CS107 Winter 2020, Lecture 4
C Strings

reading:

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

Slides by Jerry Cain and Lisa Yan, who leveraged prior work by Nick Troccoli, Julie Zelenski, Marty Stepp, Cynthia Lee, Chris Gregg, and others.

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

Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

Warmup exercise

What do the following 8-bit binary numbers represent? (select all that apply)

`10010101`

A. Unsigned integer, 149
B. Two's complement signed integer, -21
C. Two's complement signed integer, -107
D. C `char` variable
E. None/other

`01100011`

A. Unsigned integer, 99
B. Two's complement signed integer, 99
C. C `char` variable, `'c'`
D. None/other

Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

Char

A **`char`** is a 1-byte variable type that represents a single character or “glyph”.

```
char letterA = 'A';
char plus = '+';
char zero = '0';
char space = ' ';
char newLine = '\n';
char tab = '\t';
char singleQuote = '\'';
char backSlash = '\\';
```

ASCII

Under the hood, C represents **`char`** as an 8-bit unsigned *integer* ("ASCII value").

- Uppercase/lowercase letters, digits are sequentially numbered.
- Lowercase letters are 32 more than their uppercase equivalents.

```
char uppercaseA = 'A';              // Actually 65
char lowercaseA = 'a';              // Actually 97
char zeroDigit = '0';               // Actually 48

// prints out every lowercase character
for (char ch = 'a'; ch <= 'z'; ch++) {
      printf("%c", ch);
}
```

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`

Code study: `ctype.h` implementations

```
int my_isdigit(int ch) {
    return ch >= '0' && ch <= '9';
}
```

Small note: `bool` (e.g., `true` and `false`) is not a built-in C type. Import `stdbool.h` if you want to use `bool`s.

Standard ASCII maps 0x00 - 0x7f to letters, digits, and punctuation

- `man ascii` to display table
- 8th bit used as parity/error check in some situations
- No consensus for characters mapped to 0x80-0xff

Code study: ctype.h implementations

```
int my_isdigit(int ch) {
    return ch >= '0' && ch <= '9';
}
```

Code study: ctype_code

```
cp -r /afs/ir/class/cs107/samples/lectures/lect4 .
```

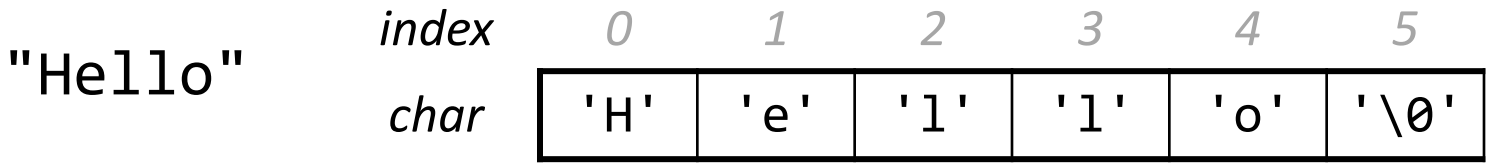
Contains a bonus example. Test your bitwise/ASCII skills!

Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

C Strings

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



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

C Strings, initialization

3 equivalent ways to create a string in local memory (on the stack):

"Hello"

'H'	'e'	'l'	'l'	'o'	'\0'

1.
```
char str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

2.
```
char str[6];
str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '\0';
```

Create an **empty** 6-character array, then set each element.

3.
```
char str[] = "Hello";
```

Shorthand: Populate array, **auto-add null terminator**

Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

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` finds 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 (might not include null-terminating character).
`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`***.

Common `string.h` Functions

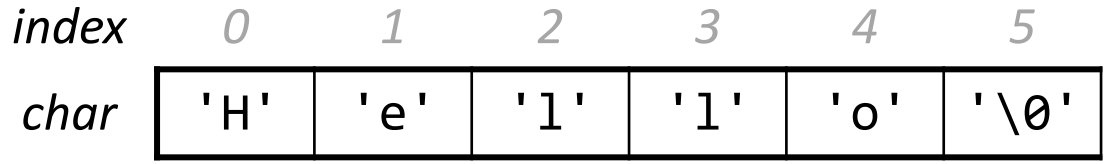
Function	Description
`strlen(str)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(str1, str2)`, `strncmp(str1, str2, n)`	compares two strings; returns 0 if identical, <0 if ***str1*** comes before ***str2*** in alphabet, >0 if ***str1*** comes after ***str2*** in alphabet. ***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` finds 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 (might not include null-terminating character).
`strcat(dst, src)`, `strncat(dst, src, n)`	concatenate ***src*** onto the end of ***dst***. **strncat** stops concatenating after at most ***n*** characters. Always null-terminating character.
`strspn(str, accept)`, `strcspn(str, reject)`	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***.

(we'll cover these this lecture)

The string library: `strlen`

`strlen`: returns the # of `chars` in a C string.

```
char str[] = "Hello";
int length = strlen(str); // 5
```



- The null-terminating character `'\0'` does ***not*** count towards the length.
- `strlen` is O(N) because it counts the characters until the null terminator.

Important: Strings are **not** objects.
- They do not embed additional information (e.g., string length).
- Many string functions assume **valid string** input; i.e., ends in a null terminator.

String Length

What is printed out by the following program?

```
1 int main(int argc, char *argv[]) {
2     char str[] = "Hi earth";
3     str[2] = '\0';
4     printf("str = %s, len = %ld\n",
5            str, strlen(str));
6     return 0;
7 }
```

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

Key takeaway #1

1. Valid strings are null-terminated.

The string library: `strcmp`

`strcmp(str1, str2)`: compares two strings.

- returns 0 if identical
- <0 if ***`str1`*** comes before ***`str2`*** in alphabet
- >0 if ***`str1`*** comes after ***`str2`*** in alphabet.

Demo: strcmp_ex

```
cp -r /afs/ir/class/cs107/samples/lectures/lect4 .
```

Questions with `strcmp`

```
1  char str1[] = "Lettuce";
2  char str2[] = "Cabbage";
3  int cmp_result = strcmp(str1, str2);
4
5  printf("%s", str1);
6  if (cmp_result == 0) {
7      printf(" is the same as ");
8  } else if (cmp_result < 0) {
9      printf(" comes before ");
10 } else {
11     printf(" comes after ");
12 }
13 printf("%s\n", str2);
14 printf("int result of strcmp: %d\n", cmp_result);
```

- Line 3: What is the return value of `strcmp()`?
- Lines 3, 6, 8: **Why can't we just use ==, >, < to compare strings?**

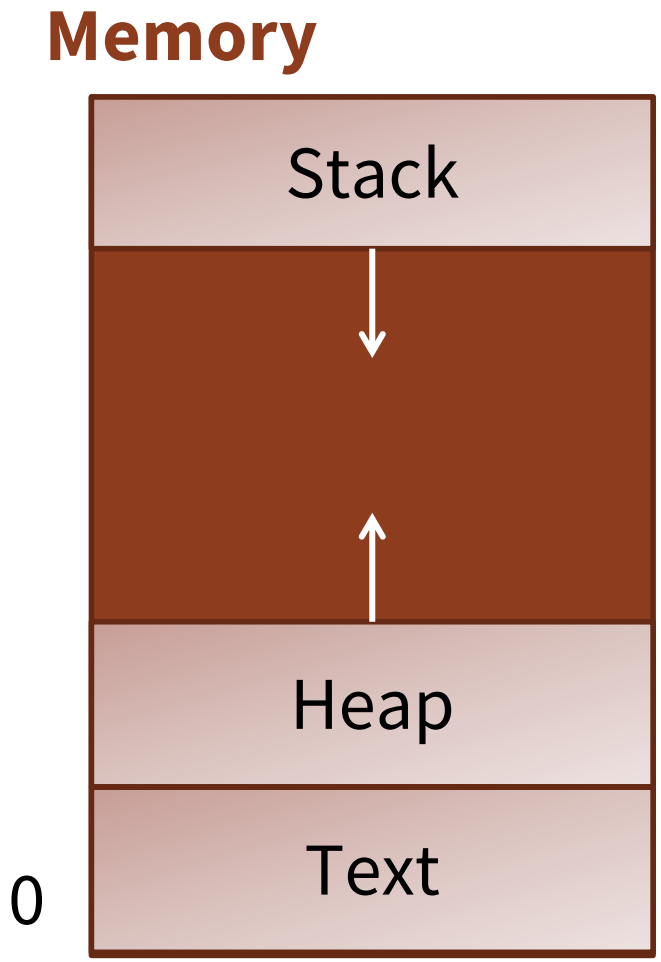
Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

Key takeaway #2

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the **address** of the first element.

Pointers and memory, a refresher



* Take CS107 to learn much more!!

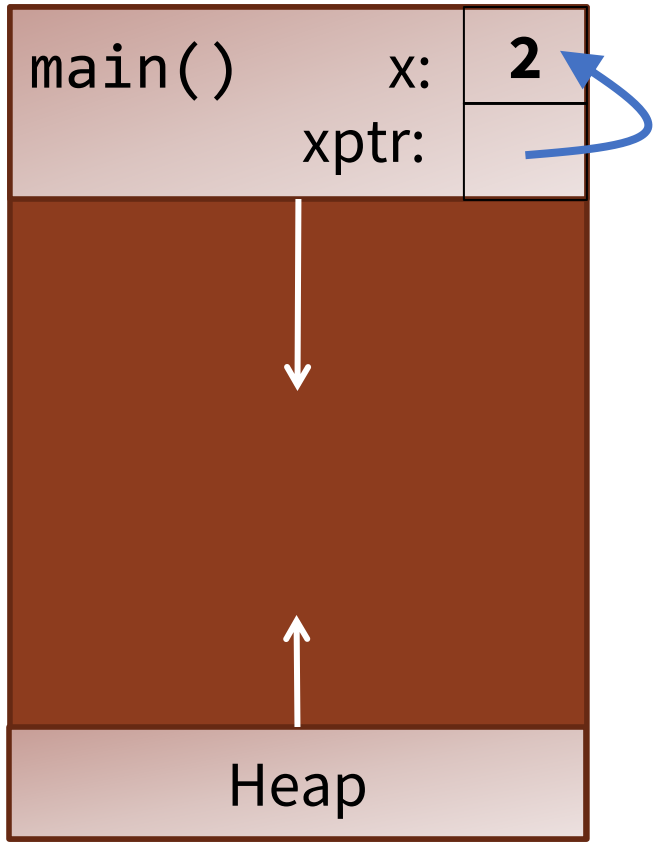
** We are currently taking CS107

Pointers and memory, a refresher

A **pointer** is a variable that stores a memory address.

```
1  int main(int argc, char *argv[]) {
2      int x = 2;
3      int *xptr = &x;
4      *xptr = 3;
5      printf("%d\n", x);
6      printf("%d\n", *xptr);
7      printf("%p\n", xptr);
8      return 0;
9  }
```

& op: “address of”

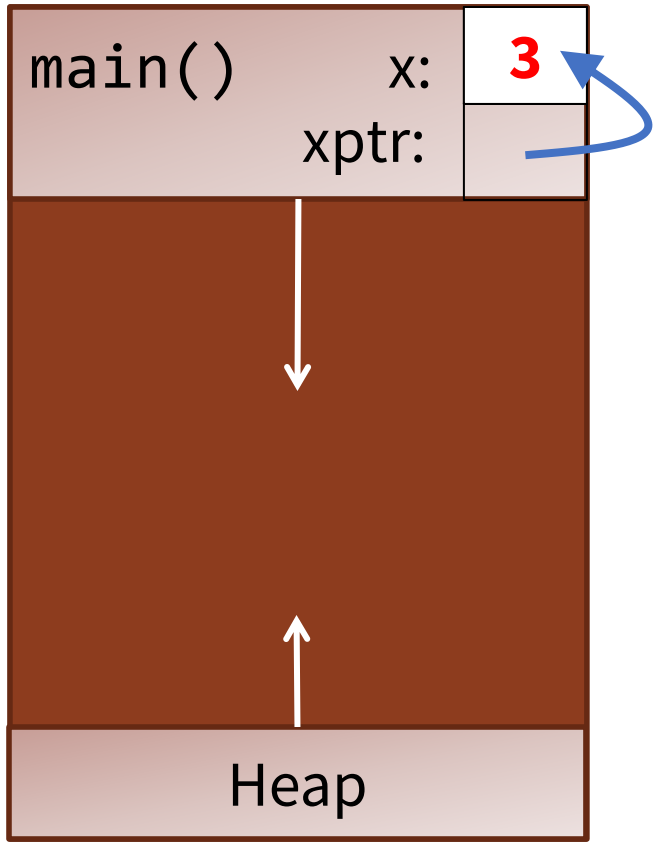


Pointers and memory, a refresher

A **pointer** is a variable that stores a memory address.

```
1  int main(int argc, char *argv[]) {
2      int x = 2;
3      int *xptr = &x;
4      *xptr = 3;
5      printf("%d\n", x);       // 3
6      printf("%d\n", *xptr); // 3
7      printf("%p\n", xptr);
8      return 0;
9  }
```

& op: “address of”
* op: “dereference”

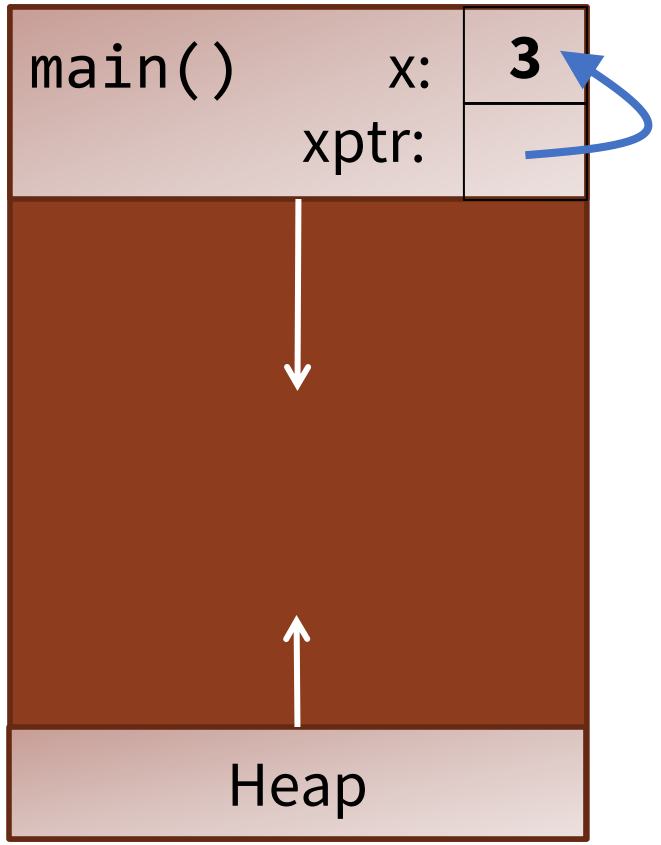


Pointers and memory, a refresher

A **pointer** is a variable that stores a memory address.

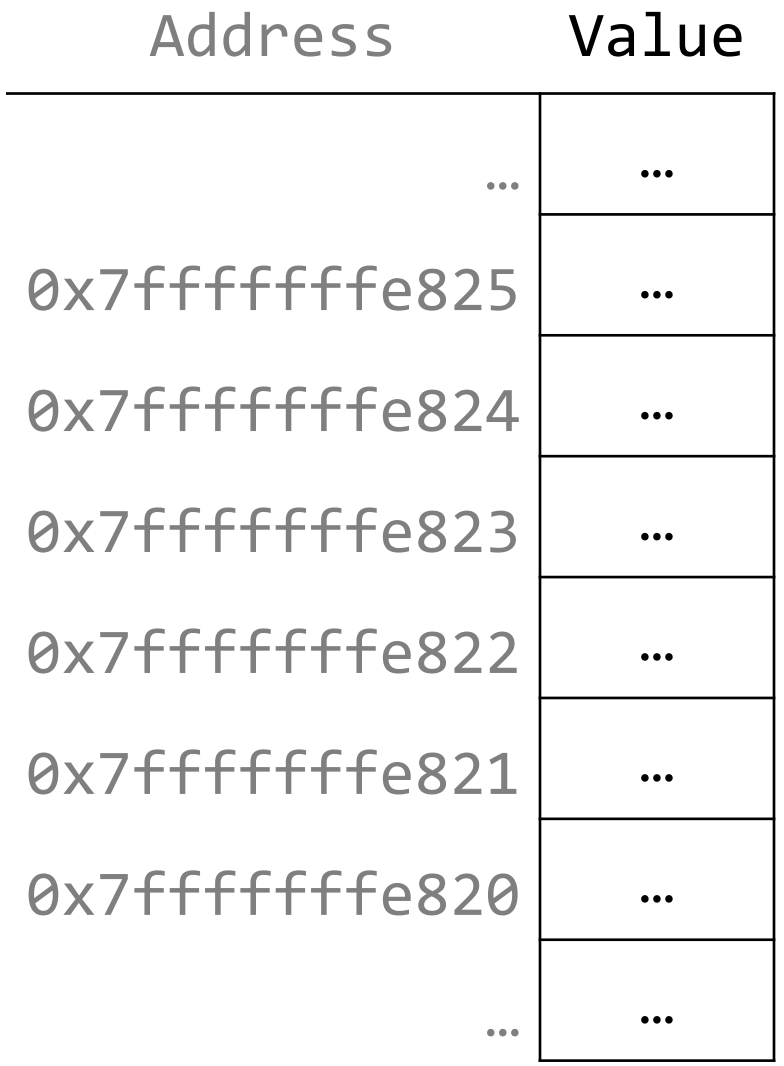
```
1 int main(int argc, char *argv[]) {
2     int x = 2;
3     int *xptr = &x;
4     *xptr = 3;
5     printf("%d\n", x);      // 3
6     printf("%d\n", *xptr);  // 3
7     printf("%p\n", xptr);   // 0x7fffffffe820
8     return 0;
9 }
```

"%p": Format as address



Memory, in detail

- Memory is a big array of bytes.
- Each byte has a memory address that is commonly written in hexadecimal.
- Local variables are created and pushed onto the **stack**, which is a part of memory.
- A **pointer** is a variable that stores a memory address.
- On the myth machines, all pointers are 64-bits, or **8 bytes** long.

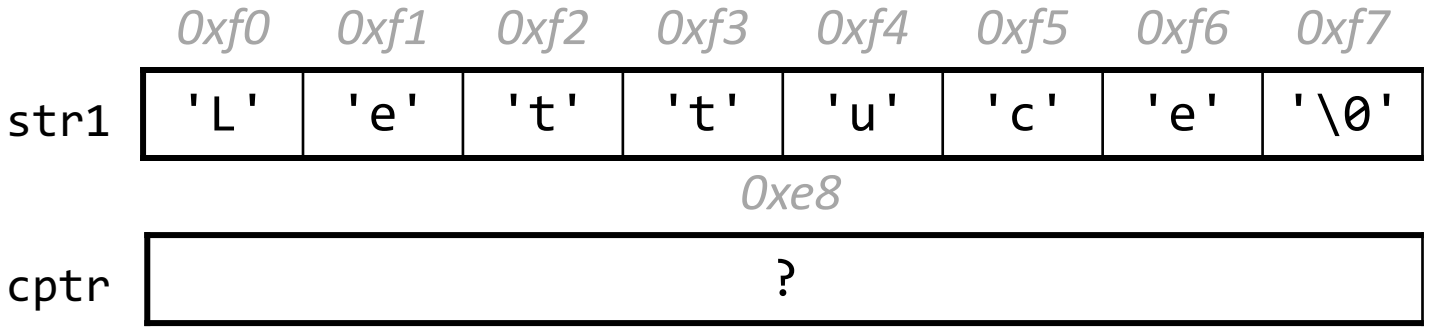


char * and char[]

- An array name (and a string name, by extension) is the address of the first element.

```
1 char str1[] = "Lettuce";
2 printf("%c", *str1);
3
4 char *cptr = str1;
5 printf("%s\n", cptr);
6 printf("%s\n", str1);
7 printf("%c\n", *cptr);
```

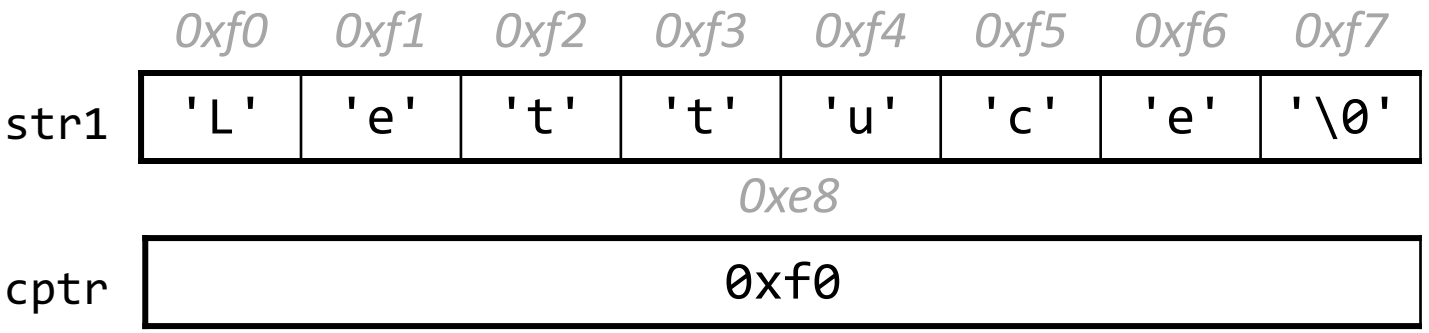
- Line 2: What is printed?
- Line 4: What is stored in ptr?
- Lines 5-7: What is printed?



char * and char[]

- An array name (and a string name, by extension) is the address of the first element.

```
1 char str1[] = "Lettuce";
2 printf("%c", *str1);          // L
3
4 char *cptr = str1;
5 printf("%s\n", cptr);         // Lettuce
6 printf("%s\n", str1);         // Lettuce
7 printf("%c\n", *cptr);        // L
```



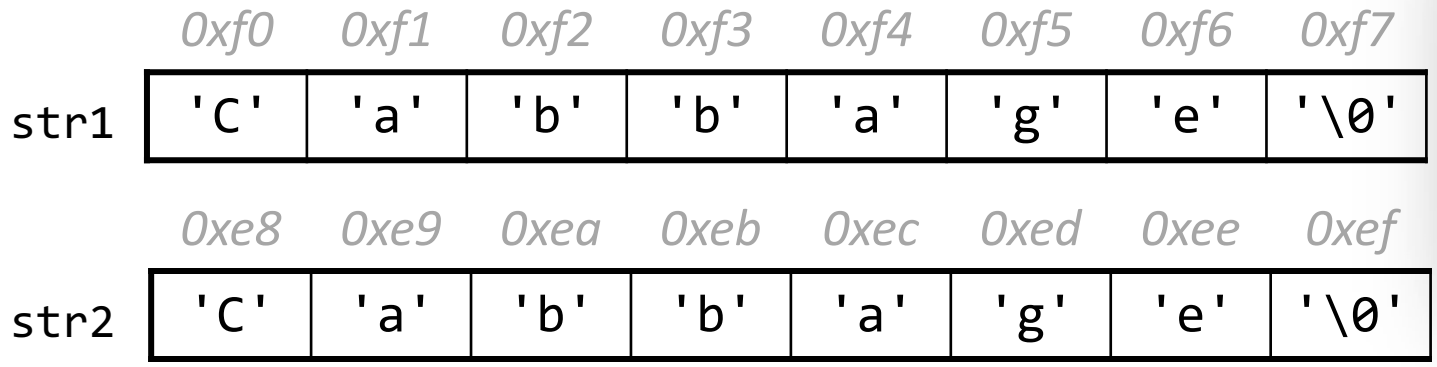
Comparing pointers

- An array name (and a string name, by extension) is the address of the first element.
- As a result, it is meaningless in C to compare strings with relational operators:

```
char str1[] = "Cabbage";
char str2[] = "Cabbage";

if (str1 > str2) {
        …
}
```

C interprets as: `compare 0xf0 > 0xe8`



Operations on arrays tend to be operations on addresses. Use string library functions when you can!

Answers with `strcmp`

```
1  char str1[] = "Lettuce";
2  char str2[] = "Cabbage";
3  int cmp_result = strcmp(str1, str2);
4
5  printf("%s", str1);
6  if (cmp_result == 0) {
7      printf(" is the same as ");
8  } else if (cmp_result < 0) {
9      printf(" comes before ");
10 } else {
11     printf(" comes after ");
12 }
13 printf("%s\n", str2);
14 printf("int result of strcmp: %d\n", cmp_result);
```

- Line 3: What is the return value of `strcmp()`?
- Lines 3, 6, 8: **Why can't we just use ==, >, < to compare strings?**

Remaining questions:

- Is char* equal to char[]?

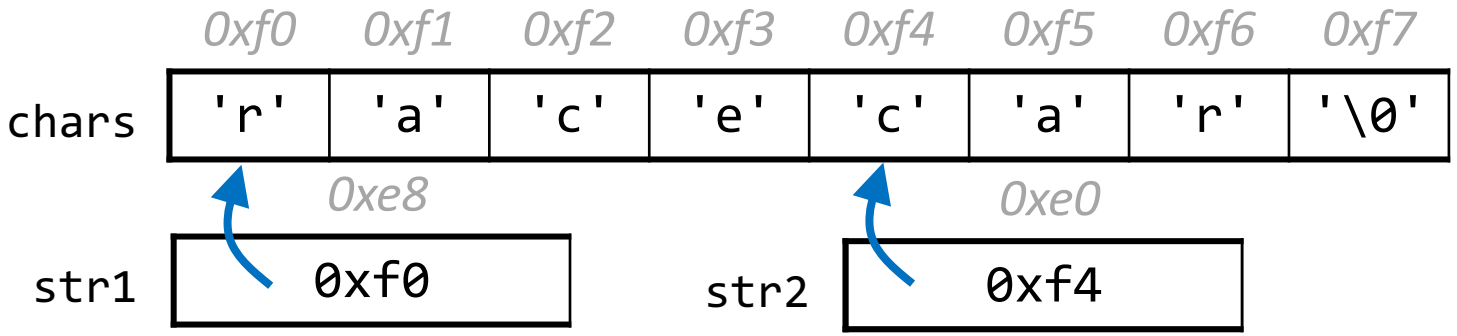
(full answer next lecture)

char * and substrings

Because char *s are pointers to characters, we can use them to create substrings of larger strings!

- Key: Incrementing a char * by 1 increases the memory address by 1 byte.

```
1 char chars[] = "racecar";
2 char *str1 = chars;
3 char *str2 = chars + 4;
4 str2[0] = 'f';
5 printf("%s, %s\n", chars, str1);
6 printf("%s\n", str2);
```

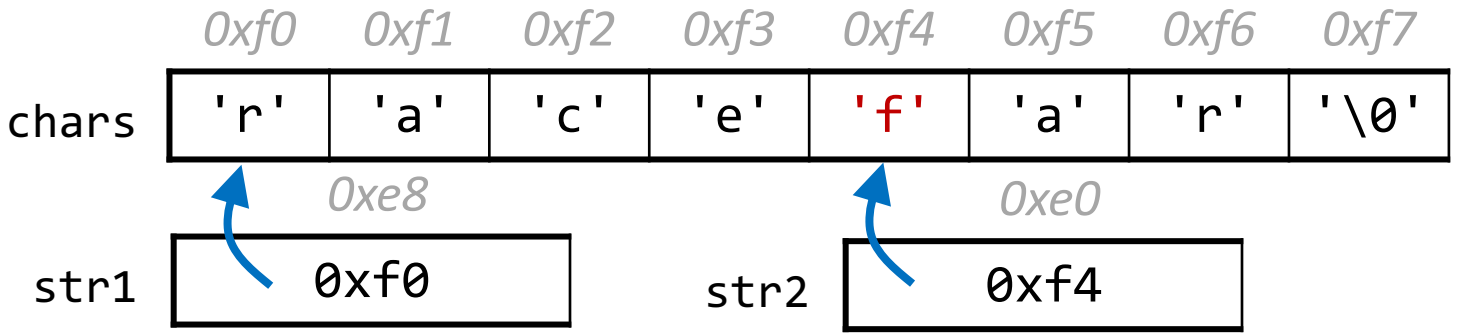


char * and substrings

Because char *s are pointers to characters, we can use them to create substrings of larger strings!

- Key: Incrementing a char * by 1 increases the memory address by 1 byte.

```
1 char chars[] = "racecar";
2 char *str1 = chars;
3 char *str2 = chars + 4;
4 str2[0] = 'f';
5 printf("%s, %s\n", chars, str1);
6 printf("%s\n", str2);
```

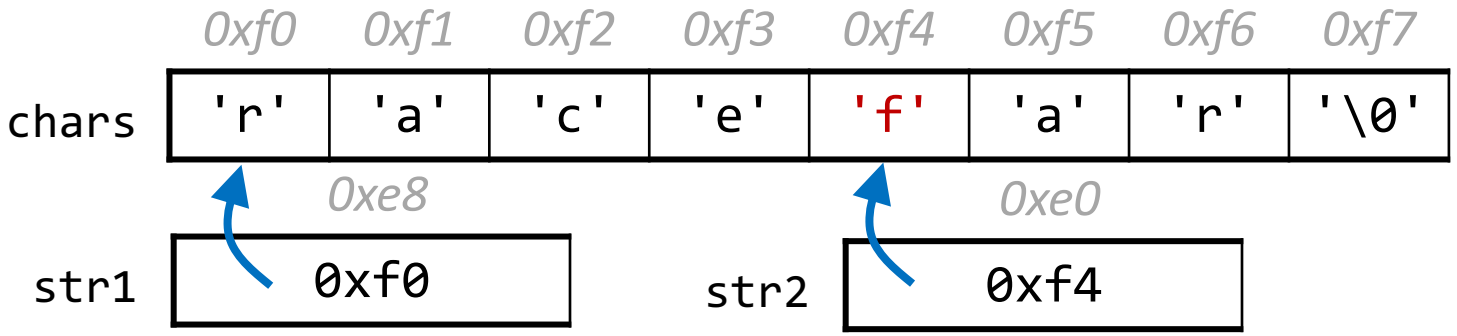


char * and substrings

Because char *s are pointers to characters, we can use them to create substrings of larger strings!

- Key: Incrementing a char * by 1 increases the memory address by 1 byte.

```
1 char chars[] = "racecar";
2 char *str1 = chars;
3 char *str2 = chars + 4;
4 str2[0] = 'f';
5 printf("%s, %s\n", chars, str1); // racefar, racefar
6 printf("%s\n", str2);            // far
```



Pointer arithmetic lets you access substrings.

Questions?

Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

Announcements

- GDB Config File (cs107.stanford.edu/resources/gdb)
- emacs/vim config files also available
- Piazza is an official channel for course communication this quarter
- We hope you enjoyed your first lab!

No class on Monday, 1/20 (Martin Luther King Jr. Day)

- Video lecture posted over the weekend

Joke break

"Hello, I'd like to hear a C joke"

```
strcpy(joke, chicken, strlen(chicken));
printf("%s", joke);
```

```
Segmentation fault (core dumped)
```

Note:
This code also just doesn’t compile

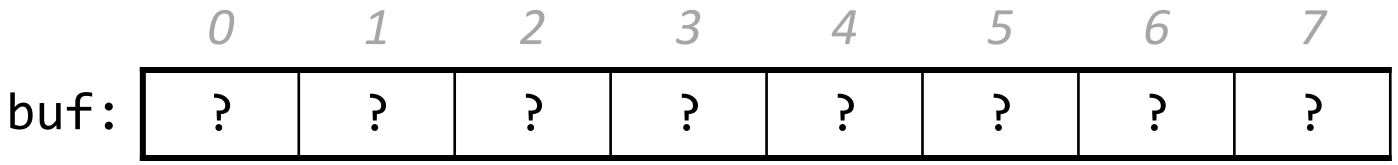
Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

The string library: `strcpy`

`strcpy(dst, src)`: copies characters in ***`src`*** to ***`dst`***, including null terminator.

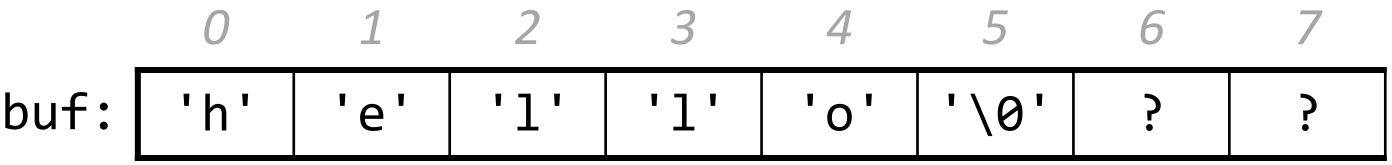
```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);
```



The string library: `strcpy`

`strcpy(dst, src)`: copies characters in ***`src`*** to ***`dst`***, including null terminator.

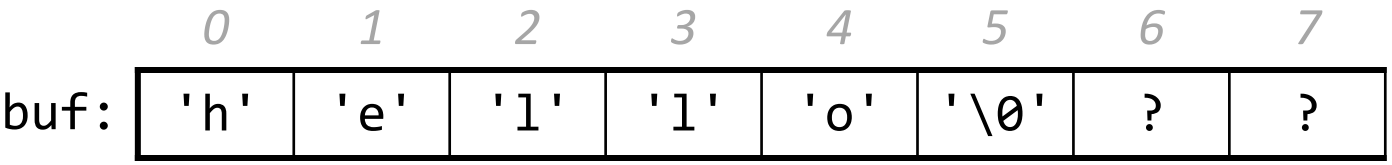
```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);
```



The string library: `strcpy`

`strcpy(dst, src)`: copies characters in ***`src`*** to ***`dst`***, including null terminator.

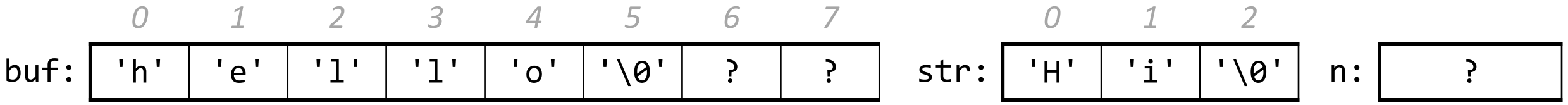
```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);          // hello
```



The string library: strncpy

`strncpy(dst, src, n)`: copies at most ***n*** characters in ***src*** to ***dst***.

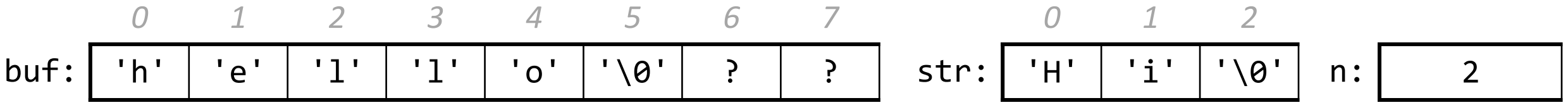
```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);          // hello
4 char str[] = "Hi";
5 int n = strlen(str);
6 strncpy(buf, str, n);
7 buf[n] = '\0';
8 printf("%s\n", buf);
```



The string library: strncpy

`strncpy(dst, src, n)`: copies at most ***n*** characters in ***src*** to ***dst***.

```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);          // hello
4 char str[] = "Hi";
5 int n = strlen(str);
6 strncpy(buf, str, n);
7 buf[n] = '\0';
8 printf("%s\n", buf);
```

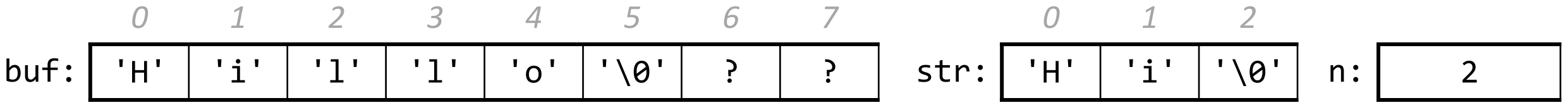


The string library: strncpy

`strncpy(dst, src, n)`: copies at most ***n*** characters in ***`src`*** to ***`dst`***.

```
1  char buf[8];
2  strcpy(buf, "hello");
3  printf("%s\n", buf);          // hello
4  char str[] = "Hi";
5  int n = strlen(str);
6  strncpy(buf, str, n);
7  buf[n] = '\0';
8  printf("%s\n", buf);
```

⚠ `strncpy()` does not automatically put a null terminator at the end of the copy.

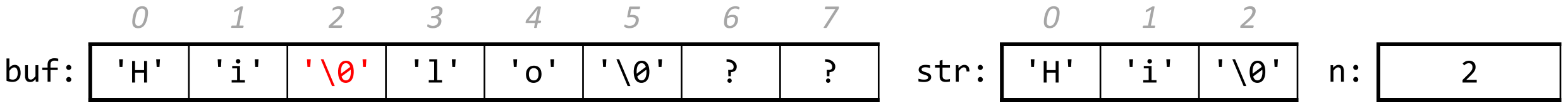


The string library: strncpy

`strncpy(dst, src, n)`: copies at most ***n*** characters in ***`src`*** to ***`dst`***.

```
1  char buf[8];
2  strcpy(buf, "hello");
3  printf("%s\n", buf);          // hello
4  char str[] = "Hi";
5  int n = strlen(str);
6  strncpy(buf, str, n);
7  buf[n] = '\0';
8  printf("%s\n", buf);
```

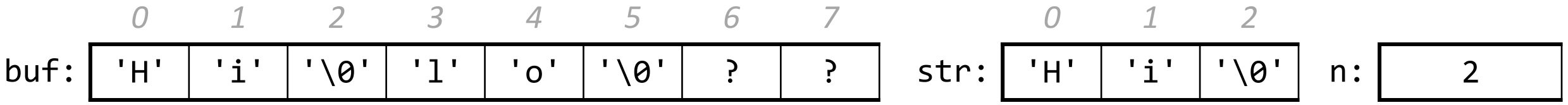
⚠ `strncpy()` does not automatically put a null terminator at the end of the copy.



The string library: strncpy

`strncpy(dst, src, n)`: copies at most ***n*** characters in ***src*** to ***dst***.

```
1 char buf[8];
2 strcpy(buf, "hello");
3 printf("%s\n", buf);          // hello
4 char str[] = "Hi";
5 int n = strlen(str);
6 strncpy(buf, str, n);
7 buf[n] = '\0';
8 printf("%s\n", buf);          // Hi
```



String copying exercise

```
1 char buf[ ____ ];
2 strcpy(buf, "Chadwick");
3 printf("%s\n", buf);
4 char *word = buf + 2;
5 strncpy(word, "ris", 3);
6 printf("%s\n", buf);
```

Line 1: What value should go in the blank?

A. `7`
B. `8`
C. `9`
D. `12`
E. `strlen("Chadwick")`
F. Something else

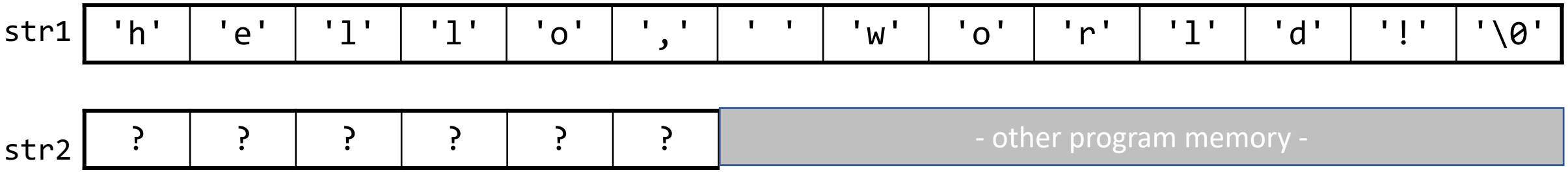
Line 6: What is printed?

A. `risick`
B. `risdwick`
C. `Chris`
D. `Chrisick`
E. Something else
F. Compile error

Pitfalls with `strcpy`: Buffer overflow

⚠ `strcpy(dst, src)` does not check if the destination is large enough:

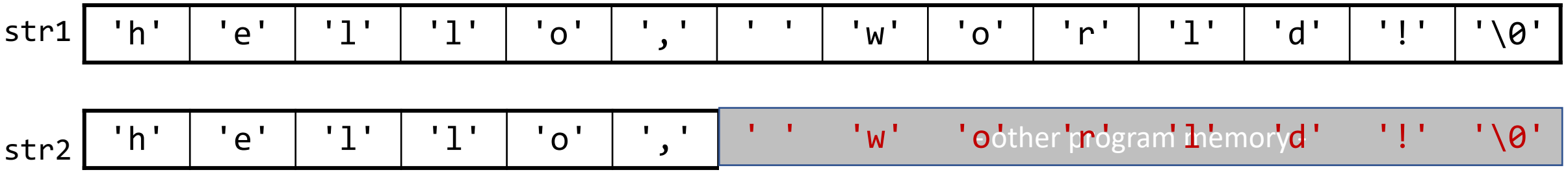
```
char str1[14];
strcpy(str1, "hello, world!");
char str2[6];                    // not enough space
strcpy(str2, str1);              // overwrites other memory!
```



Pitfalls with `strcpy`: Buffer overflow

⚠ `strcpy(dst, src)` does not check if the destination is large enough:

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



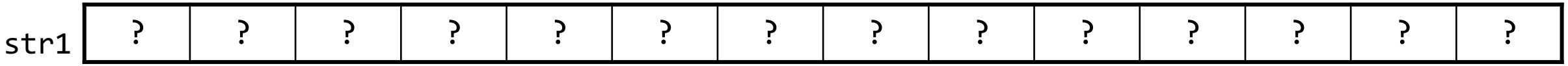
Buffer overflow: `strcpy(dst, src)` writes past the available space in `dst`.
This is a major security flaw which hackers exploit all the time!

Pitfalls with `strncpy`

✅ `strncpy(dst, src, n)`: always writes up to ***n*** bytes; as the programmer, you can avoid buffer overflow.

⚠️ However, if the resulting string is not null-terminated, you could *read* past the end of the `dst` buffer!

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

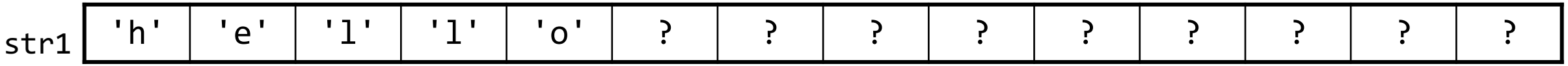


Pitfalls with `strncpy`

✅ `strncpy(dst, src, n)`: always writes up to ***n*** bytes; as the programmer, you can avoid buffer overflow.

⚠️ However, if the resulting string is not null-terminated, you could *read* past the end of the `dst` buffer!

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```

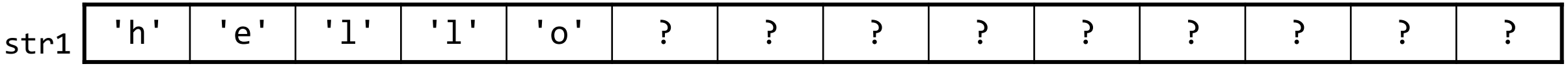


Pitfalls with `strncpy`

✅ `strncpy(dst, src, n)`: always writes up to ***n*** bytes; as the programmer, you can avoid buffer overflow.

⚠️ However, if the resulting string is not null-terminated, you could *read* past the end of the `dst` buffer!

```
char str1[14];
strncpy(str1, "hello there", 5);
printf("%s\n", str1);
```




```
hello?J???
```

Questions?

Plan For Today

- Characters
- Strings
- Common string operations: String length and comparing strings
- Strings, memory, and pointers, part 1
- **Break**: Announcements
- Common string operations: Copying strings
- Strings as function parameters

What's the deal?

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.

Why did C bother with this representation?

C is a powerful, efficient language that requires a solid understanding of computer memory.

Over the next two weeks, we will hone this understanding with lots of practice!

👉 Key takeaway #3

1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.
3. All parameters in C are “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.

Pass by value

- When you pass a value as a parameter, C passes a **copy** of that value.

What gets printed out by the following program?

```
void foo(int a) {
      a++;
      printf("%d\n", a);    // 3
}

int main(int argc, char *argv[]) {
      int x = 2;
      foo(x);
      printf("%d\n", x);    // *2*
      return 0;
}
```

"Pass by reference" with pointers

- When you pass a value as a parameter, C passes a copy of that value.

To modify the original variable, use a **pointer** to that variable.

```
void foo(int *a) {
      *a = *a + 1;            // Equivalent to (*a)++
      printf("%d\n", *a);     // 3
}

int main(int argc, char *argv[]) {
      int x = 2;
      foo(&x);
      printf("%d\n", x);      // *3*
      return 0;
}
```

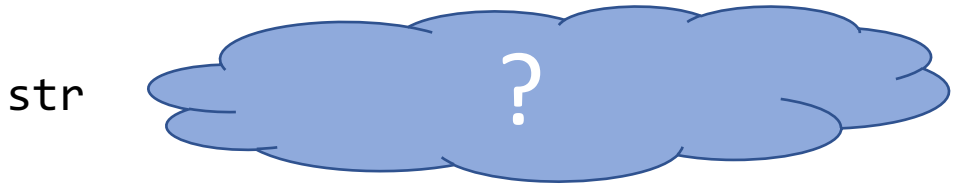
All C parameters (even pointers) are passed in by value.

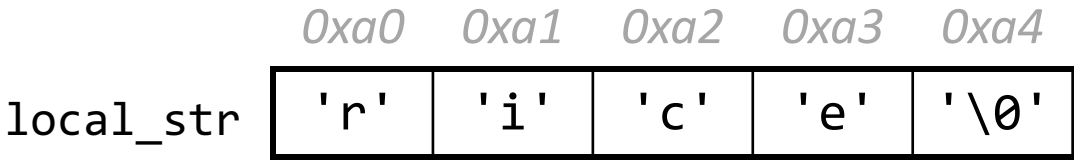
Passing in strings

How do you think the parameter `str` is being represented?

```
void fun_times(char *str) {
      str[0] = 'd';
}

int main(int argc, char *argv[]) {
      char local_str[] = "rice";
      fun_times(local_str);
      return 0;
}
```





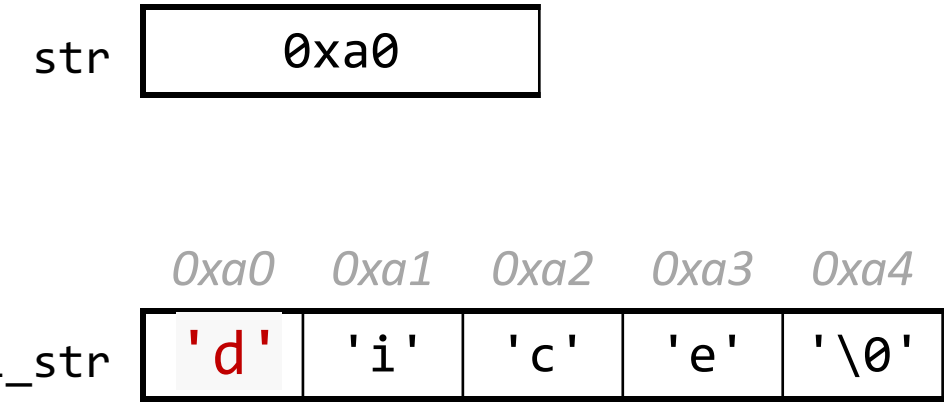
A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

Passing in strings

How do you think the parameter `str` is being represented?

```
void fun_times(char *str) {
      str[0] = 'd';
}

int main(int argc, char *argv[]) {
      char local_str[] = "rice";
      fun_times(local_str);
      return 0;
}
```



A. A copy of the array `local_str`

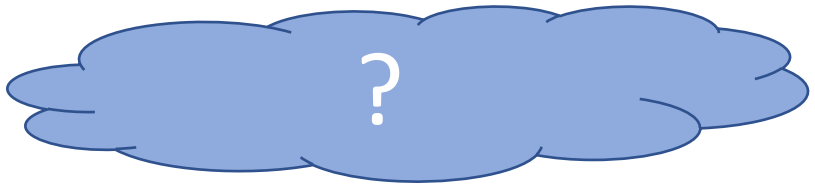
B. A pointer containing an address to the first element in `local_str`

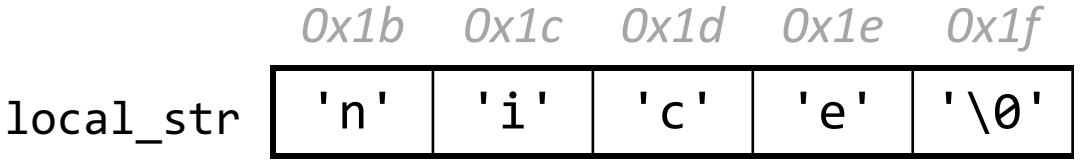
⚠️ Passing in strings, redux

How do you think the parameter `str` is being represented?

```
void more_fun_times(char str[]) {
      str[0] = 'm';
}

int main(int argc, char *argv[]) {
      char local_str[] = "nice";
      more_fun_times(local_str);
      return 0;
}
```





A. A copy of the array `local_str`
B. A pointer containing an address to the first element in `local_str`

🤔

⚠ Passing in strings, redux

How do you think the parameter `str` is being represented?

```
void more_fun_times(char str[]) {
      str[0] = 'm';
}

int main(int argc, char *argv[]) {
      char local_str[] = "nice";
      more_fun_times(local_str);
      return 0;
}
```

A. A copy of the array `local_str`

B. A pointer containing an address to the first element in `local_str`

`char arr[]` and `char *arr` are equivalent for parameter types.
(Not quite true for local variable declarations)