CS107, Lecture 6
C Strings

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

Ed Discussion: https://edstem.org/us/courses/28214/discussion/1877377

<u>CS107 Topic 2</u>: How can a computer represent and manipulate more complex data like text?

CS107 Topic 2

How can a computer represent and manipulate more complex data like text?

Why is answering this question important?

- Shows us how strings are represented in C and other languages (this time)
- Helps us better understand buffer overflows, a common bug (next time)
- Reintroduces us to pointers, because strings can be pointers (next Wednesday)

char

A **char** is a variable type that represents a single character or "glyph".

```
char letter = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
char newline = '\n';
char tab = '\t';
char single_quote = '\'';
char backslash = '\\';
```

ASCII

Under the hood, C represents each **`char`** as an *integer* (its "ASCII value").

- Uppercase letters are sequentially numbered
- Lowercase letters are sequentially numbered
- Digits are sequentially numbered
- Lowercase letters are 32 more than their uppercase equivalents (bit flip!)

```
char upper = 'A';      // Actually 65
char lower = 'a';      // Actually 97
char zero = '0';       // Actually 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 equivalent of a letter
`tolower(`***ch***`)`	returns lowercase equivalent of a letter

Remember: these **return** a char; they cannot modify an existing char!

More documentation with `man isalpha`, `man tolower`

Common `ctype.h` Functions

```
bool isLetter = isalpha('A');      // true
bool capital = isupper('f');       // false
char uppercaseB = toupper('b');
bool isADigit = isdigit('4');      // true
```

C Strings

C has no dedicated variable type for strings. Instead, a string is represented as an **array of characters** with a sentinel value marking its end.

`"Hello"`

index	*0*	*1*	*2*	*3*	*4*	*5*
char	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`'\0'`

`'\0'` is the **null-terminating character,** and you always need one extra space in an array for it.

String Length

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

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

We call the built-in **`strlen`** function to calculate string length. The null-terminating character doesn't contribute to a C string's length.

```
int length = strlen(myStr);      // e.g. 13
```

Caution: `strlen` is `O(N)` because it must scan the entire string! We should save the value if we plan to refer to the length later.

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char ***. C passes the location of the first character rather than a copy of the whole array.

```
int foo(char *str) {
    ...
}

char string[6];
...
foo(string); // equivalently foo(&str[0])
```

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **`char *`**. C passes the location of the first character rather than a copy of the whole array.

```
int foo(char *str) {
    ...
    str[0] = 'c’;           // modifies original string!
    printf("%s\n", str);    // prints cello
}

char string[6];
... // code to build string to be "Hello"
foo(string);
```

We still use a char * the same way we use a char[].

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`)`, `strncmp(`***`str1, str2, n`***`)`	compares two strings; 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*** was not found in ***str***. `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*** was not found in ***haystack***.
`strcpy(`***`dst, src`***`)`, `strncpy(`***`dst, src, n`***`)`	copies characters in ***src*** to ***dst***, including null-terminating character. Assumes enough space in ***dst***. Strings must not overlap. **`strncpy`** stops after at most ***n*** chars, and does not add null-terminating char.
`strcat(`***`dst, src`***`)`, `strncat(`***`dst, src, n`***`)`	concatenate ***src*** onto the end of ***dst***. **`strncat`** stops concatenating after at most ***n*** characters. Always adds a null-terminating character.
`strspn(`***`str, accept`***`)`, `strcspn(`***`str, reject`***`)`	**`strspn`** returns the length of the initial part of ***str*** which contains only characters in ***accept***. **`strcspn`** returns the length of the initial part of ***str*** which does not contain any characters in ***reject***.

The string library: `strcmp`

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

- returns 0 if both strings are 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
}
```

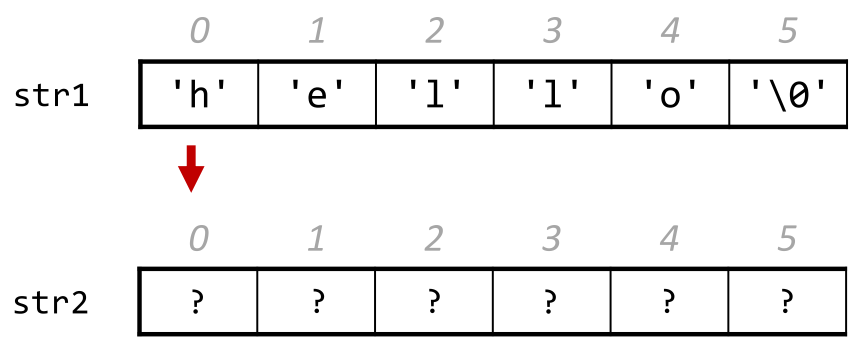
The string library: `strcpy`

`strcpy(dst, src)`: copies the contents of **src** into the string **dst**, including the null terminator. (*Note that you can't copy a C string using =.*)

```
char str1[6]; // include space for '\0'
strcpy(str1, "hello");

char str2[6];
strcpy(str2, str1);
str2[0] = 'c';

printf("%s", str1);        // hello
printf("%s", str2);        // cello
```

Copying Strings - `strcpy`

```c
char str1[6];
strcpy(str1, "hello");

char str2[6];
strcpy(str2, str1);
```



Copying Strings - `strcpy`

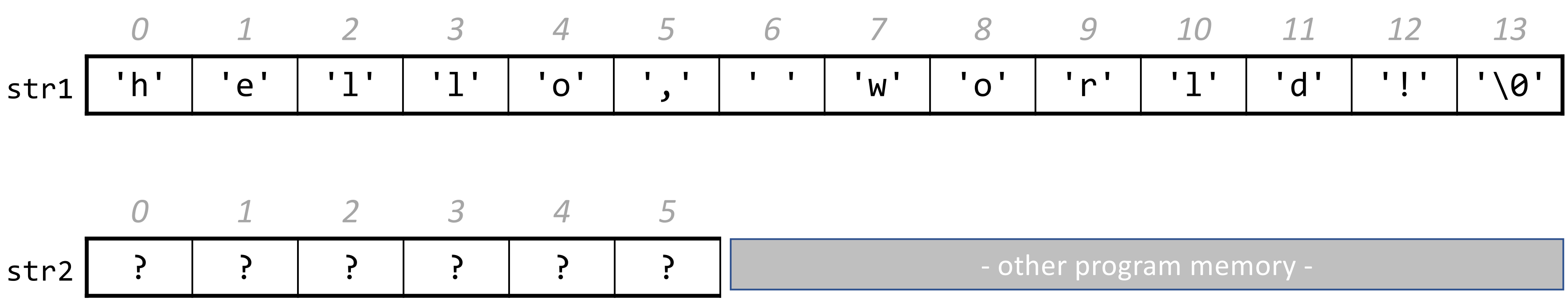
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

```
char str2[6];                       // not enough space!
strcpy(str2, "hello, world!");   // overwrites other memory!
```

Writing past memory bounds is called a "buffer overflow”.

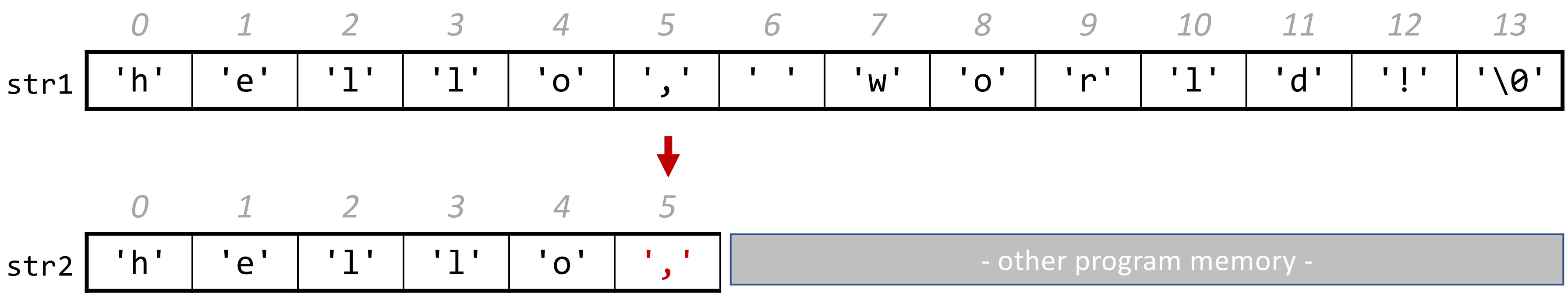
Copying Strings – Buffer Overflows

```c
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```



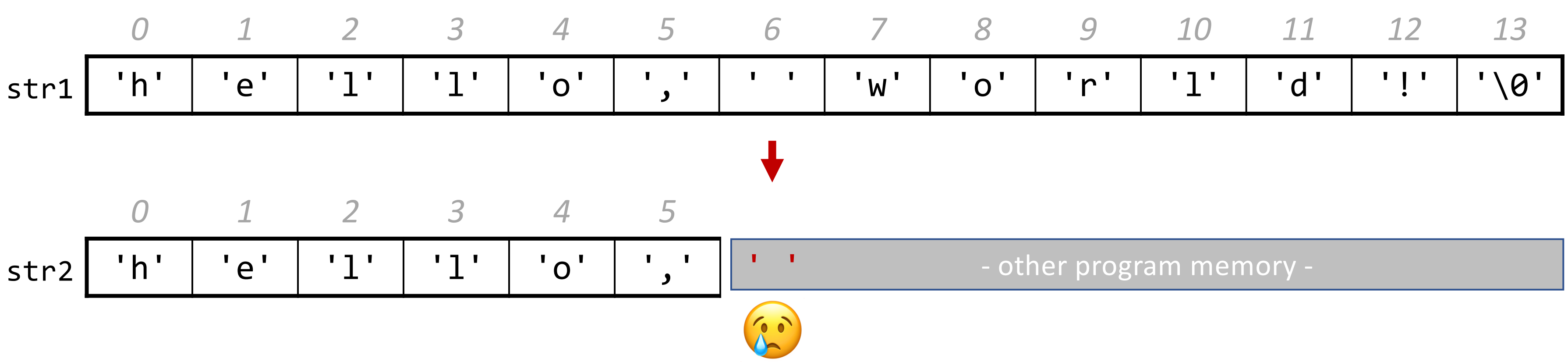
Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```



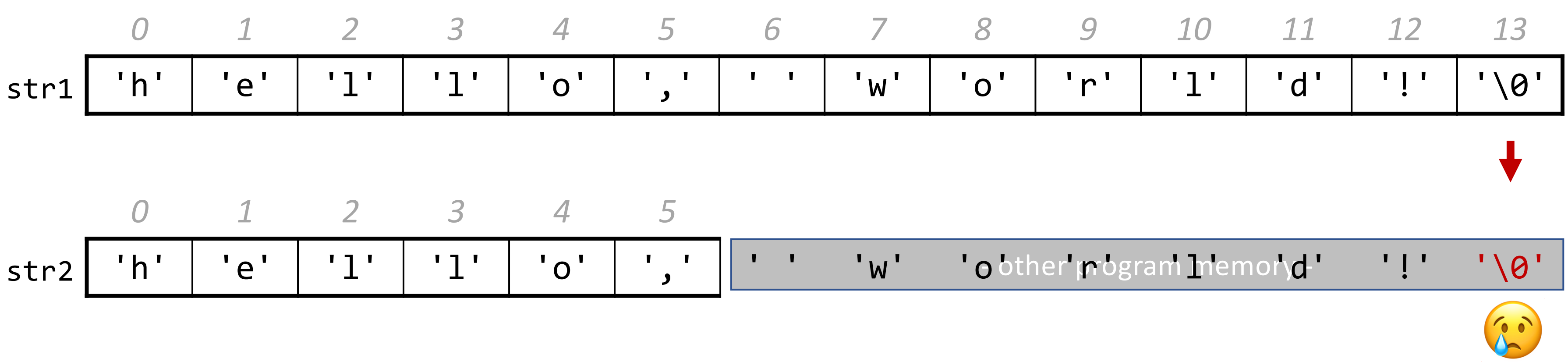
Copying Strings – Buffer Overflows

```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```



Copying Strings – Buffer Overflows

```c
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];
strcpy(str2, str1);   // not enough space - overwrites other memory!
```



Copying Strings - `strncpy`

`strncpy(dst, src, n)`: copies at most the first n bytes from **src** into the string **dst**. If there is no null-terminating character in these bytes, then **`dst`** won't get a null terminator either.

```
// copying "automata problem"
char str[7];
strncpy(str, "problem", 7); // doesn't write a '\0'!
```

When we fail to terminate a character array with a `'\0'` but treat it as a C string anyway, we can't expect C string functions to work properly, e.g., **strlen** may continue reading beyond the bounds of **str** in search of `'\0'`!

String Copying Exercise

What value should go in the blank at right?

A. 4

B. 5

C. 6

D. 12

E. strlen("hello")

F. Something else

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

String Exercise

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`
E. None/other

The string library: `str(n)cat`

`strcat(dst, src)`: concatenates the contents of **src** into the string **dst**.

`strncat(dst, src, n)`: same, but concats at most n bytes from **src**.

```
char str1[13];          // enough space for strings + '\0'
strcpy(str1, "hello ");
strcat(str1, "world!");    // removes old '\0', adds new '\0' at end
printf("%s", str1);        // hello world!
```

Both **`strcat`** and **`strncat`** remove the old '\0' and add a new one at the end. (Note that we can't concatenate C strings using + as we can in C++ or Python.)

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'\0'	?	?	?	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	?	?	?	?	?	?

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Concatenating Strings

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Substrings and char *

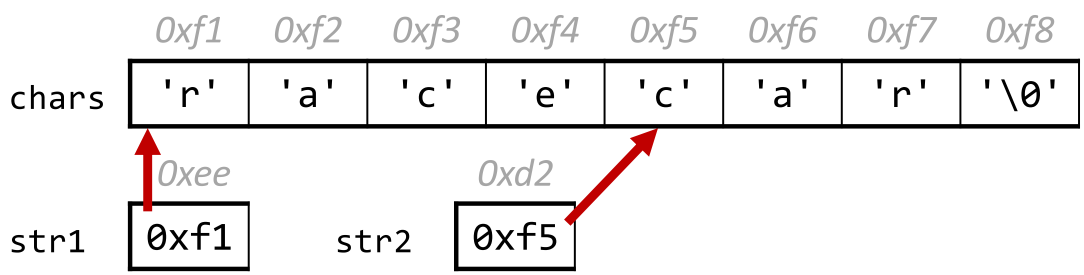
You can also create a char * variable yourself that points to an address within in an existing string.

```
char str[3];
str[0] = 'H';
str[1] = 'i';
str[2] = '\0';

char *alias = str; // points to 'H'
```

Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning.

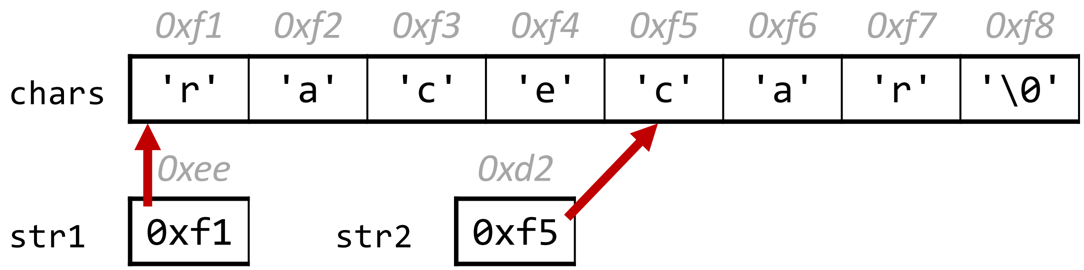
```
// Want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
```



Substrings

Since C strings are pointers to characters, we can adjust the pointer to omit characters at the beginning.

```
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4;
printf("%s\n", str1);        // racecar
printf("%s\n", str2);        // car
```

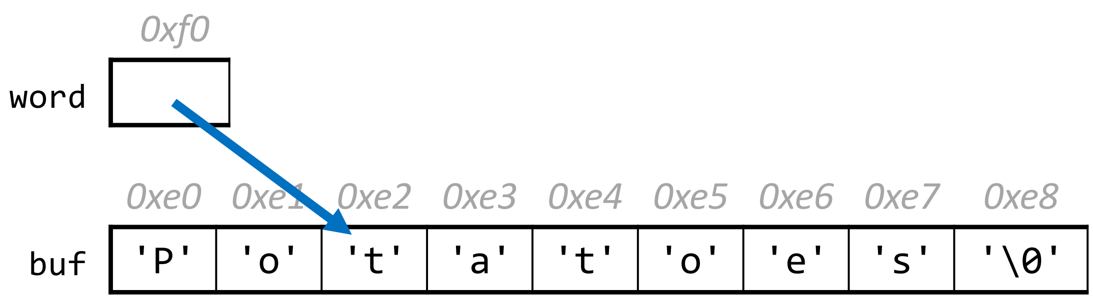


C String Etudes

```
char str[9];
strcpy(str, "potatoes");
char *word = str + 2;
strcpy(word, "mat");
printf("%s\n", str);
```

What is printed?

A. matoes
B. mattoes
C. pomat
D. pomatoes
E. pomitoes
F. pomidoes



C String Etudes

```c
char str[9];
strcpy(str, "potatoes");
char *word = str + 2;
strncpy(word, "mid", 2);
printf("%s\n", str);
```

What is printed?

A. matoes
B. mattoes
C. pomat
D. pomatoes
E. pomitoes
F. pomidoes