Strings

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 22
1.0.0

A String is an Array of Characters

In order to know a string ends, C adds a 0 byte (character `'\0'`)

Assume ASCII encoded strings (characters represented on a US keyboard)

String Literals are Between Double Quotes

For example, `"Hello world"` is a string literal

C stores the bytes used for string literals in read-only memory
Read-only means you're not allowed to modify (assign) values

The result of a string literal is a pointer to the start of the string

String Literals are Between Double Quotes

For example, `"Hello world"` is a string literal

C stores the bytes used for string literals in read-only memory
  Read-only means you're not allowed to modify (assign) values

The result of a string literal is a pointer to the start of the string

The type of a string literal is: `const char *`
  It's a pointer (array decay) representing an array of `char`
    `const` means we cannot assign the `char` values

The Format Specifier for C Strings is `%s`

We can use it in `printf`, like so:

```
printf("%s %s\n", "Hello" "world");
```

C will print all of the characters, in order,
up to (and not including) the null byte (`0`)

We'll see later that using `%s` in `scanf` is not a good idea

We Cannot Modify the Values in a String Literal

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *s = "Hello world";
    s[0] = 'h';
    printf("s: %s\n", s);
    return EXIT_SUCCESS;
}
```

We can compile this program, and when we run it we get a segfault

We Should Use `const char *` for String Literals

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *s = "Hello world";
    s[0] = 'h';
    printf("s: %s\n", s);
    return EXIT_SUCCESS;
}
```

Now we get a compiler error instead!

```
error: assignment of read-only location '*s'
  s[0] = 'h';
       ^
```

We Can Create an Array in a Function, and Modify the String

Recall, if we declare an array using: `char s[] = /* ... */;`
- The array is stored on the stack

There's a special rule for string literals, if we do:
`char s[] = /* string literal */;`
- C copies the `char` values onto the stack for `s`

This means we're allowed to modify the string

We Can Modify a String on the Stack

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char s[] = "Hello world";
    s[0] = 'h';
    printf("s: %s\n", s);
    return EXIT_SUCCESS;
}
```

Running this program prints:

```
s: hello world
```

C Knows How Many Bytes to Copy from a String Literal

If we write: `char s[] = "Hello";`
`ARRAY_LENGTH(s)` → `6`

It would be equivalent if we did:

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

We Could Create a Larger Array than We Need

For example: `char s[8] = "Hello";`

The same rule applies as before, the rest of the array fills with 0 values

It would be equivalent if we did:

```
char s[8] = {'H', 'e', 'l', 'l', 'o', '\0', '\0', '\0'};
```

We Could Create a Larger Array than We Need

For example: `char s[8] = "Hello";`

The same rule applies as before, the rest of the array fills with 0 values

It would be equivalent if we did:

```
char s[8] = {'H', 'e', 'l', 'l', 'o', '\0', '\0', '\0'};
```

We would stop when we encounter the first `'\0'`, so the others don't matter

However, A Smaller Array Causes Big Issues

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char s[8] = "Hello world";
    printf("%s\n", s);
    return EXIT_SUCCESS;
}
```

We would only copy `"Hello wo"` with no null byte

Our output would only stop when we happen to find a 0 in memory

We Can Write a Function to Count the Number of Spaces

```c
#include <stdio.h>

int countSpaces(const char *s) {
    int count = 0;
    while (*s != '\0') {
        if (*s == ' ') {
            ++count;
        }
        ++s;
    }
    return count;
}

int main(void) {
    const char *s = "Hello world";
    printf("countSpaces(s): %d\n", countSpaces(s));
    return 0;
}
```

We Could Also Count the Number Characters in a String

```c
#include <stdio.h>
#include <stdlib.h>

int stringLength(const char *s) {
    int i = 0;
    while (s[i] != '\0') {
        ++i;
    }
    return i;
}

int main(void) {
    const char *s = "Hello";
    printf("stringLength(s): %d\n", stringLength(s));
    return EXIT_SUCCESS;
}
```

Note: this is one less than the number of bytes we need to store a C string!

We Can Use `puts` Instead of `printf` for Strings

The `puts(s)` takes a single string argument
  It's equivalent to: `printf("%s\n", s);`

We Can Use “Rounding” to Print the First Few Characters

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *s = "Hello there";
    puts(s);
    printf("%s\n", s + 6);
    printf("%.5s\n", s);
    return EXIT_SUCCESS;
}
```

We Can Use “Rounding” to Print the First Few Characters

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *s = "Hello there";
    puts(s);
    printf("%s\n", s + 6);
    printf("%.5s\n", s);
    return EXIT_SUCCESS;
}
```

Prints:

```
Hello there
there
Hello
```

We Could, But Shouldn't Use `scanf` with Strings

`scanf` ignores whitespace before characters the user enters
- Matches any characters until a whitespace character
  - It'll also terminate the string with a 0 byte

This means the string produced will not have any whitespace characters

This function is **impossible** to use correctly since
you don't know how much memory you need

If the User Enters Too Many Characters, `scanf` May Use Invalid Memory

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 123456789;
    printf("Input your last name: ");
    char s[4];
    scanf("%s", s);
    printf("s: %s\n", s);
    printf("x: %d\n", x);
    return EXIT_SUCCESS;
}
```

Any input of 4 characters changes x

The `gets` Function has the Same Issue

`gets` is the opposite of `puts`
- `gets(s)` reads all character until a newline
  (but doesn't keep the newline character)

The issue is called a buffer overflow
- We're writing data beyond the space we set aside
  - We call the memory we intend a function to write to a buffer

We Could Use `fgets` Instead of `gets`

`fgets` allows you to specify the size of the buffer, its API is:

```
char* fgets(char *str, int size, FILE *stream);
```

For this function:

- `str` is a pointer to `size` bytes of valid memory
- `size` is an integer representing how many bytes to write (at most)
- `stream` should just be `stdin` (represents terminal input, declared in `stdio.h`)

The produced string will have a null byte,
so there's `size - 1` characters in the string

We'll Print at Most 3 Characters from the User's Input

```c
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 4

int main(void) {
    printf("Input your last name: ");
    char s[BUFFER_SIZE];
    fgets(s, BUFFER_SIZE, stdin);
    printf("s: %s\n", s);
    return EXIT_SUCCESS;
}
```

You Should Use `getline`

`getline` will allocate memory for you, it's API is:

```
ssize_t getline(char **bufferp, size_t *sizep, FILE *stream);
```

If you initialize the value at `bufferp` with `NULL` and `sizep` it'll `malloc` for you

It will match an entire line including whitespace and the ending newline

It returns the number of characters written (excluding the null byte)

A `getline` Example That Remembers to `free`

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *s = NULL;
    size_t size = 0;
    ssize_t bytes_written = getline(&s, &size, stdin);
    s[bytes_written - 1] = '\0'; /* Replace the newline with a null byte */
    printf("size: %lu, bytes_written: %ld, s: %s\n", size, bytes_written, s);
    free(s);
    return EXIT_SUCCESS;
}
```

Strings Use Memory, and are Difficult to Use Correctly

You need to always ensure there's a null byte at the end

The format specifier (that you should only use for `printf`) is `%s`

You need to make sure there's enough memory to hold the string
  Buffer overflows are a serious security issue

For user input you should use either `fgets` or `getline`