to the right. A descendancy chart, which depicts all the descendants of an individual will be narrowest at the top.

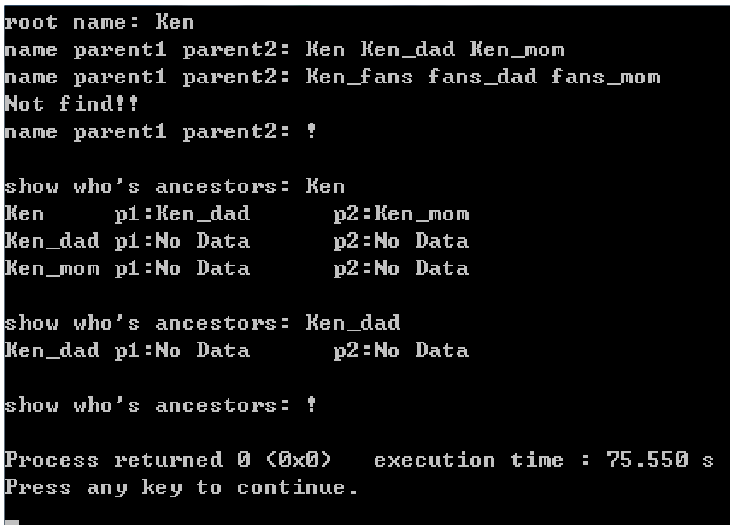
[ref by wiki: https://en.wikipedia.org/wiki/Family_tree]

Please use **link list** to save a family tree. Follow the below steps:

甲、Type root name.

乙、Keep add root’s ancestors(or parents) until user type ”!”.

丙、Show one’s family tree until user type ”!”.



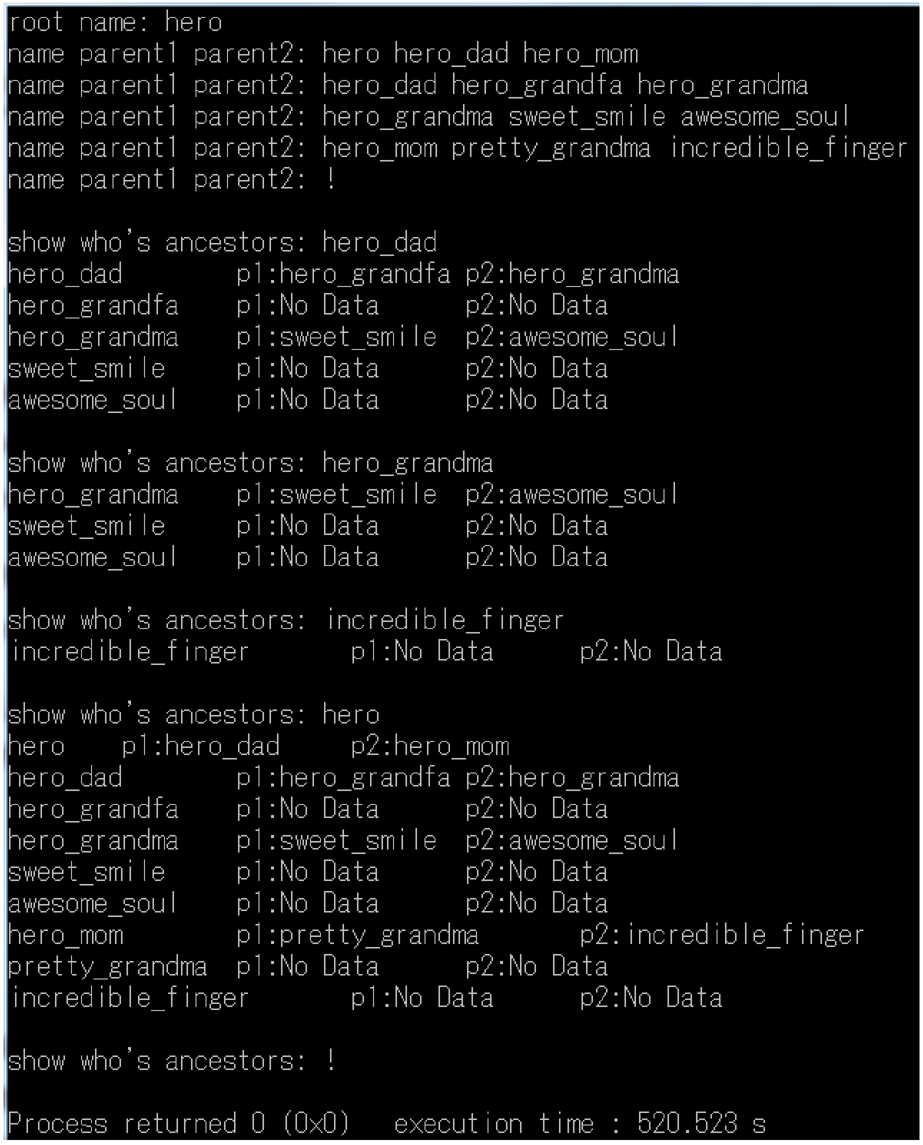
```
root name: Ken
name parent1 parent2: Ken Ken_dad Ken_mom
name parent1 parent2: Ken_fans fans_dad fans_mom
Not find!!
name parent1 parent2: !

show who's ancestors: Ken
Ken      p1:Ken_dad      p2:Ken_mom
Ken_dad p1:No Data      p2:No Data
Ken_mom p1:No Data      p2:No Data

show who's ancestors: Ken_dad
Ken_dad p1:No Data      p2:No Data

show who's ancestors: !

Process returned 0 (0x0)   execution time : 75.550 s
Press any key to continue.
```



```
root name: hero
name parent1 parent2: hero hero_dad hero_mom
name parent1 parent2: hero_dad hero_grandfa hero_grandma
name parent1 parent2: hero_grandma sweet_smile awesome_soul
name parent1 parent2: hero_mom pretty_grandma incredible_finger
name parent1 parent2: !

show who's ancestors: hero_dad
hero_dad        p1:hero_grandfa p2:hero_grandma
hero_grandfa    p1:No Data      p2:No Data
hero_grandma    p1:sweet_smile  p2:awesome_soul
sweet_smile     p1:No Data      p2:No Data
awesome_soul    p1:No Data      p2:No Data

show who's ancestors: hero_grandma
hero_grandma    p1:sweet_smile  p2:awesome_soul
sweet_smile     p1:No Data      p2:No Data
awesome_soul    p1:No Data      p2:No Data

show who's ancestors: incredible_finger
incredible_finger       p1:No Data      p2:No Data

show who's ancestors: hero
hero    p1:hero_dad     p2:hero_mom
hero_dad        p1:hero_grandfa p2:hero_grandma
hero_grandfa    p1:No Data      p2:No Data
hero_grandma    p1:sweet_smile  p2:awesome_soul
sweet_smile     p1:No Data      p2:No Data
awesome_soul    p1:No Data      p2:No Data
hero_mom        p1:pretty_grandma       p2:incredible_finger
pretty_grandma  p1:No Data      p2:No Data
incredible_finger       p1:No Data      p2:No Data

show who's ancestors: !

Process returned 0 (0x0)   execution time : 520.523 s
```