C Strings

Introduction

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++ course would be truly complete without a discussion of C strings. This handout 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 a null-terminated sequences of characters. 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 an understanding of pointer manipulation. Compounding the problem, the C string manipulation functions are cryptic and complicated.

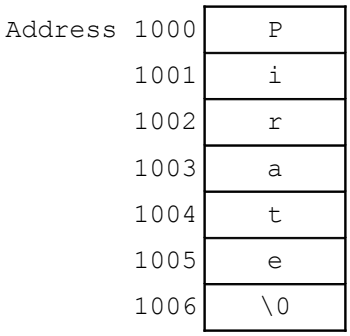
However, because C strings are so low-level, they have several benefits over the C++ `string`. Since C strings are contiguous regions of memory, many of the operations on C strings can be written in lighting-fast assembly code that can outperform even the most tightly-written C or C++ loops. Indeed,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

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 "Pirate" is actually seven characters long in C – six for "Pirate" plus one extra for the terminating null. When working with C strings, most of the time the library functions will 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:



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.

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:

```
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. When writing code that manipulates C strings, if you modify the contents of a read-only segment, you will cause your program to crash with a *segmentation fault* (sometimes also called an *access violation* or “seg fault”).

---

* There's a well-known joke about this: 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.”

Because your program cannot write to read-only segments, if you plan on manipulating the contents of a C string, you will need to first create a copy of that string, usually in the heap, where your program has write permission. Thus, for the remainder of this handout, 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, which will probably crash your program some point down the line.

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[]`.

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;
```

---

* `<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>`.

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 wreck havoc on your program.

Another common string operation is concatenation. To append one C string onto the end of another, use the `strcat` function. However, unlike in C++, when you concatenate two 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. This will only check to see if the two pointers point to the same object, not whether the strings are equal. Thus, you must use the `strcmp` function to compare two strings. `strcmp` compares two strings `str1` and `str2`, returning a negative number if `str1` precedes `str2` alphabetically, a positive number if `str1` comes after `str2`, and zero if `str1` and `str2` are equal. Thus, you can use the following code to check for string equality:

```
// Assume char *str1 and char *str2 are initialized.
if(strcmp(str1, str2) == 0)
    // strings are equal...
```

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)) // Watch out... this is incorrect!
    cout << "one and two are equal!" << endl;
else
    cout << "one and two are not equal!" << endl;
```

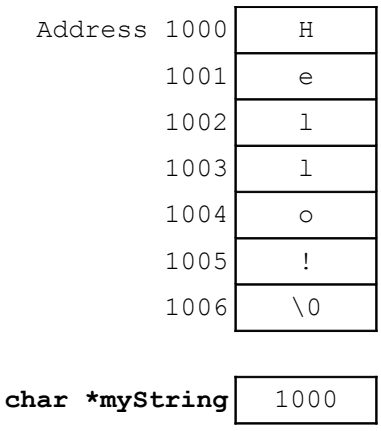
Here, we use the line `if(strcmp(one, two))` to check if `one` and `two` are equal. However, this check is completely wrong. In C++, any nonzero value is treated as "true" inside an if statement and any zero value is treated as "false." However, `strcmp` returns 0 if the two strings are equal and a nonzero 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. Furthermore, if you have been exposed to this material, over the next few weeks as we explore STL iterators, the syntax will make considerably more sense.

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.

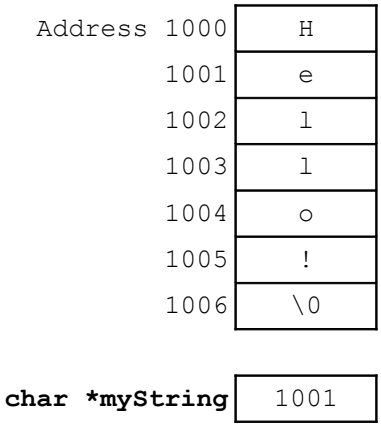
Let's begin with an example using C strings. Suppose you have the string "Hello!" and a pointer to it laid out in memory as shown below:



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 code `myString += 1000`? The string "Hello!" is less than 1000 characters long, and pointer `myString` would point to a value far beyond 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.

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.

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, int start, int length)
{
      char *result = new char[length + 1]; // Include space for \0
      strncpy(result, str + start, length);
      result[length] = '\0'; // Manually append terminating null.
      return result;
}
```

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)`	`int 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`.

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

(C string functions, contd.)

`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 string `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)`
	`int 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`.

(C string functions, contd.)

`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);`
	`int 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 handout 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. **More C++ Pointers**: This handout's section on pointers was significantly limited in the interests of space and readability. We didn't cover the address-of operator `&`, nor did we talk about pointers to pointers or the infamous `void *` pointer. If you plan on using C or C++ more seriously in the future, be sure to read up on your pointers.
2. **`malloc`, `realloc`, and `free`**: These four functions are older C memory management functions that allocate, deallocate, and resize blocks of memory. In conjunction with strings, they are somewhat easier to use than `new` and `delete`. 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. **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.
5. **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. 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.)*
2. Using pointer arithmetic and the `strpbrk` function, write a function `Exaggerate` that increases the value of each non-nine digit in a string by one (see the C++ Strings handout for more information on this function).
3. Explain why the following code generates an "array bounds overflow" error during compilation: `char myString[6] = "String";`