# C-Strings

C-strings are a special type of array. They include what is called the **null zero**, which looks like '\0' at the end of the array. The **null zero** is used to indicate the end of a C-string. This is not shown in the output of the C-string, but it exists.

'\0'

There are two forms of initialization:

```
char charArray[] = "Hello";
char charArray[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

As you can see, the first initialization does not include the **null zero**. It is included for you. If, however, you initialize using the second method, you need to manually put it in yourself. If you omit the **null zero**, then your C-string will not behave correctly. Many functions depend on the **null zero**. The first method is the preferred method to initializing an array.

Using the **sizeof()** function will prove to you that **char charArray** is one size bigger than your input, such as **cout << sizeof(charArray);** For example, **char myArray[] = "Hello";** will produce a size of 6 to include the **null zero**. If you manually put the size of the array in the square brackets, that size must include the **null zero**. Example is if you put **50**, then only the first 49 elements are allowed to be used.

Once you initialize a **char** array, you cannot change it as you could, say, an **int**. That is, you cannot change **charArray** by setting it equal to another string, such as **charArray = "World";** You are able to manipulate each indexed element of a C-string just as you would an array. For example, if you wanted to make the **'H'** lowercase, you could do so by **charArray[0] = 'h'**. Be sure not to override the **null zero** at the end.

The convenient thing about **char arrays** is to display the array, looping or individual elements need not be used. You can simply cout the array variable. An example is **cout << myArray;** This will display **Hello** directly.

## Manipulating C-Strings

It is very important to remember that when manipulating C-Strings you do not go past the allotted size and you do not delete the **null zero**. If you go past the allotted size, when running your program you may get a segmentation fault, which means your program tried to access memory it wasn't allowed to access. The following functions actively manipulate strings, but it is up to you to ensure it is done properly and safely. The library to be used (preprocessor directive) is **#include <cstring>**.

## Compare

**Compare**
**Concatenate**
**Copy**
**Length**

There are two potential functions to compare two strings. The difference is that the second one adds a third parameter and indicates how many elements you want to compare. They are:

```
strcmp(string1, string2)
strncmp(string1, string2, HowManyToCompare)
```

**Lexicographic order** is used when comparing strings. This means that the strings are compared using the values indicated in the **ASCII** table. For example, the letter A is equivalent to the number 65 and the letter B is equivalent to the number 66. Therefore, letter B is greater than letter A. With this in mind, comparing two strings starts with comparing the first elements together, then the second elements, and so forth. The strings need not be the same size to compare. The return value for these functions in an integer, and the resulting integer value is what you might not expect.

For equal strings the function returns a 0. Example:

```
cout << strcmp(“Hey”, “Hey”);
```

If **string1** is less than **string2**, then the function returns a number less than one. For example, comparing **“Bomb”** and **“Book”** would yield a negative number, because when it compares the third elements, the letter m is less than the letter o in the **ASCII** table. Example:

```
char string1[] = “Bomb”; //m = 109 in ASCII table
char string2[] = “Book”; //o = 111 in ASCII table
cout << strcmp(string1, string2);  //Outputs -2 as that’s the diff
```

If **string1** is greater than **string2**, then the function returns a number greater than one. For example, comparing **“Pensils”** and **“Pensil”** would yield a positive number, because there is one additional letter in **string1** than there is in **string2** making **string1** larger. Here is an example where the strings are the same length, but **string1** has a character that is represented as a higher number in the **ASCII** table:

```
Cout << strcmp(“hi”, “Hi”); //h = 104 H = 72 +32 difference
```

If you want to compare only the first 3 elements of the strings, use **strncmp** with 3 in plate of **HowManyToCompare**. Be sure not to go over the appropriate size of the C-String. **strncmp** is safer to use as the limit is rightfully chosen (not present in all C++ versions). Example:

```
cout << strncmp(“Buffalo”, “Burn”, 2) //Outputs 0 as first 2 are equal
```

Keep in mind capital letters are smaller in value than lowercase letters.

## Concatenate

There are two potential functions to concatenate two strings together. Concatenate means to append to a string (to join two strings together). Just like comparing strings, the difference is that one has an argument for how many elements to, in this case, concatenate. They are:

```
strcat(appendedString, stringToAppend)
strncat(appendedString, stringToAppend, HowManyElementsToAppend)
```

A couple things to keep in mind is that you have to make sure when concatenating strings, the **appendedString's** size is large enough to accommodate the **stringToAppend**. Also, **strncat** is not present in all C++ versions. Here is an example utilizing both versions:

```
char appendedString[50] = "Hello"; //Size 50 to be safe
char stringToAppend[] = " World"; //Space so it isn't HelloWorld
cout << strcat(appendedString, stringToAppend); //Outputs Hello World
cout << strncat(appendedString, stringToAppend, 3); //Outputs Hello Wo
```

For brevity, both functions are displayed using the same variables at the same time. If you were to run this program as it is, the first function will concatenate **Hello** and **World** to make **Hello World**. Since **appendedString** is now **Hello World**, then appending to it again the first three elements of **stringToAppend** would yield **Hello World Wo**.

## Copy

There are two potential functions to copy over one string to another. As the previous two functions, the difference between the two is that one has an argument for how many elements to copy over. They are:

```
strcpy(overriddenString, stringToCopyOver)
strncpy(overriddenString, stringToCopyOver, HowManyElementsToCopy)
```

Copying over a string means that starting from the first element, index zero, **overriddenString** gets its first element replaced with **stringToCopyOver's** first element. This occurs element-by-element. If you only want so many elements to be copied over, then use **strncpy**. This function is not present in all C++ versions. You also need to make sure that you have enough space in **overriddenString** to accommodate for the copy. This is just like concatenate. Here is an example:

```
char overriddenString[20] = "January";
char stringToCopyOver[] = "September";
cout << strncpy(overriddenString, StringToCopyOver, 4);
```

The output to this is **Septary** since only the first 4 were copied over.

## Length

The length function does what the name implies, returns the length of the C-string. However, keep in mind this function does not include the **null zero**. The length is only the literal length of the C-string. The function is as follows:

```
strlen(stringToBeMeasured)
```

An example of this function is:

```
char array[] = “Programming”;
cout << strlen(array);
```

This will display 11. Keep in mind the difference between this function and **sizeof()** is that **sizeof()** includes the **null zero**. Since **strlen** returns an **int**, you are able to store the result in an integer variable, such as:

```
int length = strlen(array);
```

## Conclusion

One last note is that the functions for **comparing**, **concatenation**, and **copying** is that you do not have to put those functions in a cout statement. They may be on a line of their own. If you would like to do that and display what the function returns, cout the first argument to each of those functions. An example is:

```
char string1[40] = “Concat”;
char string2[] = “enation”;
strcat(string1, string2);
cout << string1 << endl; //Outputs Concatenation
```

Since **string1** is the one being appended with **string2**, **string1** has been officially altered and will remain that way just like changing an integer variable to another number. You can alter it as much as you like as you saw on the literal implementation of **concatenate** (what was described below the example). Again, it is very important that you do not overwrite the **null zero** or go past the allotted size.

Keep in mind that using the functions in a cout statement formally alters the string in the first argument, just like the method above.

C-strings are very meticulous to work with. You always have to keep in mind the size of the c-strings as well as the **null zero**. The vast majority of your programming will deal with the **string class**, which is a lot safer to work with. There is no dealing with **null zero** nor do you have to worry about the size of each string. More details on the next page.