



CS107 Lecture 7

More C Strings

Reading: K&R (1.9, 5.5, Appendix B3) or Essential C section 3

The `string.h` library: `strchr`

`strchr` returns a pointer to the **first occurrence** of a character in a string, or **NULL** if the character is nowhere to be found.

```
char laureate[] = "Katalin Kariko";  
char *first = strchr(laureate, 'a');  
printf("%s\n", first);           // atalin Kariko  
char *second = strchr(first + 1, 'a');  
printf("%s\n", second);         // alin Kariko
```

Recall this declaration and initialization of **`laureate`** allocates an array of just the right size and effectively **`strcpy`**'s the string constant into it.

Thought Question: Why **`first + 1`**?
What's wrong with just **`first`**?

Use **`strrchr`** to find the last occurrence of a character (or **NULL** on failure).

```
char *last = strrchr(laureate, 'a');  
printf("%s\n", last);           // prints ariko
```

Aside: Katalin Karikó is a Hungarian-American biochemist whose decades-long research on messenger RNA laid the foundation for mRNA vaccines. After years of limited recognition, her work proved essential to the rapid development of COVID-19 vaccines and immunizations. She shared the 2023 Nobel Prize in Physiology or Medicine with Drew Weissman.

The `string.h` library: `strstr`

`strstr` returns a pointer to the **first occurrence** of a substring within a larger string, or **NULL** if the substring isn't present.

```
char laureate[] = "Carolyn Bertozzi";  
char *only = strstr(laureate, "zz");  
printf("%s\n", laureate); // prints Carolyn Bertozzi  
printf("%s\n", only);    // prints zzi
```

Note: There is no built-in **`strrstr`**, likely because reverse substring search is uncommon and, unlike **`strchr`**, doesn't have an especially simple linear-time implementation.

Aside: Carolyn Bertozzi is an American chemist known for pioneering bioorthogonal chemistry, enabling chemical reactions inside living cells without disrupting their biology. She's a professor here and shared the 2022 Nobel Prize in Chemistry with Barry Sharpless of Scripps and Morten Meldal of the University of Copenhagen.

Implementing **strrstr** anyway

Assuming you've a legitimate need for **strrstr** anyway, implement it to return a pointer to the last occurrence of substring within a larger string.

```
char *strrstr(char *haystack, char *needle) {  
    char *curr = haystack;  
    char *last = NULL;  
    while (true) {  
        curr = strstr(curr, needle);  
        if (curr == NULL) return last; // no more matches  
        last = curr;  
        curr++;  
    }  
}
```

Thought Question: Think about how expensive the above implementation is when you reverse search for last **"wwwwww"** in **"wwwwwwwwwwwwwwwwwwwwwwwwwwwxyz"**. What's the runtime behavior in this case?

Thought Question: What about searching for the reverse of **needle** within the reverse of haystack? Can that be made to work? Is it clearly better than the above in all cases?

The `string.h` library: `strspn`, `strcspn`

`strspn(str, accept)` returns the **length of the initial segment** of **`str`** consisting entirely of characters **found** in **`accept`**.

```
char laureate[] = "Karl Deisseroth";  
size_t count = strspn(laureate, "Kramer"); // count gets a 3
```

`strcspn(str, reject)` also returns the **length of the initial segment** of **`str`**, this time consisting entirely of characters **not found** in **`reject`**. Here the **`c`** stands for complement, as in the complement of a set.

```
char medalist[] = "Maryam Mirzakhani";  
size_t length = strcspn(medalst, "Field"); // length gets an 8
```

Aside: Karl Deisseroth is a Stanford professor who pioneered optogenetics and received the 2026 Nobel Prize in Physiology or Medicine 🥰 **this past Monday** 🥰. Maryam Mirzakhani was a Stanford mathematician who, in 2014, became the first woman to win the Fields Medal.

C Strings As Parameters

When we pass a **mutable string** as a parameter, it is often passed as a **char *** as a **visual cue** that it's a C string. We can, however, manipulate the string exactly as if it were declared as a **char []**.

```
void reverse(char *s) {  
    if (s[0] == '\\0') return;  
    size_t lh = 0, rh = strlen(s) - 1;  
    while (lh < rh) {  
        char temp = s[lh];  
        s[lh] = s[rh];  
        s[rh] = temp;  
        lh++;  
        rh--;  
    }  
}
```



SAME

```
void reverse(char s[]) {  
    if (s[0] == '\\0') return;  
    size_t lh = 0, rh = strlen(s) - 1;  
    while (lh < rh) {  
        char temp = s[lh];  
        s[lh] = s[rh];  
        s[rh] = temp;  
        lh++;  
        rh--;  
    }  
}
```

Thought Question: Why do we call out the case where `s[0] == '\\0'`?

When we pass a **string constant** as a parameter (or even a mutable string that shouldn't be changed), it's best to accept it as a **const char ***. This mandates that the string's characters be respected as **frozen** and **immutable**.

Password Validation

Write a function **validate** that accepts a **candidate password** alongside some constraints and returns **true** if and only if the candidate password is valid.

In particular, **candidate** is valid if and only if it's constructed using only those letters found in **permitted**, and none of the **forbidden** strings appear anywhere within it.

None of the strings need to change anywhere—hence the **const char** *s. All data is treated as read-only.

```
bool validate(const char *candidate, const char *permitted,
              const char *forbidden[], size_t length) {
    if (strspn(candidate, permitted) != strlen(candidate))
        return false;

    for (size_t i = 0; i < length; i++) {
        if (strstr(candidate, forbidden[i]) != NULL)
            return false;
    }

    return true;
}
```

If any character outside **permitted** appears within **candidate**, the **span length** will be **less than** the **full string length**.

If the **ith forbidden** word **appears anywhere within candidate**, **strstr** will return something **non-NULL**. We don't care what **strstr**'s exact return value is. We only care if it's **NULL** versus something else.

Buffer Overflow, Take II

We must **ensure that there is enough space at the destination** to house the entire copy, **including the null character**.

```
char greeting[8]; // not enough space
strcpy(greeting, "hello, world!"); // overwrites other memory
```

- Buffer overflows are dangerous because they allow **data meant for one part of memory to spill over into another**, potentially overwriting variables, control information, or even assembly code instructions.
- Overwrites can lead to **instability, crashes**, and the **execution of unknown code**.
- It's our job as programmers to **identify** buffer overflows (and other bugs, of course) and **fix** them, not just so our programs **run as intended**, but also to **protect** anyone who uses our software.

Infamous Buffer Overflow Exploits

AOL Instant Messenger Exploit

A flaw in AOL Instant Messenger allowed attackers to send specially crafted requests that caused data to be written beyond the bounds of the fixed-size buffer intended to hold it, intentionally corrupting adjacent memory.

AIM was written in C and C++, so copying attacker-controlled data into a fixed-size buffer without adequate bounds checking could overwrite control data and redirect program execution to attacker-injected code supplied within the request itself.

Other than being logged in, nothing was required of the AIM user. Simply receiving a malicious request was enough to trigger the buggy code.

The bug allowed attackers to execute arbitrary programs with the privileges of the logged-in user.

Scott and Jerry met via Instant Messenger, back when people lied about meeting others online and instead said they met through mutual friends, or at a party, or at church.

Morris Internet Worm

A self-replicating program—written by Cornell graduate student Robert Tappan Morris and launched from an MIT machine—exploited several vulnerabilities and weaknesses in Internet services common at the time to spread from machine to machine.

Among other things, it exploited a buffer overflow in a network service called **fingerd**, where arbitrarily long text was copied into a fixed-size buffer and overwrote adjacent stack memory.

By overwriting control data via the overflow, the worm redirected execution to attacker-injected code, allowing arbitrary commands to run remotely on vulnerable Unix machines.

After gaining control, the worm transferred a small bootstrap program (!) to download additional code (!!) and launch new processes (!!!), propagating automatically to additional machines.

Jerry was a sophomore at MIT at the time but didn't know any of this happened. Recall he'd given up on CS the year prior because **for** loops confused him.

Avoiding Buffer Overflow

There's **no single solution** that works for everything. **Finding and repairing overflow vulnerabilities** require a **combination of software development techniques**.

- **vigilance** while programming (**scrutinizing** array reads and writes, pointer arithmetic)
- **carefully reading** documentation
- **thoroughly documenting** assumptions in your own code, particularly when others are expected to use it
- **thoroughly testing** to identify issues before shipping product, specifically designing tests to **verify overflow** is either **impossible** or **detected and handled gracefully**
- **using software tools** to methodically **monitor code** for **illegal memory access** and **suspicious function calls** (example: **valgrind**)

Avoiding Buffer Overflow

myth's man page for **gets**:

```
char *gets(char *s);
```

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. **Use fgets() instead.**

Avoiding Buffer Overflow

[Linux man page](#) for **strcpy/strncpy**:

The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy. **Beware of buffer overruns!**

If the destination string of a `strcpy()` is not large enough, then **anything might happen**. Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine. Any time a program reads or copies data into a buffer, the program first needs to check that there's enough space. This may be unnecessary if you can show that overflow is impossible, but be careful: programs can get changed over time, in ways that may make the impossible possible.

Demo: Memory Errors

