CS107, Lecture 6
C Strings

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

Ed Discussion

CS107 Topic 2: 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 (this time **and** 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, which serves as its "ASCII value".

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

```
char upper = 'A'; // actually 65
char lower = 'a'; // actually 97 (or rather, 65 + 32)
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!

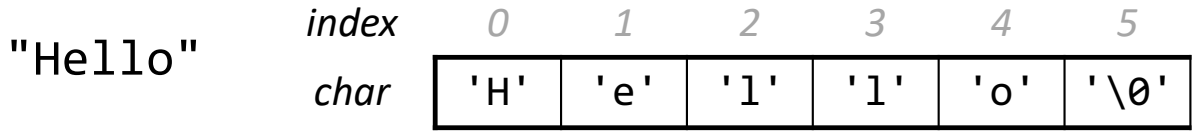
More documentation with `man isalpha`, `man tolower`

Common ctype.h Functions

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

C Strings

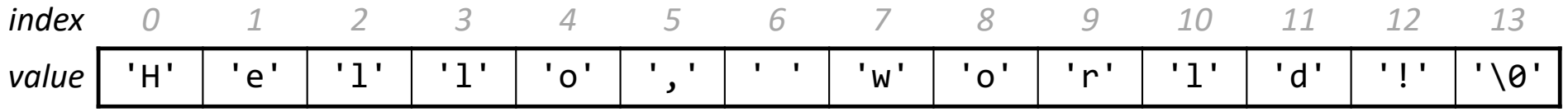
C has no dedicated variable type for strings. Instead, a string is represented as an **array of characters** with a sentinel value marking its end.



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

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 calculate it ourselves.



We typically call the built-in **`strlen`** function to calculate string length. The null-terminating character doesn't contribute to the 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(&string[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
}
```

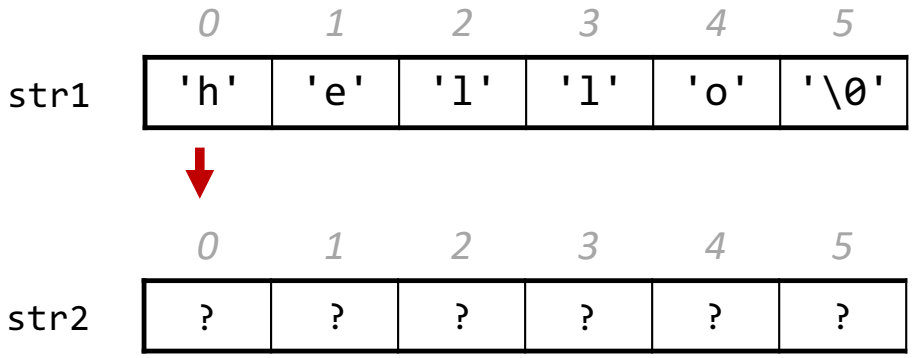
The string library: `strcpy`

`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`

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

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



Copying Strings - `strcpy`

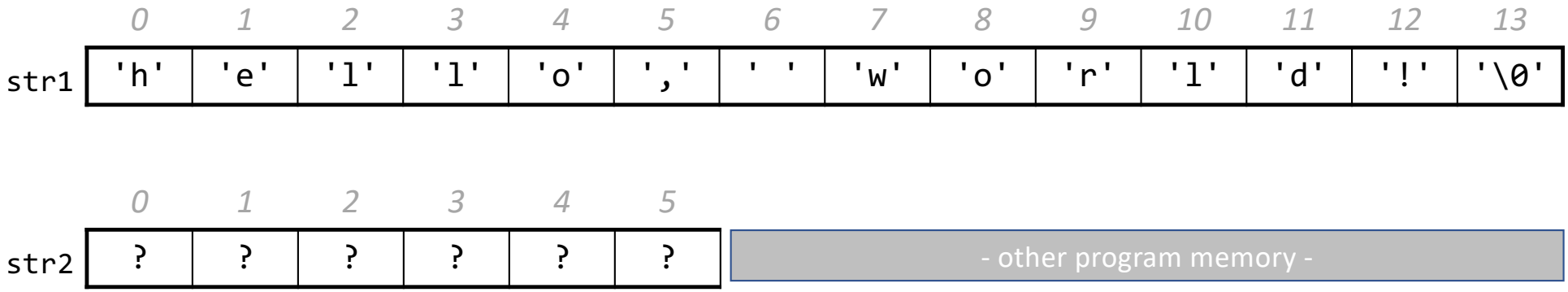
We must make sure there is enough space in the destination to hold the entire copy, *including the null-terminating character*.

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

Writing past memory bounds is called a **buffer overflow**.

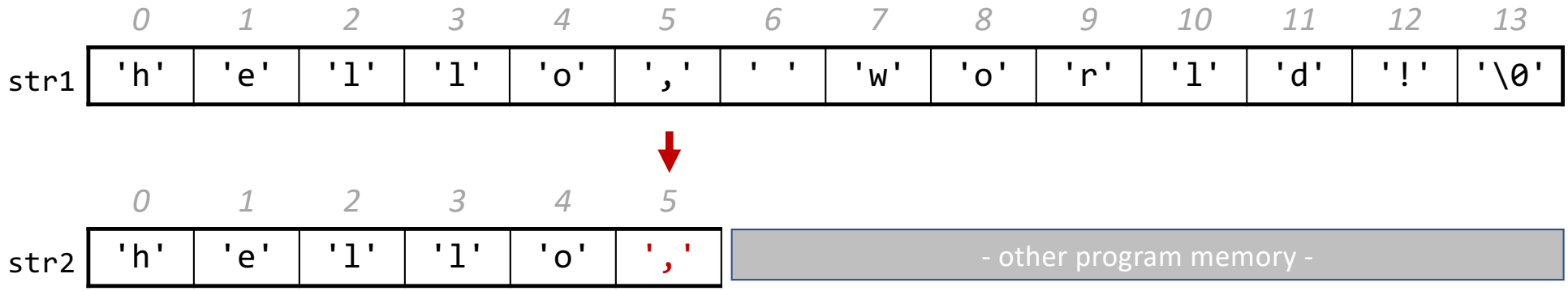
Copying Strings – Buffer Overflows

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



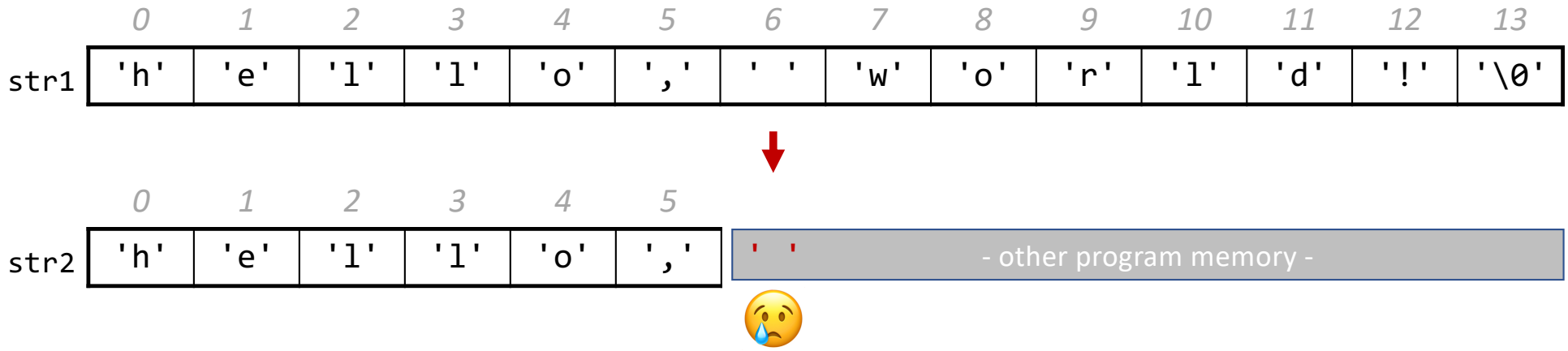
Copying Strings – Buffer Overflows

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



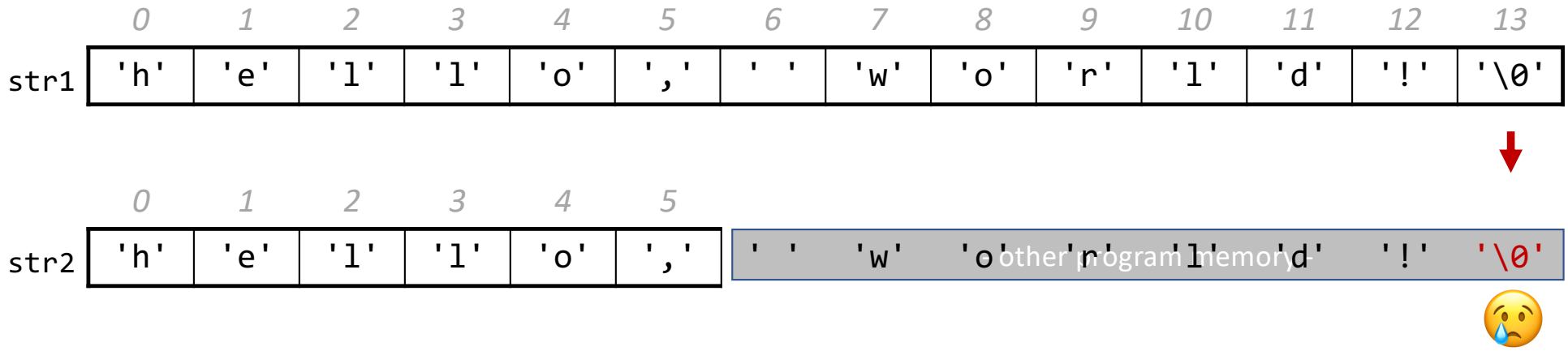
Copying Strings – Buffer Overflows

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



Copying Strings – Buffer Overflows

```
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"
char str[8];
strncpy(str, "automata", 8);      // 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**.

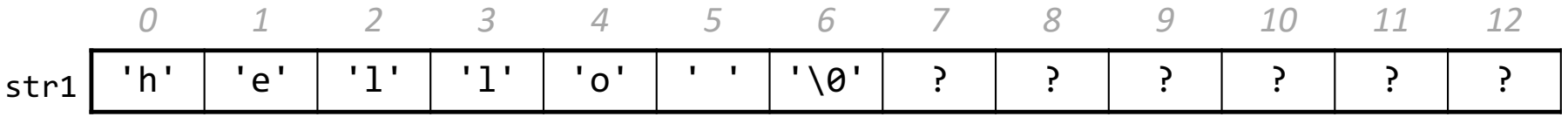
`strncat(dst, src, n)`: same, but concats at most n bytes from **src**.

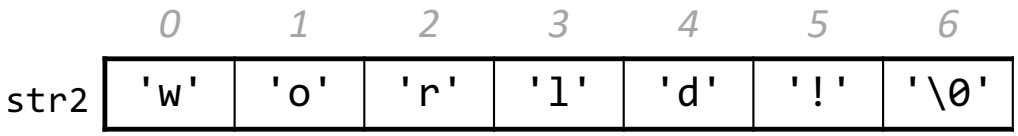
```
char str1[13];          // enough space for strings + '\0'
strcpy(str1, "hello ");
strcat(str1, "world!");     // removes old '\0', adds new '\0' at end
printf("%s", str1);         // hello world!
```

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

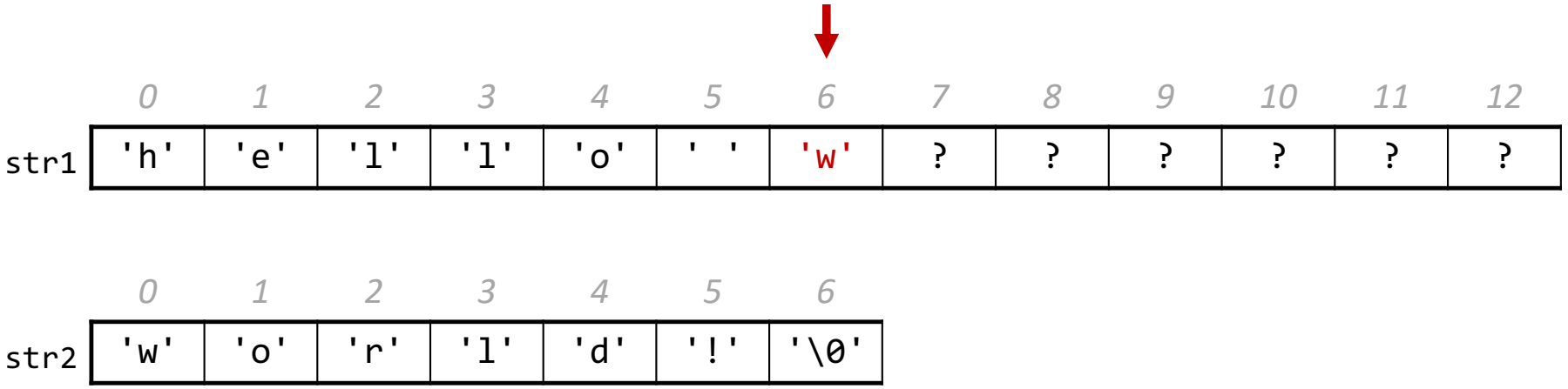
```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");

strcat(str1, str2);
```





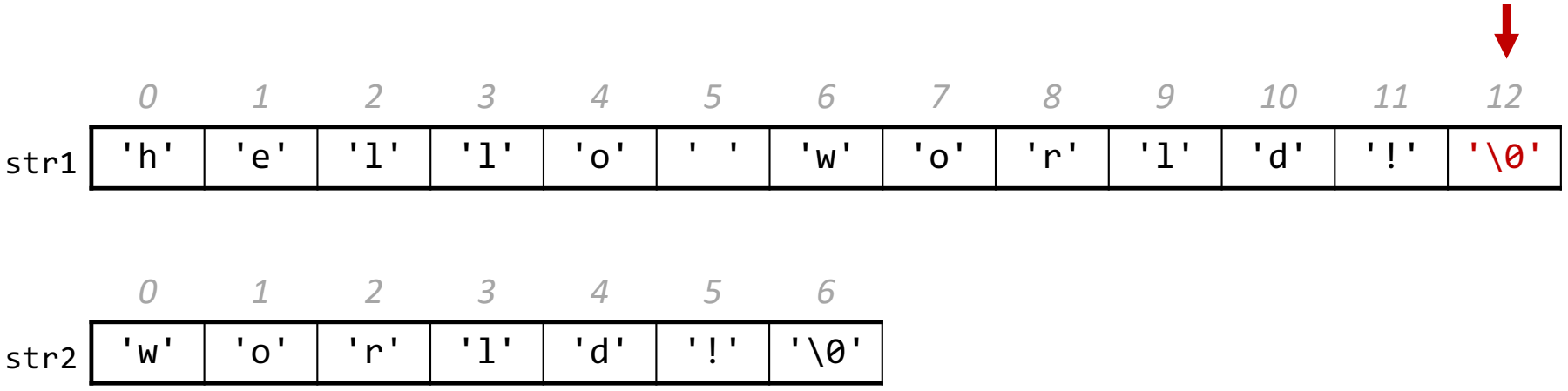
Concatenating Strings

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

strcat(str1, str2);
```



Concatenating Strings

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

strcat(str1, str2);
```



Substrings and char *

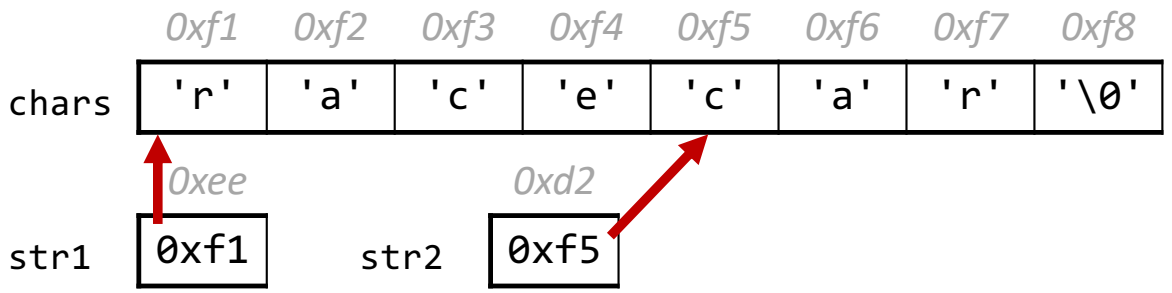
You can also create a char * variable yourself that points to an address within in an existing string.

```
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 overlook characters at the beginning.

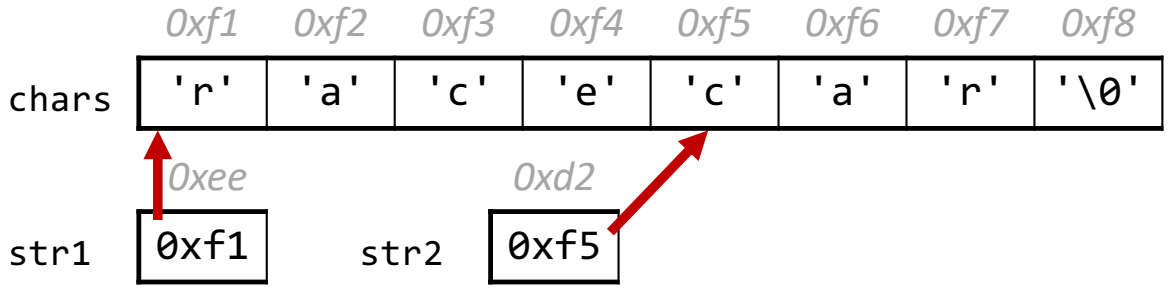
```
// want just "car"
char chars[8];
strcpy(chars, "racecar");
char *str1 = chars;
char *str2 = chars + 4; // equivalently, str2 = &chars[4]
```



Substrings

Since C strings are pointers to characters, we can adjust the pointer to overlook 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
```

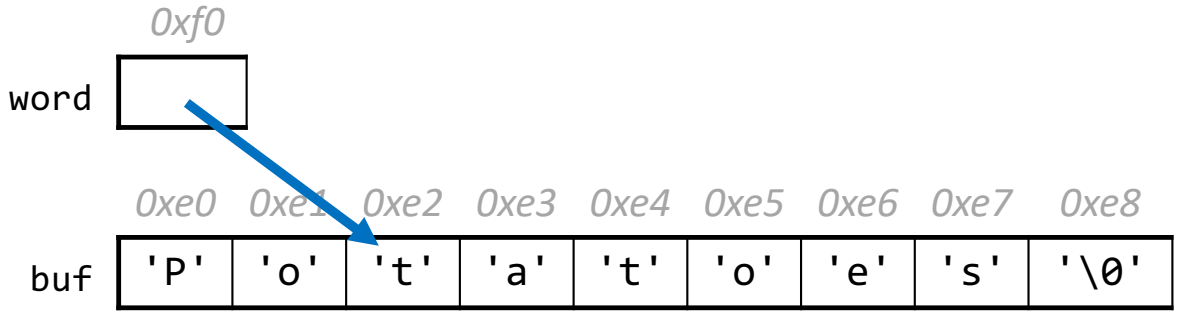


C String Etudes

```
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

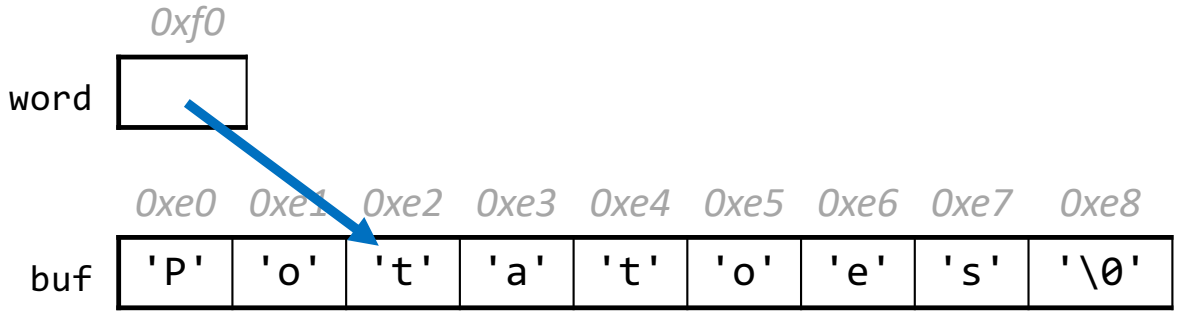


C String Etudes

```
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

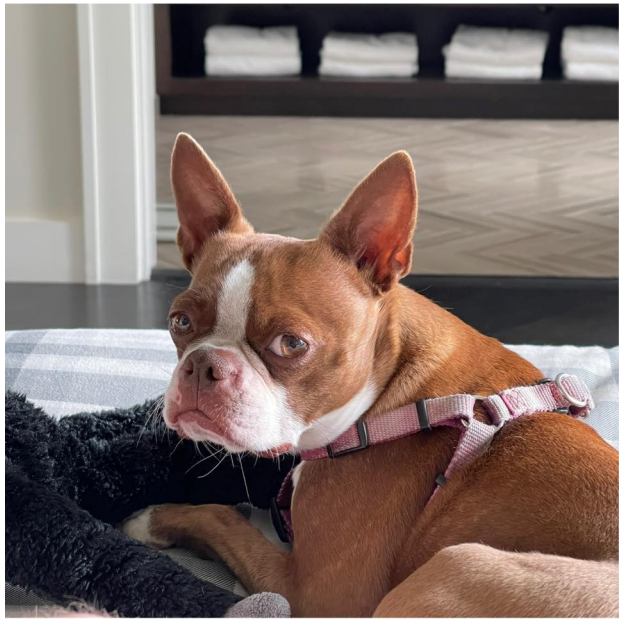


String Diamond

Write a function **`diamond`** that accepts a string 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
```



Practice: String Diamond

`string_diamond.c`

Searching For Letters

`strchr` returns a pointer to the first occurrence of a character in a string, or NULL if the character is not in the string.

```
char laureate[15];
strcpy(laureate, "Katalin Kariko");
char *first = strchr(laureate, 'a');
char *last = strrchr(laureate, 'a');
printf("%s\n", laureate);  // Katalin Kariko
printf("%s\n", first);     // atalin Kariko
printf("%s\n", last);      // ariko
```

If there are multiple occurrences of the letter, `strchr` returns a pointer to the *first* one. Use `strrchr` to obtain a pointer to the *last* occurrence.

Searching For Strings

`strstr` returns a pointer to the first occurrence of the second string in the first, or NULL if it cannot be found.

```
char laureate[17];
strcpy(laureate, "Carolyn Bertozzi");
char *zz = strstr(laureate, "zz");
printf("%s\n", laureate);          // Carolyn Bertozzi
printf("%s\n", zz);                // zzi
```

If there are multiple occurrences of the string, `strstr` returns a pointer to the *first* one.

String Spans

`strspn` returns the *length* of the initial part of the first string which contains only characters in the second string.

```
char laureate[17];
strcpy(laureate, "Barry Sharpless");
int length = strspn(laureate + 1, "road"); // 3
```

"How many places can we go in the first string before I encounter a character not in the second string?"

String Spans

`strcspn` (c = "complement") returns the *length* of the initial part of the first string which contains only characters not in the second string.

```
char laureate[17];
strcpy(laureate, "Barry Sharpless");
int length = strcspn(laureate + 2, "abcde");    // 6
```

"How many places can we go in the first string before I encounter a character in the second string?"

C Strings As Parameters

When we pass a string as a parameter, it is passed as a **char *** . We can still operate on the string the same way as with a char[].

```
int foo(char *str) {
     char ch = str[1];
     ...
}

// can also write this, but it is really a pointer
int foo(char str[]) { ...
```

Arrays of Strings

We can make an array of strings to group multiple strings together:

```
char *array[5];    // space to store 5 char *s
```

We can also use the following shorthand to initialize a string array:

```
char *array[] = {
     "Hello",
     "Hi",
     "Hey there"
};
```

Arrays of Strings

We can access each string using bracket syntax:

```
printf("%s\n", array[0]);   // print out first string
```

When an array is passed as a parameter in C, C passes a *pointer to the array's first element*. In fact, you're already seen this with **main**'s **argv** parameter! This means we write the parameter type as:

```
void func(char **array) {

// equivalent to this, but it is really a double pointer
void func(char *array[]) {
```

Practice: Password Verification

Write a function **`verifyPassword`** that accepts a candidate password and certain password criteria and returns whether the password is valid.

```
bool verifyPassword(char *password, char *validChars,
                    char *badSubstrings[], size_t count);
```

password is valid if it contains only letters in **validChars** and does not contain any substrings in **badSubstrings**.

Practice: Password Verification

```
bool verifyPassword(char *password, char *validChars,
                    char *badSubstrings[], size_t count);
```

Example:

```
char *invalidSubstrings[] = {"1234" , "4132"};

bool valid1 = verifyPassword("1572", "0123456789",
              invalidSubstrings, 2);  // true
bool valid2 = verifyPassword("141234", "0123456789",
              invalidSubstrings, 2);  // false
```

Practice: Password Verification

`verify_password.c`