Chapter 8: C Strings

C++ was designed to maximize backward compatibility with the C programming language, and consequently absorbed C's representation of character strings. Unlike the C++ `string` type, C strings are very difficult to work with. *Very* difficult. In fact, they are so difficult to work with that C++ programmers invented their own `string` type so that they can avoid directly using C strings.

While C strings are significantly more challenging than C++ `string`s and far more dangerous, no C++ text would be truly complete without a discussion of C strings. This chapter enters the perilous waters of C strings and their associated helper functions.

What is a C String?

In C++, `string` is a class that expresses many common operations with simple operator syntax. You can make deep copies with the = operator, concatenate with +, and check for equality with ==. However, nearly every desirable feature of the C++ `string`, such as encapsulated memory management and logical operator syntax, uses language features specific to C++. C strings, on the other hand, are simply `char *` character pointers that store the starting addresses of specially-formatted character sequences. In other words, C++ `string`s exemplify abstraction and implementation hiding, while C strings among the lowest-level constructs you will routinely encounter in C++.

Because C strings operate at a low level, they present numerous programming challenges. When working with C strings you must manually allocate, resize, and delete string storage space. Also, because C strings are represented as blocks of memory, the syntax for accessing character ranges requires a sophisticated understanding of pointer manipulation. Compounding the problem, C string manipulation functions are cryptic and complicated.

However, because C strings are so low-level, they have several benefits over the C++ `string`. Since C strings are contiguous regions of memory, library code for manipulating C strings can be written in lighting-fast assembly code that can outperform even the most tightly-written C or C++ loops. Indeed, C strings will almost consistently outperform C++ `string`s.

Why study C strings?

In the early days of C++, the standard `string` type did not exist and C strings were the norm. Over time, C strings have been declining in usage. Most modern C++ programs either use the standard `string` type or a similarly-designed custom string type. So why should we dedicate any time to studying C strings? There are several reasons:

- **Legacy code**. C++ code often needs to interoperate with pure C code or older C++ code that predates the `string` type. In these cases you are likely to bump into C strings or classes layered on top of them, and without an understanding of C strings you are likely to get confused or stuck.

- **Edge cases**. It is legal C++ code to add an integer to a string literal, but doing so almost always results in a crash. Unless you understand how C strings and pointer arithmetic work, the root cause of this behavior will remain a mystery.

- **Library implementation**. You may be called upon to implement a string class that supports functionality beyond that of the standard `string`. Knowing how to manipulate C strings can greatly simplify this task.

We will encounter each of these cases in the remainder of this course.

Memory representations of C strings

A C string is represented in memory as a consecutive sequence of characters that ends with a "terminating null," a special character with numeric value 0. Just as you can use the escape sequences `'\n'` for a newline and `'\t'` for a horizontal tab, you can use the `'\0'` (slash zero) escape sequence to represent a terminating null. Fortunately, whenever you write a string literal in C or C++, the compiler will automatically append a terminating null for you, so only rarely will you need to explicitly write the null character. For example, the string literal "Pirate" is actually *seven* characters long in C++ – six for "Pirate" plus one extra for the terminating null. Most library functions automatically insert terminating nulls for you, but you should always be sure to read the function documentation to verify this. Without a terminating null, C and C++ won't know when to stop reading characters, either returning garbage strings or causing crashes.[*]

The string "Pirate" might look something like this in memory:

`Address 1000`	**`P`**
`1001`	**`i`**
`1002`	**`r`**
`1003`	**`a`**
`1004`	**`t`**
`1005`	**`e`**
`1006`	**`\0`**

Note that while the end of the string is delineated by the terminating null, there is no indication here of where the string begins. Looking solely at the memory, it's unclear whether the string is "Pirate," "irate," "rate," or "ate." The only reason we "know" that the string is "Pirate" is because we know that its starting address is 1000.

This has important implications for working with C strings. Given a starting memory address, it is possible to entirely determine a string by reading characters until we reach a terminating null. In fact, provided the memory is laid out as shown above, it's possible to reference a string by means of a single `char *` variable that holds the starting address of the character block, in this case 1000. If you encounter a function that accepts a parameter of type `char*`, chances are that it accepts a C string rather than a pointer to a single character.

Memory Segments

Before we begin working with C strings, we need to quickly cover memory segments. When you run a C++ program, the operating system usually allocates memory for your program in "segments," special regions dedicated to different tasks. You are most familiar with the stack segment, where local variables are stored and preserved between function calls. Also, there is a heap segment that stores memory dynamically allocated with the `new` and `delete` operators. There are two more segments, the code (or text) segment and the data segment, of which we must speak briefly.

When you write C or C++ code like the code shown below:

[*] Two C strings walk into a bar. One C string says "Hello, my name is John#30g4nvu342t7643t5k...", so the second C string turns to the bartender and says "Please excuse my friend... he's not null-terminated."

```
int main()
{
	char* myCString = "This is a C string!";
	return 0;
}
```

The text “This is a C string!” must be stored somewhere in memory when your program begins running. On many systems, this text is stored in either the read-only code segment or in a read-only portion of the data segment. This enables the compiler to reduce the overall memory footprint of the program, since if multiple parts of the code each use the same string literal they can all point to the same string. But this optimization is not without its costs. If you modify the contents of a read-only segment, you will crash your program with a *segmentation fault* (sometimes also called an *access violation* or “seg fault”).

Because your program cannot write to read-only memory segments, if you plan on manipulating the contents of a C string, you will need to first create a copy of that string somewhere where your program has write permission, usually in the heap. Thus, for the remainder of this chapter, any code that modifies strings will assume that the string resides either in the heap or on the stack (usually the former). Forgetting to duplicate the string and store its contents in a new buffer can cause many a debugging nightmare, so make sure that you have writing access before you try to manipulate C strings.

Allocating Space for Strings

Before you can manipulate a C string, you need to first allocate memory to store it. While traditionally this is done using older C library functions (briefly described in the “More to Explore” section), because we are working in C++, we will instead use the `new[]` and `delete[]` operators for memory management.

When allocating space for C strings, you must make sure to allocate enough space to store the entire string, *including the terminating null character.* If you do not allocate enough space, when you try to copy the string from its current location to your new buffer, you will write past the end of the buffer into memory you do not necessarily own. This is known as a *buffer overrun* and can crash your program. Worse, it can allow crafty hackers to hijack your program and take control of your operating system. In short, always ensure that when allocating space to hold a C string, you have enough space to hold the null terminator.

The best way to allocate space for a string is to make a new buffer with size equal to the length of the string you will be storing in the buffer. To get the length of a C string, you can use the handy `strlen` function, declared in the header file `<cstring>`.* `strlen` returns the length of a string, *not* including the terminating null character. For example:

```
cout << strlen("String!") << endl; // Value is 7

char* myStr = "01234";
cout << strlen(myStr) << endl; // Value is 5
```

Thus, if you want to make a copy of a string, to allocate enough space you can use the following code:

```
/* Assume char *text points to a C string. */
char* myCopy = new char[strlen(text) + 1]; // Remember +1 for null
```

As always, remember to deallocate any memory you allocate with `new[]` with `delete[]`.

* `<cstring>` is the Standard C++ header file for the C string library. For programs written in pure C, you'll need to instead include the header file `<string.h>`.

Basic String Operations

When working with C++ `string`s, you can make copies of a `string` using the = operator, as shown here:

```
string myString = "C++ strings are easy!";
string myOtherString = myString; // Copy myString
```

This is not the case for C strings, however. For example, consider the following code snippet:

```
char* myString = "C strings are hard!";
char* myOtherString = myString;
```

Here, the second line is a pointer assignment, *not* a string copy. As a result, when the second line finishes executing, `myString` and `myOtherString` both point to the same memory region, so changes to one string will affect the other string. This can be tricky to debug, since if you write the following code:

```
cout << myString << endl;
cout << myOtherString << endl;
```

You will get back two copies of the string `C strings are hard!`, which can trick you into thinking that you actually have made a deep copy of the string. To make a full copy of a C string, you can use the `strcpy` function, as shown below:

```
/* Assume char* source is initialized to a C string. */
char* destination = new char[strlen(source) + 1];
strcpy(destination, source);
```

Here, the line `strcpy(destination, source)` copies the data from `source` into `destination`. As with most C string operations, you must manually ensure that there is enough space in `destination` to hold a copy of `source`. Otherwise, `strcpy` will copy the data from `source` past the end of the buffer, which will wreak havoc on your program.

You might be wondering why the `strcpy` function is necessary for C strings. After all, couldn't you just write the following code:

```
char* destination = new char[strlen(source) + 1];
*destination = *source; // Problem: Legal but incorrect
```

Intuitively, it seems like this should copy the string pointed at by `source` into the string pointed at by `destination`. Unfortunately, though, this is not the case. This code only copies the first character from `source` to `destination`, since `*source` is a single character, not the entire C string. The fact that `char*`s are used to point to C strings does not mean that C++ interprets them that way. As far as C++ is concerned, a `char*` is a pointer to a single character, and you will have to explicitly treat it like a C string using the library functions to make C++ think otherwise.

Another common string operation is concatenation. To append one C string onto the end of another, use the `strcat` function. Unlike in C++, when you concatenate two C strings, you must manually ensure there is enough allocated space to hold both strings. Here is some code to concatenate two C strings. Note the amount of space reserved by the `new[]` call only allocates space for one terminating null.

```
/* Assume char* firstPart, * secondPart are initialized C strings. */
char* result = new char[strlen(firstPart) + strlen(secondPart) + 1];
strcpy(result, firstPart);  // Copy the first part.
strcat(result, secondPart); // Append the second part.
```

Because of C++'s array-pointer interchangability, we can access individual characters using array syntax. For example:

```
/* Assume myCString is initialized to "C String Fun!"
 * This code might crash if myCString points to memory in a read-only
 * segment, so we'll assume you copied it by following the above steps.
 */
cout << myCString << endl;    // Output: C String Fun!
cout << myCString[0] << endl; // Output: C

myCString[10] = 'a';
cout << myCString << endl;    // Output: C String Fan!
```

Comparing Strings

When dealing with C strings, you *cannot* use the built-in relational operators (<, ==, etc.) to check for equality. Consider the following code snippet:

```
char* strOne = /* ... */;
char* strTwo = /* ... */;

if (strOne == strTwo) // Problem: Legal but incorrect
    cout << "Equal" << endl;
```

Intuitively, this says "if `strOne` is equal to `strTwo`, print out that the strings are equal." However, think about what the code literally says to do. `strOne` and `strTwo` are pointers, and although they point to C strings, the meaning of the == operator as applied to pointers is "do these pointers point to the same location?" rather than "do these pointers point to objects with the same value?" Given this, your next intuition might be to try something like this:

```
char* strOne = /* ... */;
char* strTwo = /* ... */;

if (*strOne == *strTwo) // Problem: Legal but incorrect
    cout << "Equal" << endl;
```

That is, we dereference the pointers and check whether the elements being pointed at are identical. This is perfectly legal, but will not tell us what we want to know. The problem in this case is, again, that dereferencing a C string with the `*` operator only yields the first character of that C string, not the entire sequence of characters. Consequently, this code says "if the two strings start with the same character, print that they are equal."

To properly compare C strings, we will need to rely on a library function called `strcmp`. `strcmp` takes in two parameters, then returns an integer that is

- Zero if the two strings are equal.
- Negative if the first string lexicographically precedes the second string.
- Positive if the first string lexicographically follows the second string.

Consequently, you can check if two strings are equal as follows:

```
char* strOne = /* ... */;
char* strTwo = /* ... */;

if (strcmp(strOne, strTwo) == 0) // Legal and correct
    cout << "Equal" << endl;
```

That `strcmp` returns zero if the two strings are equal is a common source of programming errors. For example, consider the following code:

```
char* one = "This is a string!";
char* two = "This is an entirely different string!";

if(strcmp(one, two)) // Problem: Legal but incorrect
    cout << "one and two are equal!" << endl;
else
    cout << "one and two are not equal!" << endl;
```

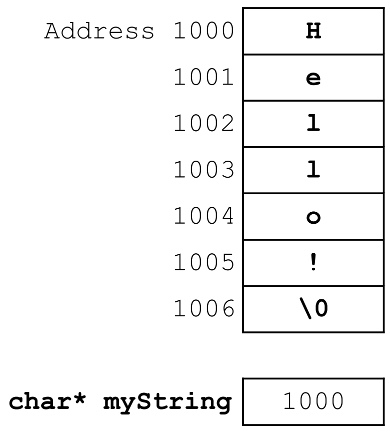
Here, we use the line `if(strcmp(one, two))` to check if `one` and `two` are equal. However, this check is *completely wrong*. In C++, any nonzero value is treated as "true" inside an `if` statement and any zero value is treated as "false." However, `strcmp` returns 0 if the two strings are equal and a nonzero value otherwise, meaning the statement `if(strcmp(one, two))` will be true if the two strings are *different* and false if they're equivalent. When working with `strcmp`, make sure that you don't accidentally make this mistake.

Pointer Arithmetic

Because C strings are low-level constructs, string functions assume a familiarity with *pointer arithmetic* – the manipulation of pointers via arithmetic operators. This next section is tricky, but is necessary to be able to fully understand how to work with C strings. Fortunately, if you think back to our discussion of STL iterators, this material should be considerably less intimidating.

In C and C++, pointers are implemented as integral data types that store memory addresses of the values they point to. Thus, it is possible to change where a pointer points by adding and subtracting values from it.

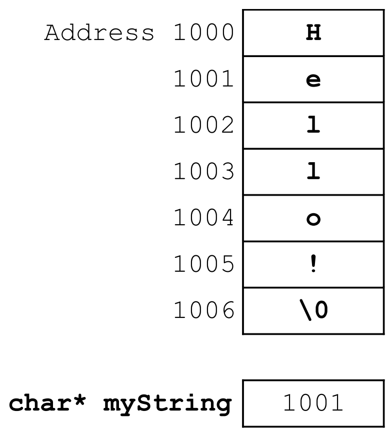
Let's begin with an example using C strings. Suppose you have the string "Hello!" and a pointer to it laid out in memory as shown below:



Currently, because `myString` stores memory address 1000, it points to the string "Hello!" What happens if we write a line of code like the one shown below?

```
myString = myString + 1;
```

In C and C++, adding one to a pointer returns a new pointer that points to the item one past the current pointer's location. In our current example, this is memory address 1001, the start of the string "ello!" Here is a drawing of the state of memory after performing the pointer arithmetic:



In general, adding `n` to a pointer returns a pointer that points `n` items further than the original pointer. Thus, given the above state of memory, if we write

```
    myString++;
```

we increment `myString` to point to memory location 1002, the string "llo!" Similarly, if afterwards we were to subtract two from `myString` by writing

```
    myString -= 2;
```

`myString` would once again contain the value 1000 and would point to the string "Hello!"

Be careful when incrementing string pointers – it is easy to increment them beyond the ends of the buffers they point to. What if we were to write the following?

```
    myString += 1000;
```

The string "Hello!" is less than 1000 characters long, and so this operation will launch the pointer far past the end of the string and into random memory. Trying to read or write from this pointer would therefore have undefined behavior and would probably result in a crash.

Let us consider one final type of pointer arithmetic, subtracting one pointer from another. Suppose we have the following C or C++ code:

```
    char* ptr1 = "This is my string!";
    char* ptr2 = ptr1 + 4;

    cout << ptr2 - ptr1 << endl;
```

What will the output be? Logically, we'd expect that since we set the value of `ptr2` to be four greater than `ptr1`, the result of the subtraction would be four. In general, subtracting two pointers yields the number of elements between them. Another way to interpret the result of pointer subtraction is as an array index. Assuming that `ptr1` points to the beginning of a C string and that `ptr2` points to an element somewhere in that string, `ptr2 - ptr1` will return the numeric index of `ptr2` in the string. This latter interpretation will be important in the upcoming section.

When working with pointer arithmetic, you will sometimes encounter the special type `ptrdiff_t`. Just as the `size_t` type represents a value encoding a number of elements, the `ptrdiff_t` type represents a value encoding a distance between two points. Unlike `size_t`, `ptrdiff_t` can be positive or negative. In most cases it is safe to use the standard `int` type instead of `ptrdiff_t`, but for correctness' sake in this text we will use `ptrdiff_t` where appropriate.

More String Functions

Armed with a understanding of pointer arithmetic, we can consider some more powerful string manipulation functions. Let us first consider the `strstr` function, which returns a pointer to the first occurrence of a given substring inside the specified string. If the substring isn't found, `strstr` returns `NULL` to signify an error.

`strstr` is demonstrated here:

```
char* myString = "C strings are difficult.";
char* found = strstr(myString, "if");

if(found == NULL)
    cout << "Substring not found." << endl;
else
    cout << "Substring occurs at index " << found - myString << endl;
```

You can also use the `strchr` function in a similar way to determine the first instance of a given character in a string. For example:

```
char* myString = "C strings are difficult.";
char* found = strchr(myString, 'i');

if(found == NULL)
    cout << "Character not found." << endl;
else
    cout << " Character occurs at index " << found - myString << endl;
```

One of the more useful string functions is the `strncpy` function, which copies a specified number of characters from the source string to the destination. However, `strncpy` is perhaps one of the most complicated library functions ever introduced.* Unlike the functions we've seen until this point, `strncpy` is not guaranteed to append a terminating null to a string. When you call `strncpy`, you specify a destination string, a source string, and a character count. If the end of the source string is reached before the specified number of characters have been copied, then `strncpy` will fill the remainder of the buffer with null characters. Otherwise, you must manually append a terminating null.

Although `strncpy` is complicated, it can be quite useful. For example, the following code demonstrates how to use `strncpy` in conjunction with pointer arithmetic to extract a substring from a source string:

```
char* GetSubstring(char* str, size_t start, size_t length)
{
    char* result = new char[length + 1]; // Include space for \0

    /* Copy length characters, starting at position start, from str into
     * result.
     */
    strncpy(result, str + start, length);

    result[length] = '\0'; // Manually append terminating null.
    return result;
}
```

This code is fairly dense, so let's walk through it in detail. The first line of code,

```
char* result = new char[length + 1];
```

* The CS department's Nick Parlante calls `strncpy` the "Worst API design ever."

allocates space to hold the substring that we will be creating. Never forget that you are responsible for allocating and deallocating C string storage space!

The next line is this somewhat cryptic call to `strncpy`:

```
strncpy(result, str + start, length);
```

`strncpy` takes three parameters. The first parameter indicates where the copy should be stored, and in our case this will be the `result` buffer we just allocated. The next parameter indicates which string to copy from. Here, we are interested in copying characters out of the string `str` beginning at position `start`. By passing as a parameter `str + start`, we mean to copy characters from `str` starting at position `start`. The final parameter specifies the number of characters to copy, which in this case is `length`.

The final line of this function is as follows:

```
result[length] = '\0';
```

Recall that `strncpy` has somewhat odd behavior with terminating nulls. In particular, if `strncpy` hits the end of the source string before the proper number of characters are read, then it will append a null terminator. Otherwise, it does not. By explicitly setting the last character of the result buffer to a null terminator, we ensure that a null will always appear at the end of the string.

C String Manipulation Reference

The following table summarizes some of the more useful C string functions. As usual, we have not covered the `const` keyword yet, but it's safe to ignore it for now.

`size_t strlen (const char* str)`	`size_t length = strlen("String!");` Returns the length of the C string `str`, excluding the terminating null character. This function is useful for determining how much space is required to hold a copy of a string.
`char* strcpy (char* dest, const char* src)`	`strcpy(myBuffer, "C strings rule!");` Copies the contents of the C string `str` into the buffer pointed to by `dest`. `strcpy` will not perform any bounds checking, so you must make sure that the destination buffer has enough space to hold the source string. `strcpy` returns `dest`.
`char* strcat (char* dest, const char* src)`	`strcat(myString, " plus more chars.");` Appends the C string specified by `src` to the C string `dest`. Like `strcpy`, `strcat` will not bounds-check, so make sure you have enough room for both strings. `strcat` returns `dest`.
`int strcmp(const char* one, const char* two)`	`if(strcmp(myStr1, myStr2) == 0) // equal` Compares two strings lexicographically and returns an integer representing how the strings relate to one another. If `one` precedes `two` alphabetically, `strcmp` returns a negative number. If the two are equal, `strcmp` returns zero. Otherwise, it returns a positive number.
`int strncmp(const char* one, const char* two, size_t numChars)`	`if(strncmp(myStr1, myStr2, 4) == 0) // First 4 chars equal` Identical to `strcmp`, except that `strncmp` accepts a third parameter indicating the maximum number of characters to compare.

`const char* strstr(const char* src, const char* key)`
 	
`char* strstr(char* src, const char* key)`
	`if(strstr(myStr, "iffy") != NULL) // found`
	
	Searches for the substring `key` in the string `source` and returns a pointer to the first instance of the substring. If `key` is not found, `strstr` returns `NULL`.

`char* strchr(char* src, int key)`
 	
`const char* strchr(const char* src, int key)`
	`if(strchr(myStr, 'a') != NULL) // found`
	
	Searches for the character `key` in the string `source` and returns a pointer to the first instance of the character. If `key` is not found, `strchr` returns `NULL`. Despite the fact that `key` is of type `int`, `key` will be treated as a `char` inside of `strchr`.

`char* strrchr(char* src, int key)`
 	
`const char* strrchr(const char* src, int key)`
	`if(strrchr(myStr, 'a') != NULL) // found`
	
	Searches for the character `key` in the `source` and returns a pointer to the *last* instance of the character. If `key` is not found, `strchr` returns `NULL`.

`char* strncpy (char* dest, const char* src, size_t count)`	`strncpy(myBuffer, "Theta", 3);` Copies up to `count` characters from the string `src` into the buffer pointed to by `dest`. If the end of `src` is reached before `count` characters are written, `strncpy` appends null characters to `dest` until `count` characters have been written. Otherwise, `strncpy` does not append a terminating null. `strncpy` returns `dest`.
`char* strncat (char* dest, const char* src, size_t count)`	`strncat(myBuffer, "Theta", 3); // Appends "The" to myBuffer` Appends up to `count` characters from the string `src` to the buffer pointed to by `dest`. Unlike `strncpy`, `strncat` will always append a terminating null to the string.
`size_t strcspn(const char* source, const char* chars)`	`size_t firstInt = strcspn(myStr, "0123456789");` Returns the index of the first character in `source` that matches any of the characters specified in the `chars` string. If the entire string is made of characters not specified in `chars`, `strcspn` returns the length of the string. This function is similar to the `find_first_of` function of the C++ `string`.

`char* strpbrk(char* source, const char* chars)`
 	
`const char* strpbrk(const char* source, const char* chars)`
	`if(strpbrk(myStr, "0123456789") == NULL) // No ints found`
	
	Returns a pointer to the first character in the source string that is contained in the second string. If no matches are found, `strpbrk` returns `NULL`. This function is functionally quite similar to `strcspn`.

`size_t strspn (const char* source, const char* chars);`	`size_t numInts = strspn(myStr, "0123456789");` Returns the index of the first character in `source` that is *not* one of the characters specified in the `chars` string. If the entire string is made of characters specified in `chars`, `strspn` returns the length of the string. This function is similar to the `find_first_not_of` function of the C++ `string`.

More to Explore

While this chapter has tried to demystify the beast that is the C string, there are several important topics we did not touch on. If you're interested in learning more about C strings, consider looking into the following topics:

1. **Command-line parameters:** Have you ever wondered why `main` returns a value? It's because it's possible to pass parameters to the `main` function by invoking your program at the command line. To write a `main` function that accepts parameters, change its declaration from `int main()` to

   ```
   int main(int argc, char* argv[])
   ```

   Here, `argc` is the number of parameters passed to `main` (the number of C strings in the array `argv`), and `argv` is an array of C strings containing the parameters. This is especially useful for those of you interested in writing command-line utilities. You will most certainly see this version of `main` if you continue writing more serious C++ code.

2. **`malloc`, `realloc`, and `free`**: These three functions are older C memory management functions that allocate, deallocate, and resize blocks of memory. These functions can be unsafe when mixed with C++ code (see Appendix 0: Moving from C to C++), but are nonetheless frequently used. Consider reading up on these functions if you're interested in programming in pure C.

3. **`sprintf` and `sscanf`**: The C++ `stringstream` class allows you to easily read and write formatted data to and from C++ `string`s. The `sprintf` and `sscanf` functions let you perform similar functions on C strings.

4. **C memory manipulation routines:** The C header file `<cstring>` contains a set of functions that let you move, fill, and copy blocks of memory. Although this is an advanced topic, some of these functions are useful in conjunction with C strings. For example, the `memmove` function can be used to shift characters forward and backward in a string to make room for insertion of a new substring. Similarly, you can use the `memset` function to create a string that's several repeated copies of a character, or to fill a buffer with terminating nulls before writing onto it.

Practice Problems

1. How do you allocate space to hold a copy of a C string?

2. How do you make a copy of a C string?

3. How do you compare two C strings?

4. Explain why the code

   ```
       string myString = "String" + '!';
   ```

   will not work as intended. What is it actually doing? *(Hint: `char`s can implicitly be converted to `int`s.)*

5. Write a function `Exaggerate` that accepts a C string and increases the value of each non-nine digit in it by one. For example, given the input string "I worked 90 hours and drove 24 miles" the function would change it to read "I worked 91 hours and drove 35 miles."

6. Explain why the following code generates an "array bounds overflow" error during compilation:

   ```
       char myString[6] = "Hello!";
   ```

7. When working with C++ strings, the `erase` function can be used as `myString.erase(n)` to erase all characters in the string starting at position n, where n is assumed to be within the bounds of the string. Write a function `TruncateString` that accepts a `char *` C-style string and an index in the string, then modifies the string by removing all characters in the string starting at that position. You can assume that the index is in bounds. *(Hint: Do you actually need to remove the characters at that position, or can you trick C++ into thinking that they're not there?)*

8. There are several ways to iterate over the contents of a C string. For example, we can iterate over the string using bracket syntax using the following construct:

```
int length = strlen(myStr); // Compute once and store for efficiency.
for(int k = 0; k < length; ++k)
    myStr[k] = /* ... */
```

   Another means for iterating over the C string uses pointer arithmetic and explicitly checks for the terminating null character. This is shown below:

```
for(char* currLoc = myStr; *currLoc != '\0'; ++currLoc)
    *currLoc = /* ... */
```

   Write a function `CountFrequency` which accepts as input a C-style string and a character, then returns the number of times that the character appears in the string. You should write this function two ways – once using the first style of for loop, and once using the second.

9. It is legal to provide raw C++ pointers to STL algorithms instead of STL iterators. Rewrite `CountFrequency` using the STL `count` algorithm.

10. Compiler vendors often add their own nonstandard library functions to the standard header files to entice customers. One common addition to the C string library is a function `strcasecmp`, which returns how two strings compare to one another case-insensitively. For example, `strcasecmp("HeLlO!", "hello!")` would return zero, while `strcasecmp("Hello", "Goodbye")` would return a negative value because `Goodbye` alphabetically precedes `Hello`.

    `strcasecmp` is not available with all C++ compilers, but it's still a useful function to have at your disposal. While implementing a completely-correct version of `strcasecmp` is a bit tricky (mainly when deciding how to compare letters and punctuation symbols), it is not particularly difficult to write a similar function called `StrCaseEqual` which returns if two strings, compared case-insensitively, are equal to one another.

    Without using `strcasecmp`, implement the `StrCaseEqual` function. It should accept as input two C-style strings and return whether they are exactly identical when compared case-insensitively. The `toupper`, `tolower`, or `isalpha` functions from `<cctype>` might be useful here; consult a C++ reference for more details. To give you more practice directly manipulating C strings, your solution should not use any of the standard library functions other than the ones exported by `<cctype>`.

11. Another amusing nonstandard C string manipulation function is the `strfry` function, available in GNU-compliant compilers. As described in [GNU]:

    > (`strfry`) addresses the perennial programming quandary: “How do I take good data in string form and painlessly turn it into garbage?” This is actually a fairly simple task for C programmers who do not use the GNU C library string functions, but for programs based on the GNU C library, the `strfry` function is the preferred method for destroying string data.

`strfry` accepts as input a C-style string, then randomly permutes its contents. For example, calling `strfry` on a string containing the text "Unscramble," `strfry` might permute it to contain "barmsc-lUne." Write an implementation of `strfry`. *(Hint: Isn't there an algorithm that scrambles ranges for you?)*

12. Suppose that we want to allocate a C string buffer large enough to hold a copy of an existing C string. A common mistake is to write the following:

```
char* buffer = new char[strlen(kSourceString + 1)]; // <-- Error here!
```

What does this code do? What was it intended to do? Why doesn't it work correctly?