



CS107 Lecture 6

C Strings

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

C Strings

How can a computer represent more complex data items, like text?

Why answering this question is important:

- Illustrates **how strings are represented** in C and other languages (today)
 - **Hot take 1:** C strings aren't a true data type so much as they are a contract .
 - **Hot take 2:** Every convenience you're used to from other languages—length tracking, bounds checks, immutability—is **gone** and **your** responsibility now.
- Helps us better understand **buffer overflows** (today and Wednesday)
 - **Hot take 3:** A good number of infamous bugs and security exploits reduce to "someone misunderstood C strings".
- Serves as a gentle reintroduction to pointers (Wednesday and Friday)

The char type

A **char** is a variable type that represents a single character (aka a "glyph").

```
char letter = 'M';  
char plus = '+';  
char space = ' ';
```

```
char newline = '\n';  
char tab = '\t';  
char quote = '\'';
```

```
char backslash = '\\';
```

ASCII

Under the hood, C represents each **char** as a single-byte integer that serves as its ASCII value.

- Uppercase letters are numbered sequentially.
- Lowercase letters are numbered sequentially.
- Digit characters are numbered sequentially.
- Lowercase letters are 32 more than their uppercase equivalents (via a bit flip!)

```
char upper = 'A'; // really 65
char lower = 'a'; // really 97 (i.e., 'A' + 32)
char zero = '0'; // really 48
```

Common ctype.h Functions

Function	Description
isalpha(ch)	true if ch is 'a' through 'z' or 'A' through 'Z'
islower(ch)	true if ch is 'a' through 'z'
isupper(ch)	true if ch is 'A' through 'Z'
isspace(ch)	true if ch is a space, tab, new line, etc.
isdigit(ch)	true if ch is '0' through '9'
toupper(ch)	returns uppercase version of a letter
tolower(ch)	returns lowercase version of a letter

C Strings: The Agreement

C doesn't include a **dedicated** data type for strings. Instead, a string is represented as an **array of chars** with a **sentinel** marking its end.

"Hello"	<i>index</i>	0	1	2	3	4	5
	<i>char</i>	'H'	'e'	'l'	'l'	'o'	'\0'

'\0' is the **null character**, and you always need one extra byte in the array for it. You'll also hear it called the **null byte** or the **zero byte**. I use all three interchangeably.

String Length

C strings are **not** objects. (In fact, nothing in C is an object.) If we want to compute the length of the string, we must compute it ourselves.

<i>index</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>value</i>	'H'	'e'	'l'	'l'	'o'	','	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

We typically call the built-in **strlen** function to compute string length. The null byte doesn't contribute to the computed length.

```
size_t len = strlen(str); // e.g., 13
```

Alert: strlen runs in linear time: It walks the entire string from beginning to end and counts. Save the length if you need it later.

Common `string.h` Functions

Function	Description
strlen(str)	returns the number of chars in a C string (excluding the ' \0 ').
strcmp(str1, str2), strncmp(str1, str2, n)	compares two strings and returns 0 if identical, < 0 if str1 comes before str2 in alphabet, > 0 if str1 comes after str2 in alphabet. strncmp stops comparing after at most n characters.
strchr(str, ch) strrchr(str, ch)	character search: returns a pointer to the first occurrence of ch in str , or NULL if ch isn't found. strrchr find the last occurrence.
strstr(haystack, needle)	string search: returns a pointer to the start of the first occurrence of needle in haystack , or NULL if needle isn't there.
strcpy(dst, src), strncpy(dst, src, n)	copies characters in src to dst , including null-terminating character. Assumes enough space in dst . strncpy stops after at most n chars, and does not add ' \0 ' unless strlen(src) < n .
strcat(dst, src), strncat(dst, src, n)	concatenate src onto the end of dst . strncat stops concatenating after at most n characters. This one always adds the ' \0 '.
strspn(str, accept), strcspn(str, reject)	strspn returns the length of the initial segment of str consisting entirely of characters from accept . strcspn returns the length of the initial segment of str consisting of characters not found in reject .

The `string.h` library: `strcmp`

`strcmp(str1, str2)`: compare two strings (note: `==`, `<`, etc. don't work)

- returns 0 if both strings are truly identical
- `< 0` if **`str1`** is lexicographically smaller than **`str2`**
- `> 0` if **`str1`** is lexicographically larger than **`str2`**

```
int cmp = strcmp(str1, str2);
if (cmp == 0) {
    // equal
} else if (cmp < 0) {
    // str1 comes before str2
} else {
    // str1 comes after str2
}
```

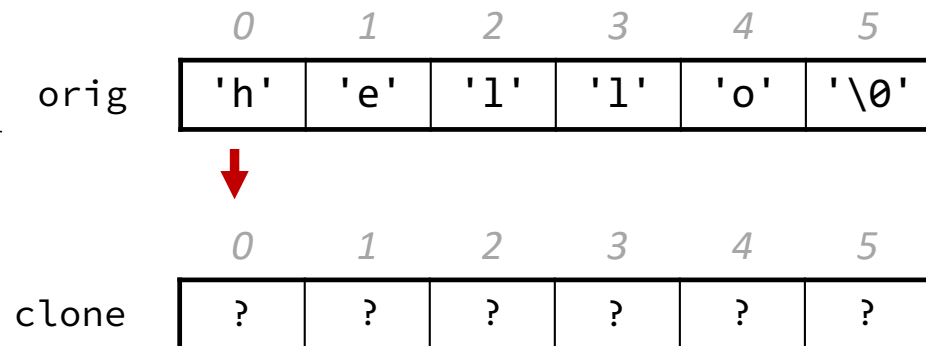
The `string.h` library: `strcpy`

`strcpy(dst, src)`: copies the contents of the **`src`** string—assumed to be **`'\0'`**-terminated—into the space addressed by **`dst`**.

```
char orig[6]; // include space for '\0'
strcpy(orig, "hello");

char clone[6];
strcpy(clone, orig); // strcpy copies the '\0' as well
clone[0] = 'c';

printf("%s", orig); // hello
printf("%s", clone); // cello
```



The string.h library: strcpy

We **must make sure there is enough space** at the destination to hold the **entire copy**, including the `'\0'` character.

```
char tiny[6];  
strcpy(tiny, "hello, world!"); // bigger than tiny can bear
```

Writing past array bounds is an example of a **buffer overflow**.

Buffer overflows are a leading source of security bugs and account for a disproportionate share of vulnerabilities.



In this case, the **buffer overflow** could very well overwrite memory set aside for other local variables, since they are generally allocated next to each other.



The `string.h` library: `strncpy`

- **`strncpy(dst, src, n)`**: writes exactly **`n`** bytes into **`dst`**, padding with **`'\0'`** if **`src`** is shorter than **`n`**. If **`src`** is of length **`n`** or more, **`dst`** won't be null-terminated.

```
char tight[8];  
strncpy(tight, "enormous", 8); // no room for '\0'  
char snug[8];  
strncpy(snug, "perfect", 8);   // exactly one '\0'  
char roomy[8];  
strncpy(roomy, "svelte", 8);   // writes "svelte\0\0"
```

When we fail to terminate a character array with a **`'\0'`** but treat it as a C string **anyway**, string library functions have undefined behavior—e.g., **`strlen`** may read beyond its bounds in search of a mystery **`'\0'`**!

Fun fact: **`strncpy`** is considered [sufficiently dangerous](#) that, as of this past June, it has now been completely removed from the Linux kernel. Most string-copy uses were migrated to safer alternatives such as **`strscpy`**.

string.h Etudes

What value should fill in the blank?

- A. 4
- B. 5
- ☒ C. 6
- D. 12

Thought question: Will 12 work, even if it's not ideal? What about `strlen("hello") + 1`?

```
char str[_____];  
strcpy(str, "hello");
```



string.h Etudes

What is printed out by the following program?

```
int main(int argc, char *argv[]) {  
    char str[9];  
    strcpy(str, "Hi earth");  
    str[2] = '\\0';  
    printf("str = %s, len = %zu\\n",  
          str, strlen(str));  
    return 0;  
}
```

- A. str = Hi, len = 8
- ☒ B. str = Hi, len = 2
- C. str = Hi earth, len = 8
- D. str = Hi earth, len = 2



The `string.h` library: `strcat`, `strncat`

`strcat(dst, src)`: appends the contents of **`src`** to the contents of **`dst`**

`strncat(dst, src, n)`: same, but appends at most **`n`** characters from **`src`**

```
char greeting[13];           // enough space for strings + '\0'
strcpy(greeting, "hello ");
strcat(greeting, "world!");
printf("%s", greeting);      // hello world!
```

```
char alert[16];
strcpy(alert, "Alert: ");
strncat(alert, "overflow detected", sizeof(alert) - strlen(alert) - 1);
printf("%s\n", alert);       // Alert: overflow
```

Both **`strcat`** and **`strncat`** overwrite the old `'\0'` and add a new one at the end. Restated, both assume the destination already contains a properly null-terminated string and has enough room for everything they append, plus the final `'\0'`.

Note that we can't concatenate strings using `+` as we can in C++ and Python. That's a modern idea that C is too old, stubborn, and curmudgeonly to support.

C Strings As Parameters

When we pass a C string to a function, the array **decays** to a **char ***: C passes the address of its first character rather than copying the whole array.

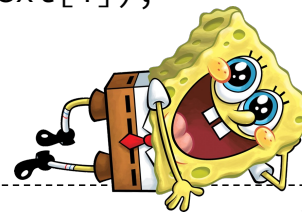
```
int main(int argc, char *argv[]) {
    char reprimand[] = "I told you to clean your room!!";
    printf("SpongeMom: %s\n", reprimand);
    mockmeme(reprimand); // same as mockmeme(&reprimand[0]);
    printf("SpongeBob: %s\n", reprimand);
    return 0;
}

void mockmeme(char *text) { // catch location of some C string
    bool upper = true;
    for (size_t i = 0; i < strlen(text); i++) {
        if (isalpha(text[i])) {
            text[i] = upper ? toupper(text[i]) : tolower(text[i]);
            upper = !upper;
        }
    }
}
```

The declaration of **reprimand** allocates a **char** array of length 32 and effectively **strcpy**'s the string into it.

Passing a C string to **mockmeme** is really passing the address of its leading **char**—or equivalently, the base address of the full **char** array.

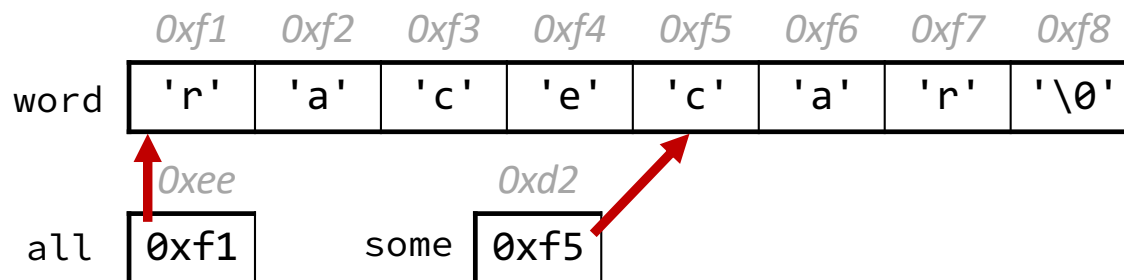
Changes through **text** are reflected in the original **reprimand**.



C Strings and Suffixes

Since C strings are **pointers to characters**—i.e., **char ***s—we can adjust the pointer to overlook characters at the beginning.

```
char word[8];  
strcpy(word, "racecar");    // fits perfectly!  
char *all = word;  
char *some = word + 4;      // same as some = &word[4]  
printf("%s\n", all);        // prints racecar  
printf("%s\n", some);       // prints car
```

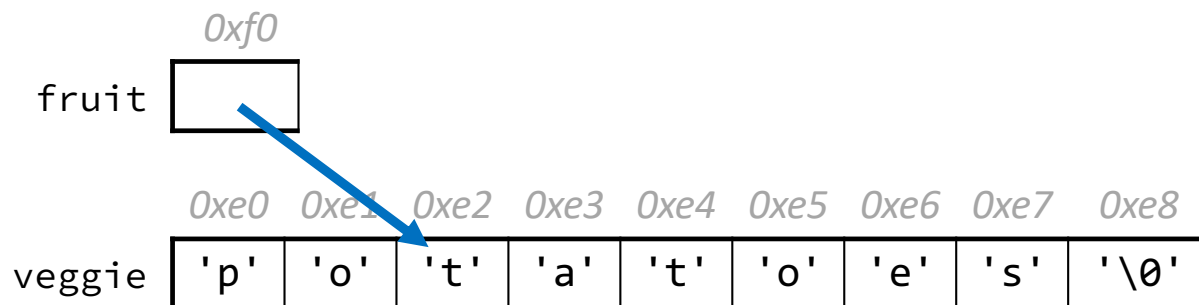


string.h Etudes

What is printed out by the following program?

```
int main(int argc, char *argv[]) {  
    char veggie[9];  
    strcpy(veggie, "potatoes");  
    char *fruit = veggie + 2;  
    strcpy(fruit, "mag");  
    printf("%s\n", veggie);  
}
```

- a. magoes
- b. magtoes
- c. pomag
- d. pomagoes
- e. pomitoes
- f. pomidoes

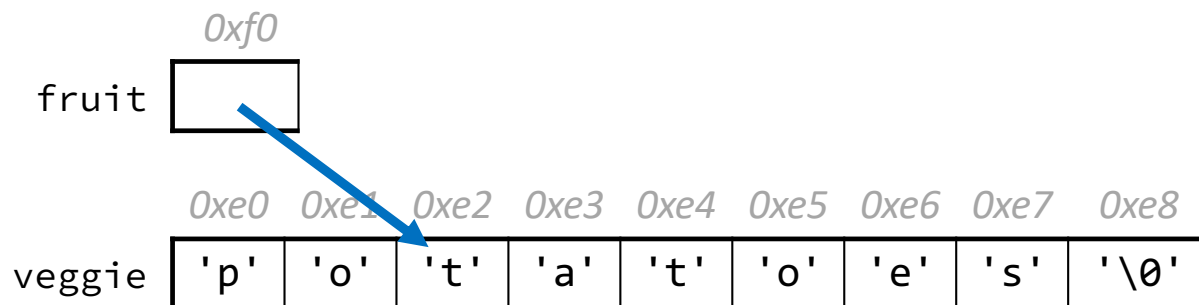


string.h Etudes

What is printed out by the following program?

```
int main(int argc, char *argv[]) {  
    char veggie[9];  
    strcpy(veggie, "potatoes");  
    char *fruit = veggie + 2;  
    strncpy(fruit, "mid", 2);  
    printf("%s\n", veggie);  
}
```

- a. magoes
- b. magtoes
- c. pomag
- d. pomagoes
- e. pomitoes
- f. pomidoes



String Diamonds

Write a function **diamond** that accepts a string as its only parameter and prints its letters in a diamond format as shown below.

For example,
diamond("doris")
should print:

```
d
do
dor
dori
doris
 oris
  ris
   is
    s
```

```
void diamond(char *str) {
    size_t length = strlen(str);
    for (size_t i = 1; i < length; i++) {
        char prefix[i + 1];
        strncpy(prefix, str, i);
        prefix[i] = '\0';
        printf("%s\n", prefix);
    }
    printf("%s\n", str);
    for (size_t i = 1; i < length; i++) {
        for (size_t j = 0; j < i; j++) {
            printf(" ");
        }
        printf("%s\n", str + i);
    }
}
```

Each iteration of the first **for** loop prints each **prefix**. **strncpy** is the right call here—we want to copy exactly **i** visible **chars**—but we need to stamp down our own **'\0'**.

Printing **suffixes** is even easier, since the **ith** iteration can skip the first **i** characters when printing by adding **i** to **str**.

