Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-1 CS112
2018/2019

C-Strings

Dr. Amin Allam

1 C-Strings

Arrays can be of any type, such as `char`. A *string* is a sequence of characters, or an array of characters. The C programming language provides additional support for handling strings. A C-String is a character array that contains the string characters followed by the *null character* `'\0'` which has the ascii code of `0`. The null character is not part of the string characters, but it serves to know where the string ends. The null character is an alternative to specifying an integer which accompanies the string to identify its length.

```
1 char a[4]={'S', 'e', 'a', 0};
2 cout << a << endl; // Prints: Sea
```

The above code is an example of the additional support of C for strings. The `cout<<` detects that `a` is a string, because it is defined as `char` array. The `cout<<` prints all characters of `a` on screen, until it finds the null character `0` and then it stops printing. The null character is not printed, but it is just used by `cout<<` to know when the array stops. Note that we must allocate an array of size at least 4, which is larger than the string length by 1, in order to allocate space to hold the null character.

The `cin>>` also supports strings. The following code will take a string from the user. The `cin>>` will take a sequence of characters entered by the user and stops when it encounters an empty space (space, tab, or end of line). The `cin>>` will add the null character to the end of the input string.

```
1 char a[10];   // This string can hold up to 9 characters
2 cin >> a;
```

In the above example, if the user entered "Desert" followed by empty space, the array will look like the following in memory (the dashes - mean garbage values):

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
'D'	'e'	's'	'e'	'r'	't'	'\0'	-	-	-

The values stored in memory are actually the ascii codes that correspond to these characters. We write the characters inside quotes only for clarity.

Note that although `cout<<` and `cin>>` support `char` array, they do not support other array types. For example, this code is wrong:

```
1 int a[4]={1, 2, 3, 0};
2 cout << a << endl;      ✖
```

C provides a shortcut for initializing strings without specifying the size in brackets. Instead of writing the null character, we can use double quotes and C will append the null character:

```
1 char a[4]={'S', 'e', 'a', 0};   ✔
2 char b[4]={'S', 'e', 'a'};      ✔  // Unused spaces are filled by zeros
3 char c[] ={'S', 'e', 'a', 0};   ✔
4 char e[] ={'S', 'e', 'a'};      ✖  // The compiler will count as 3 not 4
5 char d[] ="Sea";                ✔  // The compiler will add 0 and count 4
```

Since array `b` is initialized and its size is larger than the number of initialized characters, the non-initilaized characters are set to `0` by default, so `b[3]` will contain `0` and `b` will be equivalent to `a`. Array `e` is valid, but it is not C-String. For example, `cout<<e` will not work properly since it will not find the null character so it will not know where the string ends. Array `d` is a C-String, since the double quotes " " will direct the compiler to consider it as C-String. The compiler adds the null character to the end and sets the array size to `4`. Array `d` is a C-String, since the double quotes " "will direct the compiler to consider it as C-String. The compiler adds the null character to the end and sets the array size to `4`.

Note that there is a big difference from the printed character `'0'` (digit zero) which has ascii code of `48` and the non-printed null character `'\0'` which has ascii code of `0`.

2 sizeof() operator

`sizeof()` operator takes variable, array, or data type and returns its allocated size in memory in bytes, as follows:

```
1  char c;
2  char a[10]="Sea";
3  int  t;
4  int  b[10]={1,2,3,4,5};
5  int  r[]={1,2,3,4,5};
6  cout << sizeof(char) << endl;   // Prints 1 since a char takes one byte
7  cout << sizeof(c)    << endl;   // Prints 1 since a char takes one byte
8  cout << sizeof(a[6]) << endl;   // Prints 1 since a char takes one byte
9  cout << sizeof(int)  << endl;   // Prints 4 since an int takes 4 bytes
10 cout << sizeof(t)    << endl;   // Prints 4 since an int takes 4 bytes
11 cout << sizeof(b[2]) << endl;   // Prints 4 since an int takes 4 bytes
12 cout << sizeof(a)    << endl;   // Prints 10 since a[] allocates 10 char variables
13 cout << sizeof(b)    << endl;   // Prints 40 since b[] allocates 10 int variables
14 cout << sizeof(r)    << endl;   // Prints 20 since r[] allocates 5 int variables
15 cout << sizeof(r)/sizeof(int)   << endl;  // Prints 5 since r[] allocates 5 ints
16 cout << sizeof(r)/sizeof(r[0]) << endl;  // Prints 5 since r[] allocates 5 ints
```

3 Built-in functions

A built-in function is a function which is already implemented. We can use a built-in function if we include the appropriate header and call it using appropriate parameters from inside our code. In this section, we discuss a few built-in functions that deal with C-Strings. All functions discussed in this section need to include the `cstring` header:

```
#include <cstring>
using namespace std;
```

The `strlen()` function takes a C-String and returns the number of characters inside it. `strlen()` works by counting all characters of the parameter array and stops when it finds the null character. The null character is not counted but used only to stop counting. The following example illustrates the difference between `sizeof()` and `strlen()`:

```
char a[10]="Sea";
cout << sizeof(a) << endl;   // Prints 10 since a[] allocates 10 char variables
cout << strlen(a) << endl;   // Prints 3 since a[] contains 3 characters
```

The `strcmp()` function compares two C-Strings. It returns 0 if the two strings are equal. It returns a positive integer if the first string is lexicographically larger than the second string. Otherwise it returns a negative number. The function compares the first letters of each string, if they have different ascii codes, it returns a non-zero integer, otherwise it compares the second letters of each string. The function is case-sensitive and all comparisons are based on the ascii codes as illustrated by examples:

```
int a;
a=strcmp("sea", "sea");   // a=0 since the two strings are the same
a=strcmp("tea", "sea");   // a>0 since the ascii code of 't' is larger than of 's'
a=strcmp("sea", "tea");   // a<0 since the ascii code of 's' is smaller than of 't'
a=strcmp("sea", "t");     // a<0 since the ascii code of 's' is smaller than of 't'
a=strcmp("sea", "sun");   // a<0 since the first letter equal and 'e'<'u'
a=strcmp(""   , "sea");   // a<0 since the first string is empty and the second not
a=strcmp("s"  , "sea");   // a<0 since the first string is prefix of the second
a=strcmp("Sea", "sea");   // a<0 since the ascii code of 'S' is smaller than of 's'
a=strcmp("SEA", "sea");   // a<0 since the ascii code of 'S' is smaller than of 's'
a=strcmp("sea", "set");   // a<0 since the ascii code of 'a' is smaller than of 't'
```

The `strcpy()` function takes two C-Strings and copies the second one into the first one. The order of parameters is specified to be similar to the assignment operator:

```
char a[10]="Sea";
char b[8]="Sea";
char c[7];
strcpy(c, "Desert");
strcpy(b, c);
cout << b << " " << c << endl;   // Prints "Desert Desert"
```

The `strcat()` function takes two C-Strings and appends the second one to the first one. The order of parameters is specified to be similar to the += operator:

```
char a[20]="Sea";
char b[]="Desert";
strcpy(a, " and ");
strcpy(a, b);
cout << a << endl;   // Prints "Sea and Desert"
```

4 Examples

1. Write a function that takes three strings. After the function ends, the first string should be composed of the concatenation of the second and third strings.

```
void Concatenate(char a[], char b[], char c[])
{
   strcpy(a,b);   // Copy b[] into a[] (b[] string overwrites old a[] string )
   strcat(a,c);   // Append c[] to a[]
}
```

   Another solution:

```
void Concatenate(char a[], char b[], char c[])
{
   a[0]=0;        // Let a[] be an empty string
   strcat(a,b);   // Append b[] to a[]
   strcat(a,c);   // Append c[] to a[]
}
```

2. Write a function that takes a string and returns true if the string is palindrome. A palindrome string is read the same from left to right and from right to left.

```
bool IsPalindrome(char a[])
{
   int i, n=strlen(a);
   for(i=0; i<n/2; i++)
   {
      if(a[i]!=a[n-1-i])
         return false;
   }
   return true;
}
```

3. Write a function that takes a string and converts all existing uppercase letters into lowercase.

```
1  void ToLower(char a[])
2  {
3     int i, n=strlen(a);
4     for(i=0; i<n; i++)
5     {
6        if(a[i]>='A' && a[i]<='Z')    // if a[i] is uppercase letter
7        {
8           a[i]-='A';    // Now a[i] is 0 for 'A', 1 for 'B' and so on
9           a[i]+='a';    // Now a[i] is 'a' for 0, 'b' for 1 and so on
10       }
11    }
12 }
```

Another solution:

```
1  void ToLower(char a[])
2  {
3     for(int i=0; a[i]!=0; i++)  // Continue as we do not reach the null char
4        if(a[i]>='A' && a[i]<='Z')  // If a[i] is uppercase letter
5           a[i]=a[i]-'A'+'a';  // Convert a[i] into lowercase letter
6  }
```

Another solution:

```
1  void ToLower(char a[])
2  {
3     for(int i=0; a[i]; i++)  // Continue as we do not reach the null character
4        if(a[i]>='A' && a[i]<='Z')  // If a[i] is uppercase letter
5           a[i]=a[i]-'A'+'a';  // Convert a[i] into lowercase letter
6  }
```

The last solution replaces `a[i]!=0` by `a[i]`. When a value (such as `a[i]`) is put in place of a boolean expression, it is treated as `false` only if it equals `0`, otherwise it is treated as `true` if it is non-zero.