C-Strings

Shaan Mathur

November 14, 2018

It probably needs little to no motivation that the C++ `string` type is insanely useful.

Listing 1: So Many Strings...

```
string username;
cout << "Username: ";
cin >> username;

string password;
cout << "Password: ";
cin >> password;
```

In fact, approaching a program as simple as getting a username and password from the user without `string`s seems like quite the headache. Yet at some point before that class[1] was designed, programmers had to figure out ways to actually manage words when all they had were characters (`char`). This was particularly the case for those programming in C, which still don't have a notion of a `string` variable (or classes for that matter). Let's study their solution.

1 Making Strings Without `string`

A word is just a sequence[2] of characters. We already have the character type of `char`; we can also capture the notion of a sequence using something as simple as an array. Therefore we can view words as just arrays of characters.

So maybe we could do something like this:

```
char word[] = {'H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r',
               'l', 'd', '!'};
```

But as always with arrays, we also need to know the length. So we'd have to have a separate variable to remember the length.

```
long word_length = 13;
```

[1]Yeah `string` is not your mother's kind of variable type. Unlike `int`, `double`, and other primitive types, `string` is a special type known as a *class*; this is just a fancy way of saying it is a programmer-defined type that can take on its own values (e.g. `s = "Hello"`) and has its own behaviors (e.g. `s.size()` or `s += ", World!"`)

[2]Or dare I say, a *string*

So altogether, every time we write a string, we would need to do the following.

Listing 2: A First Attempt at Strings

```c
char word[] = {'H', 'e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l
    ', 'd', '!'};
long word_length = 13;
```

But let's be real, this really sucks. First of all, that is an atrocious way to have to write "Hello, World!". Second of all, we now have to somehow keep track of two different variables in our code: one for the content of the string, and one for the length. Even if it looks simple now, what about if we want an array of strings? We'd have to manage *two* arrays where one corresponds to the contents of the strings and the other corresponds to lengths of those strings. Finally (and perhaps most subtly), the type `long` is finite, meaning we can only represent strings of particular length. Although today longs are 8 bytes, they used to be 4 bytes long before, meaning there were some (absurdly long) strings that could not be properly managed in this representation. It would be nice to have a representation that is agnostic to the length of the string.

However, perhaps we could get away with not having an accompanying variable for length by having some sort of flag at the end of the array marking that the string is terminated. As it turns out, there is a special character known as the null byte[3], '\0', that people have universally decided to use to put at the end of strings. Such strings are called *null-terminated strings* or *C-strings*.

```c
char cstring[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

This convention is so deeply a part of C that the designers decided to make life easier by introducing double quote (`""`) notation, where a word in double quotes is actually just a C-string with the null byte implicitly placed at the end.

```c
char cstring[] = "Hello!"; // cstring[6] == '\0'
```

2 C-String Algorithms

There are plenty of algorithms we can discuss when operating C-strings, but the general structure of them all is a single loop (though not always!) that keeps looping while the current character is not a null byte.

Listing 3: General Looping on C-Strings

```c
char cstring[] = "Hello, World!";
int i = 0;
while (cstring[i] != '\0') {
    // Do something
    i++;
}
```

[3]It's called null because the integer value of '\0' is literally a 0.

In fact if you want to be particularly devious, when evaluated as boolean expressions integer values are interpreted as true when nonzero. So the condition will fail only when a null character (a zero byte) is found.

Listing 4: "Dirty" Looping Trick on C-Strings

```c
char cstring[] = "Hello, World!";
int i = 0;

while (cstring[i]) {
    // Do something

    // If i is irrelevant above, cstring[i++] would also work. Why?
    i++;
}
```

2.1 C-String Length

This one is simple. While we don't see a null character, increment the counter and move forward in the string.

Listing 5: Simple C-String Length

```c
long length(char cstring[]) {
    long count = 0;
    while (cstring[count] != '\0')
        count++;
    return count;
}
```

For those looking for a nice, short, and dirty way to do this...

Listing 6: Dirty C-String Length

```c
long length(char cstring[]) {
    long count = 0;
    while (cstring[count++]); // ; is the empty statement
    return count - 1; // Since we overcount by 1 due to count++
}
```

2.2 C-String Concatenation

This too is simple, but there's a slight hiccup. The parameters we have are a destination C-string and a source C-string, where we append the source C-string to the destination C-string. However we also are (unsafely) assuming that the caller's destination C-string has enough space to actually append the source C-string. This code can lead to some serious bugs, and it actually can lead to serious security risks if malicious input is provided (see buffer overflow attacks if interested).

Other than that hiccup, the algorithm is fairly natural: find the end of the destination string and then start pasting the contents of the source C-string after it.

Listing 7: Simple C-String Concatenation

```
void concat(char dest[], char src[]) {
    long length = 0;
    while (dest[length] != '\0')
        length++;

    // Now length is length of cstring in dest

    // Paste contents of src into end of dest
    long i = 0;
    while (src[i] != '\0') {
        dest[length] = src[i];
        length++;
        i++;
    }

    // Null-terminate c-string in dest
    dest[length] = '\0';
}
```

And although that code is clear, here's a slightly dirtier way to do it.

Listing 8: Dirty C-String Concatenation

```
// Why does this work?
void concat(char dest[], char src[]) {
    long length = -1, i = 0;
    while (dest[++length]);

    do {
        dest[length++] = src[i++];
    } while (src[i - 1]);
}
```

2.3 C-String Reversal

This one is a bit less trivial than the others because usually we approach reversing arrays by swapping the outermost elements, then the next outermost elements, and then the next, and so on. However we do not know the length of the C-string beforehand. But clearly once we know the length, we can just apply the technique of reversing arrays.

Listing 9: C-String Reversal

```c
void reverse(char cstring[]) {
    long length = 0;
    while (cstring[length])
        length++;

    // Now reverse as we would for an array
    for (int i = 0; i < length / 2; i++) {
        char temp = cstring[i];
        cstring[i] = cstring[length - 1 - i];
        cstring[length - 1 - i] = temp;
    }
}
```