CS107 Winter 2020, Lecture 6
More Pointers and Arrays

reading:

K&R (5.2-5.5) or Essential C section 6

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 3: How can we effectively manage all types of memory in our programs?

Plan For Today

- Pointers and addresses, a review
- Arrays of different types
- **Break**: Announcements
- Programming with stack memory

```
cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
```

Warmup exercise

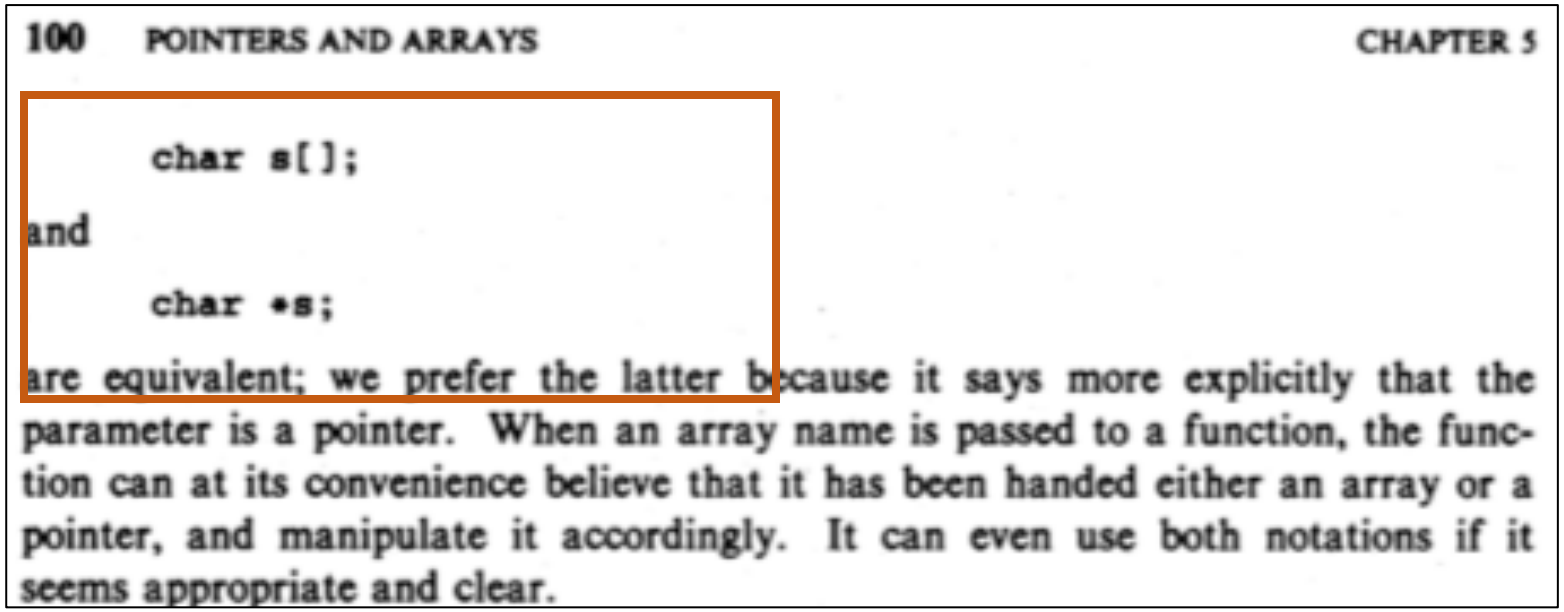
Compare and contrast the following declarations of x (some may not compile):

1. `char *x;` `char x[4];`
2. `char *x;` `char &x;`
3. `char *x;` `char **x;`
4. `char *x;` `char* x;`

A most unfortunate page break

Review

- K&R, top of p. 100



100 POINTERS AND ARRAYS CHAPTER 5

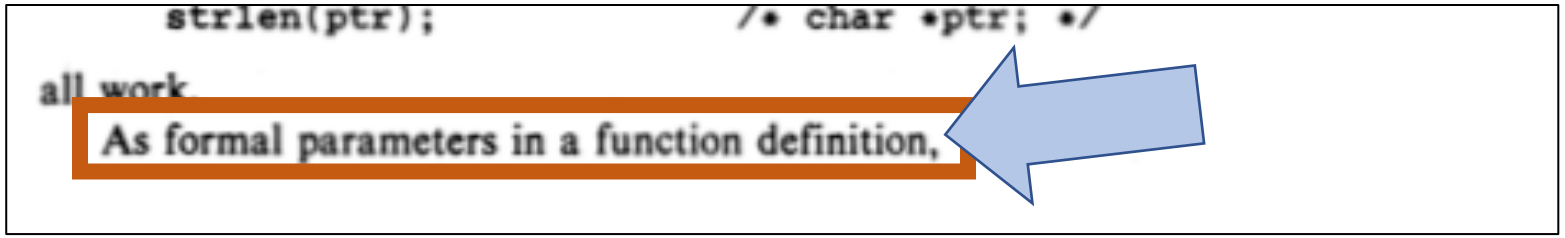
```
char s[];
```

and

```
char *s;
```

are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

- K&R, bottom of p. 99



```
strlen(ptr);                /* char *ptr; */
```

all work.

As formal parameters in a function definition,

⚠ Arrays are **NOT** pointers, even if they sometimes behave like them.

* Wars: Episode I (of 2)

In variable **declaration**, * creates a **pointer**.

```
char ch = 'r';
```

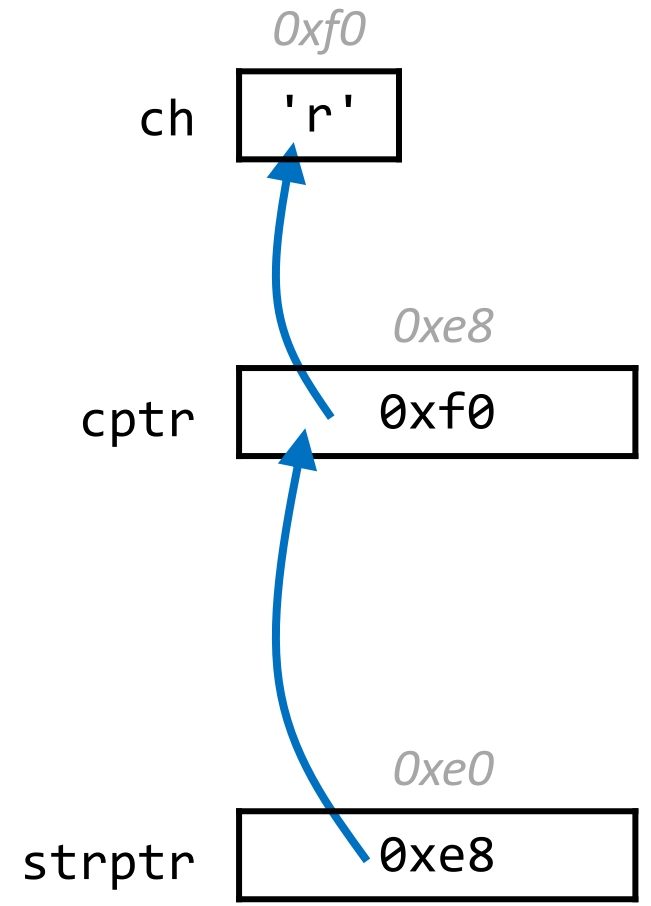
ch stores a char

```
char *cptr = &ch;
```

cptr stores an address of a char
(**points to** a char)

```
char **strptr = &cptr;
```

strptr stores an address of a char *
(**points to** a char *)



Warmup exercise

Compare and contrast the following declarations of x (some may not compile):

1. `char *x;` Pointer variable: stores an address to a `char`
2. `char *x;`
3. `char *x;` Variable stores an address of a `char`
4. `char *x;` No difference; left is preferred stylistically

`char x[4];` Array variable: contiguous block of memory on stack

`char &x;` Not a valid type; & operator means “address of”

`char **x;` Variable stores an address of a `char *`

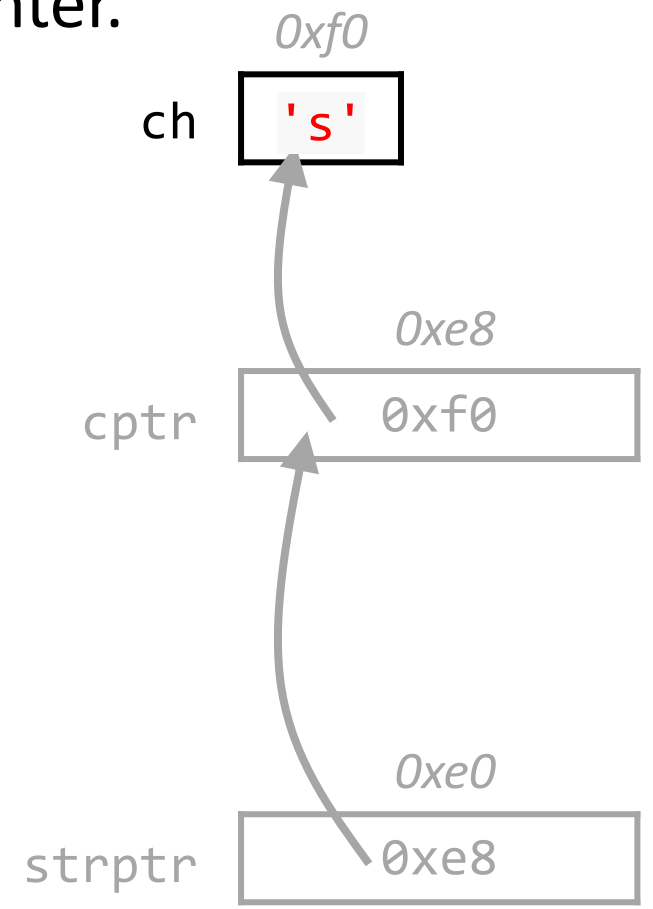
`char* x;`

* Wars: Episode II (of 2)

In **reading values from/storing values**, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in ch




```
char *cptr = &ch;
```

```
char **strptr = &cptr;
```

* Wars: Episode II (of 2)

In **reading values from/storing values**, * **dereferences** a pointer.

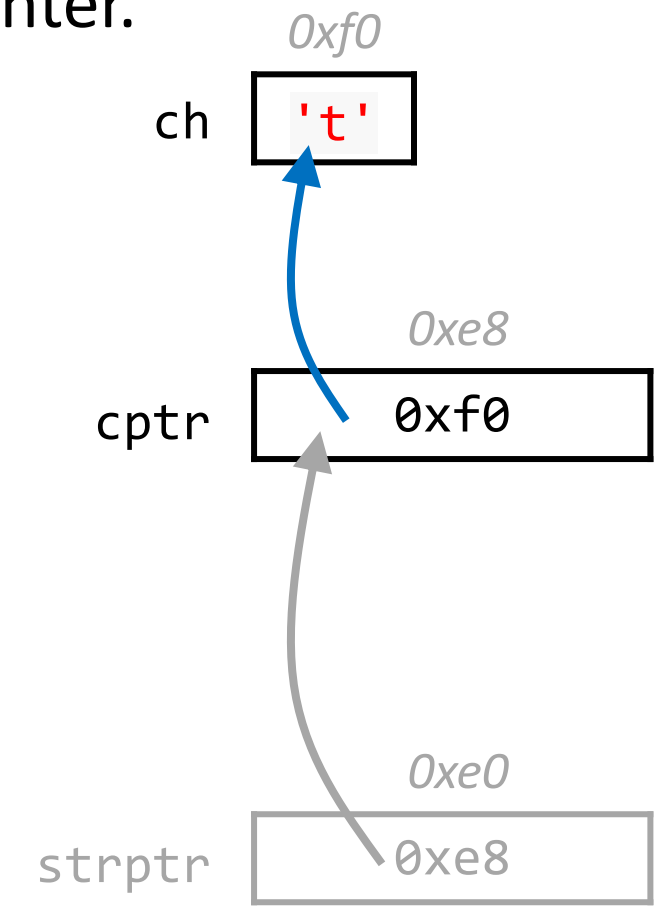
```
char ch = 'r';
ch = ch + 1;
```

Increment value stored in ch

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in cptr (increment char **pointed to**)

```
char **strptr = &cptr;
```



* Wars: Episode II (of 2)

In **reading values from/storing values**, * **dereferences** a pointer.

```
char ch = 'r';
ch = ch + 1;
```

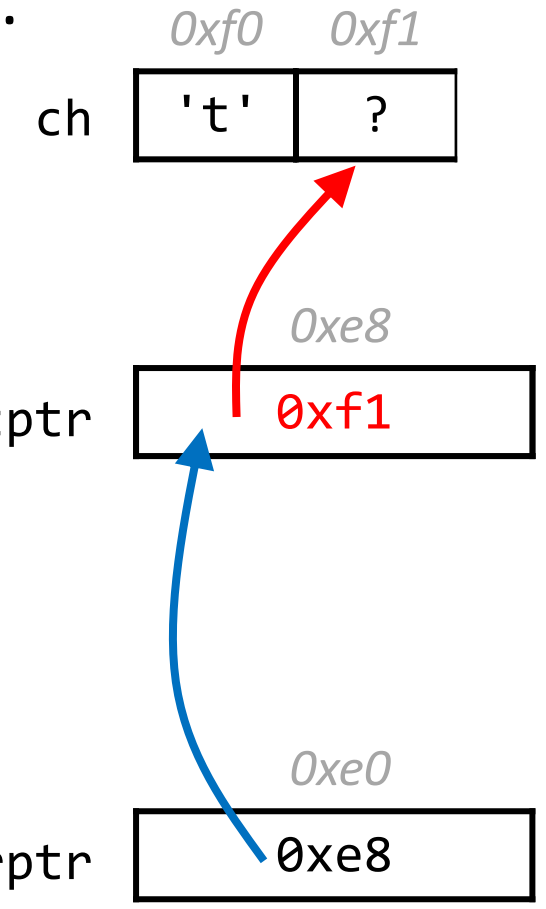
Increment value stored in ch

```
char *cptr = &ch;
*cptr = *cptr + 1;
```

Increment value stored at memory address in cptr (increment char **pointed to**)

```
char **strptr = &cptr;
*strptr = *strptr + 1;
```

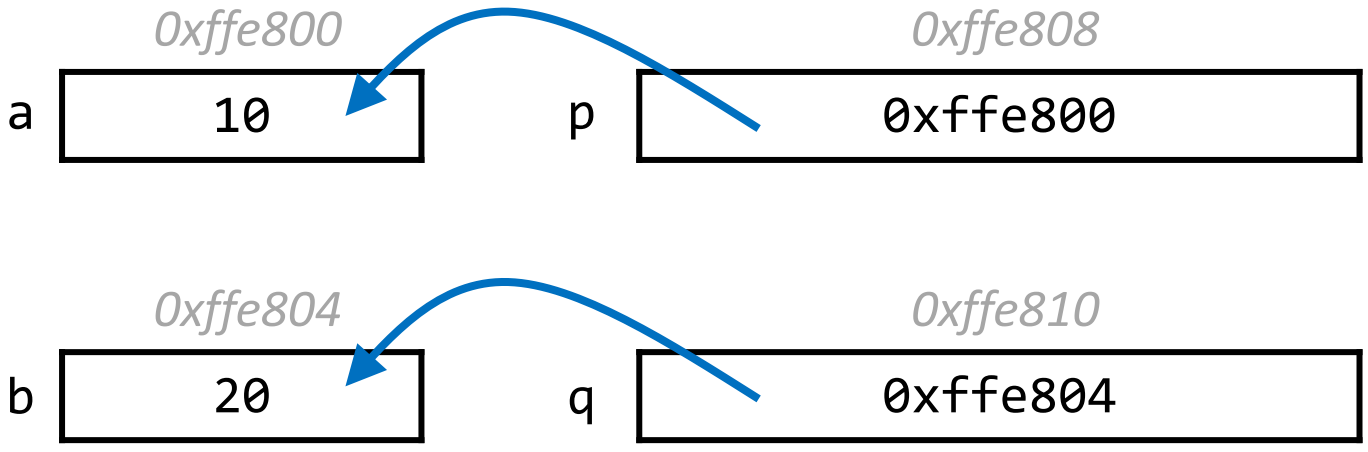
Increment value stored at memory address in cptr (increment address **pointed to**)



Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

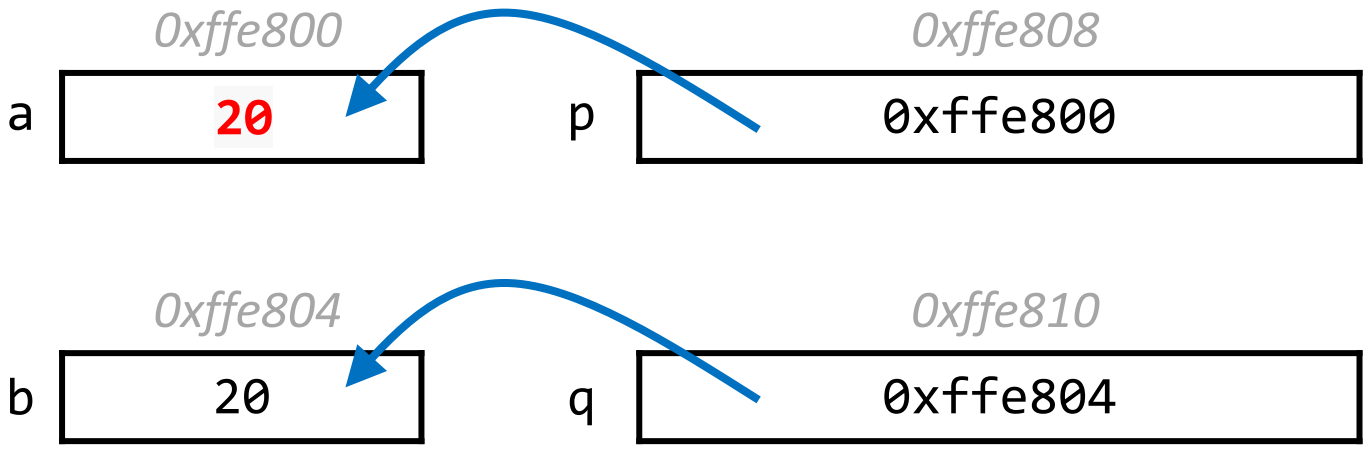


arrptr.c

Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

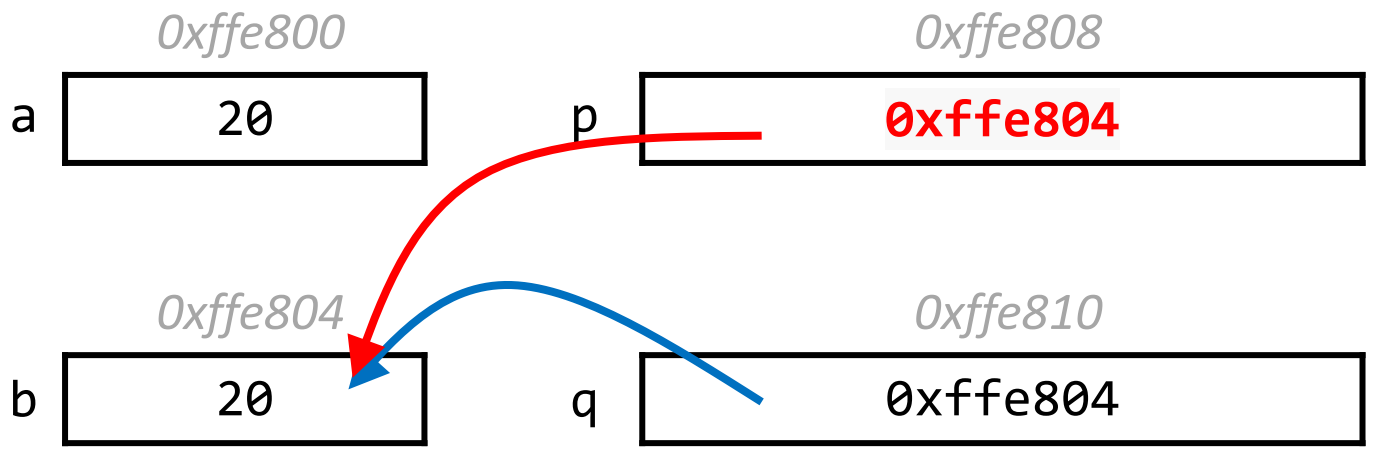


arrptr.c

Pen and paper: A * Wars Story

```
1 void binky() {
2     int a = 10;
3     int b = 20;
4     int *p = &a;
5     int *q = &b;
6
7     *p = *q;
8     p = q;
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



arrptr.c

Beyond char* and char[]

Recall: `sizeof()` returns the size of a variable type **in bytes** at compile time.

```
sizeof(char *) // 8
```

What will `sizeof(arr)` return in the following cases?

1.
```
char arr[] = "rey";
```

2.
```
int arr[] = {1, 32, 128, 256};
```

3.
```
char *arr[] = {
    "have a",
    "nice",
    "day"
};
```

A.	4	D.	24
B.	8	E.	32
C.	16	F.	Other

Pointer arithmetic

Array indexing is “syntactic sugar” for pointer arithmetic:

```
ptr + i          ⇔        &ptr[i]
*(ptr + i)       ⇔        ptr[i]
```

⚠ Pointer arithmetic **does not work in bytes**; it works on the type it points to. On `int*` addresses scale by `sizeof(int)`, on `char*` scale by `sizeof(char)` .

- This means too-large/negative subscripts will compile ☺

```
arr[99]                    arr[-1]
```

- You can use either syntax on either pointer or array.

Demo: Pointer arithmetic

`arrptr.c`

Useful Linux commands

Working with multiple terminals:

- Create new terminal, click and drag to resize, then ssh into both
- Useful to keep gdb open in one window, editor in another
- ⚠️ Don't edit the same file in two different windows

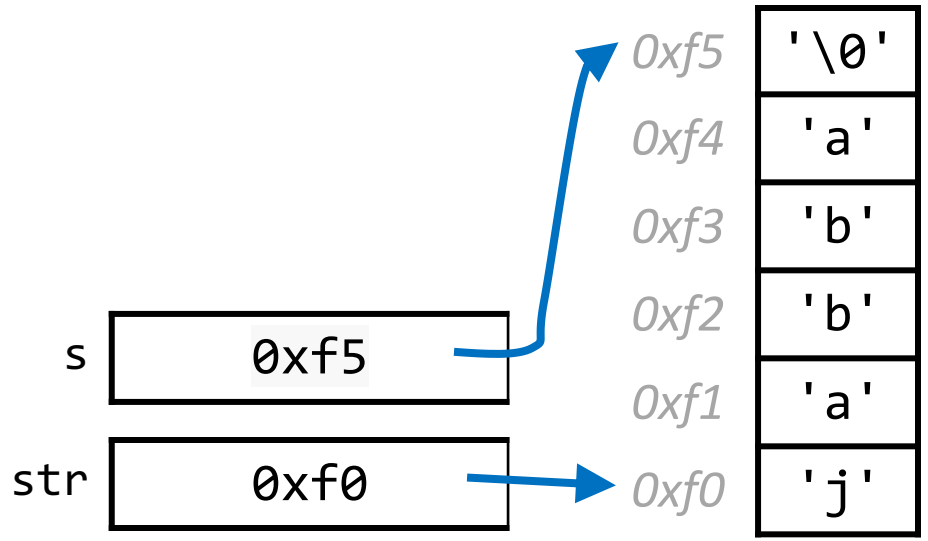
Whatever your workflow is, be sure to quit all jobs ("close all windows") when you log out/disconnect internet

Working with one terminal:

- `clear` (or Ctrl-k/⌘-k)
  clears current terminal output
- Ctrl-z
  temporarily suspends current **job** (editor, gdb, etc.)
- `jobs` lists all jobs
- `fg %1` continues job [1]
  (see `jobs` for job list)
- `kill %1` kills job
  ("force quit" – only use if necessary)

Code study: `strlen`

```
1 /* Based on FreeBSD 6.2's strlen */
2 size_t freebsd_strlen(char *str) {
3     char *s;
4     for (s = str; *s; s++) ;
5     return (s - str);
6 }
```



Some notes:

- The `size_t` (`unsigned long`) represents size of objects in bytes.
- Line 4: Recall the character `'\0'` has ASCII value 0.

This function exclusively uses pointer arithmetic.

- Line 4: ++ operator on pointers needs to know that `s` points to `chars`!
- Line 5: - operator on pointers needs to know both `s` and `str` point to `chars`!

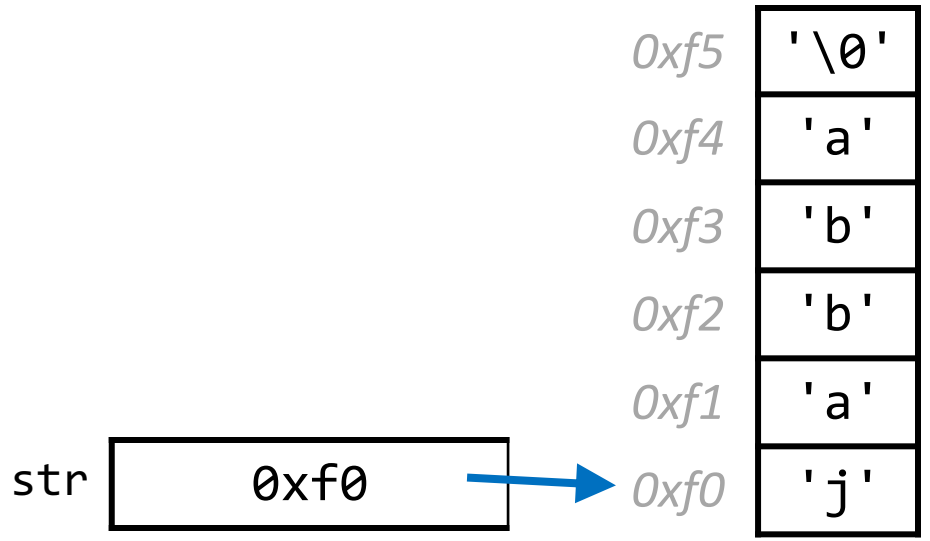
`strlen_code.c`

Write your own: `strlen`

Bonus

```
1 /* Based on FreeBSD 6.2's strlen */
2 size_t freebsd_strlen(char *str) {
3     char *s;
4     for (s = str; *s; s++) ;
5     return (s - str);
6 }
```

```
1 /* Using array indexing */
2 size_t my_strlen(char *str) {
3     int n = 0;
4     while (    ?    ) {
5         n++;
6     }
7     return n;
8 }
```



Line 4: Which loop condition?

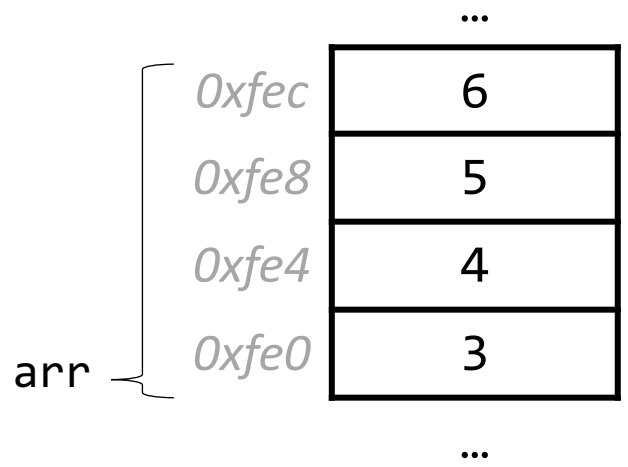
A. `str[n]`, or equivalently, `str[n] != '\0'`
B. `str + n != ""`
C. `str + n != NULL`
D. Something else

Bonus: Can we write a strlen function for other array types (e.g., int arrays)?

`strlen` for other array types?

- **No!** Arrays in general are not null-terminated.
- For strings, C designer Dennis Ritchie decided to use a special terminating character (`'\0'`). Other array types don't have this (e.g., what's a "null-terminating integer"?)

```
int arr[] = {3, 4, 5, 6};
```



When writing functions for other array types, you need to explicitly pass in the length of an array:

```
void traverse_int_array(int arr[], int len);
```

- Note: assign2's `const char *envp[]` has a NULL pointer placed after its last entry, but this is extremely uncommon.

Questions?

Plan For Today

- Pointers and addresses, a review
- Arrays of different types
- Double pointers, a review
- Programming with stack memory
- **Announcements**
- Other topics: **const**, **struct** and ternary

```
cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
```

C parameters are pass-by-value

Review

A parameter is the callee's variable—distinct from caller's variable.

In all cases (other than array-as-parameter case), values are copied.

callee

```
void change(char ch) {
    ch = toupper(ch);
}
```

caller

```
int main(...) {
    char letter = 'a';
    change(letter);
    return 0;
}
```

`letter` is **unchanged**

```
void change(char *p_ch) {
    *p_ch = toupper(*p_ch);
}

int main(...) {
    char letter = 'a';
    change(&letter);
    return 0;
}
```

`letter` is **changed**

To achieve pass-by-reference, manually use pointers.

The same applies to pointer parameters

callee

```
void change(char *str) {
    str = str + 1;
}
```

caller

```
int main(...) {
    char *name = "drain";
    change(name);
    return 0;
}
```

name is **unchanged**

```
void change(char **p_str) {
    *p_str = *p_str + 1;
}

int main(...) {
    char *name = "drain";
    change(&name);
    return 0;
}
```

name is **changed**

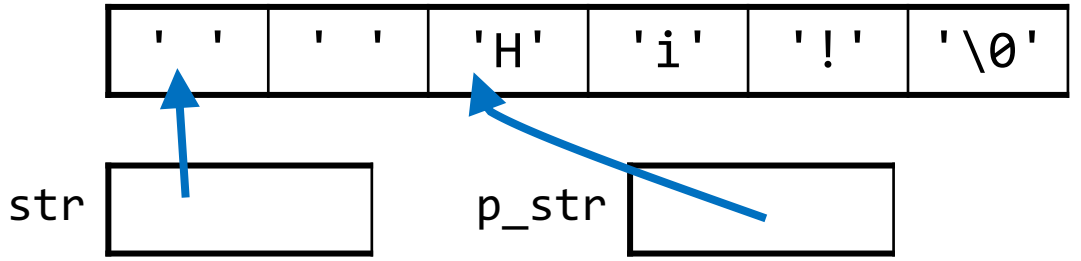
Tip: Always draw a picture!!

Skip spaces

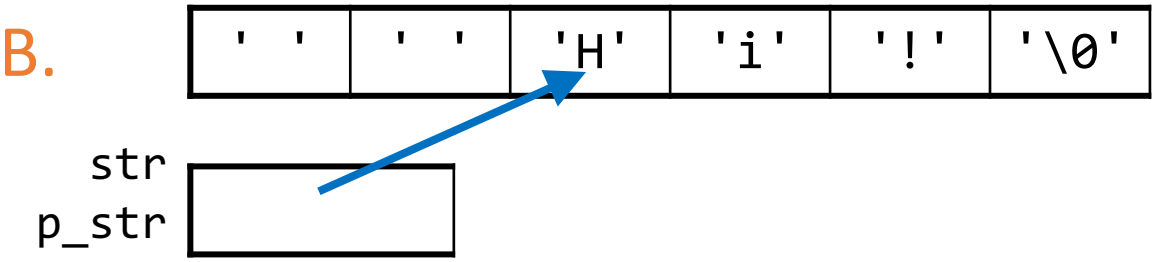
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?

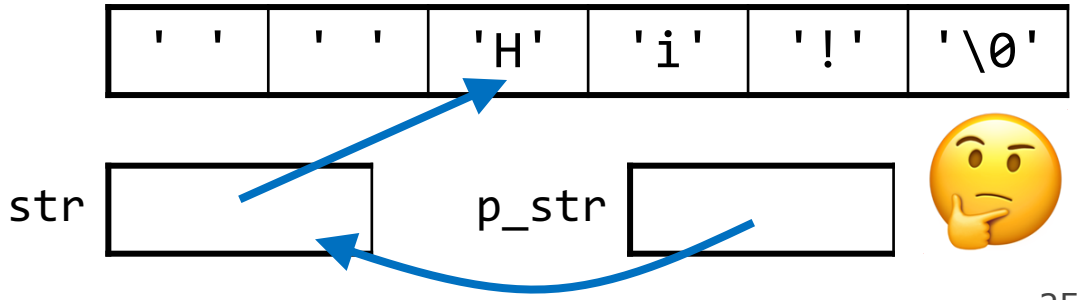
A.



B.



C.

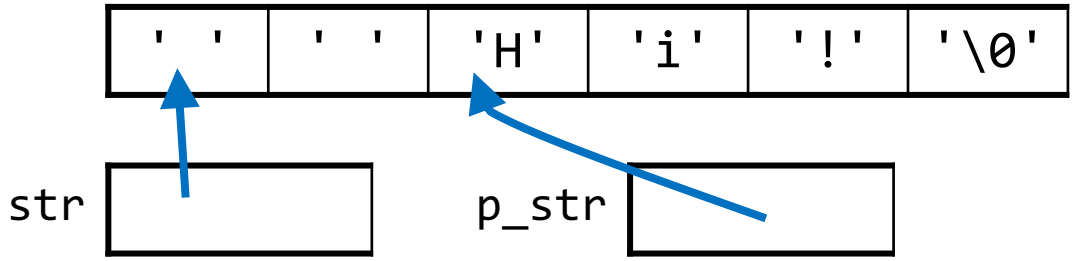


Skip spaces

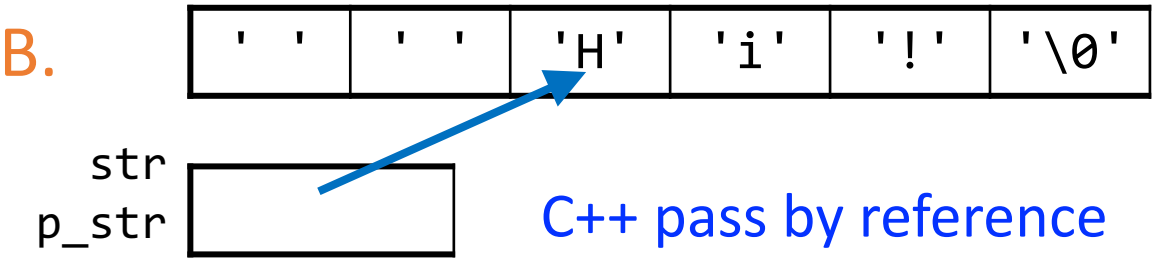
```
1  void skip_spaces(char **p_str) {
2    int num = strspn(*p_str, " ");
3    *p_str = *p_str + num;
4  }
5  int main(int argc, char *argv[]){
6    char *str = "  Hi!";
7    skip_spaces(&str);
8    printf("%s", str); // "Hi!"
9    return 0;
10 }
```

What diagram most accurately depicts program state at Line 4 (before `skip_spaces` returns to `main`)?

A.

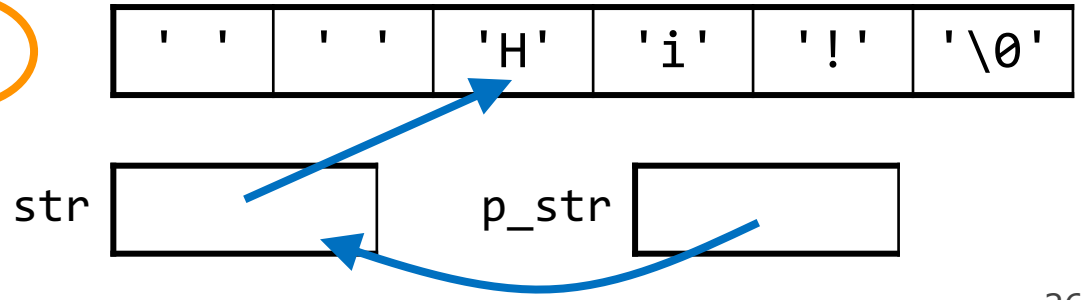


B.



C++ pass by reference

C.



Questions?

Plan For Today

- Pointers and addresses, a review
- Arrays of different types
- Double pointers, a review
- **Break**: Announcements
- Programming with stack memory

```
cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
```

Announcements

- lab2 check-in
- assign2: lots of string practice, some double pointers
- Read Piazza FAQ for each assignment

- Education Research Survey: http://bit.ly/2TwXjcS
- Social Good Fellowship: https://web.stanford.edu/class/cs107/resources/cs-sg-fellowship.pdf
- CS198: deadline Thursday 1/30, 11:59PM cs198.stanford.edu

Joke break

What data type do you need to store this number?

```
INT_MAX:      2147483647
UINT_MAX:     4294967295
FLOAT_MAX: 3.40282e+38
```

(we'll come back to this in 2 weeks)

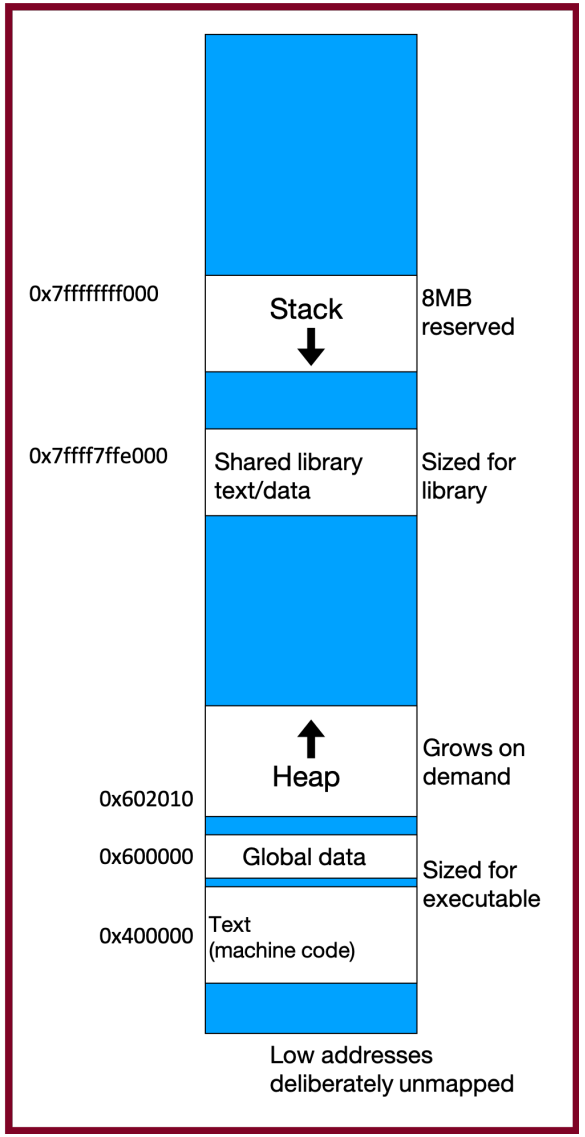
Plan For Today

- Pointers and addresses, a review
- Arrays of different types
- Double pointers, a review
- **Break**: Announcements
- Programming with stack memory

```
cp -r /afs/ir/class/cs107/samples/lectures/lect6 .
```

Memory Layout**

- The **stack** is the place where all local variables and parameters live for each function.*
- A function's **stack frame** goes away when the function returns.
- The stack grows **downwards** when a new function is called. It shrinks **upwards** when the function is finished.



*gcc, our compiler, uses option `-O0` to force everything (including parameters) on the stack (see week 6, "registers," for more info)

**Assume all 18 exabytes of system memory accessible (read about "virtual memory" for more info)

The Stack

- The stack behaves like a…well…stack! A new function call **pushes** on a new frame. A completed function call **pops** off the most recent frame.
- A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.

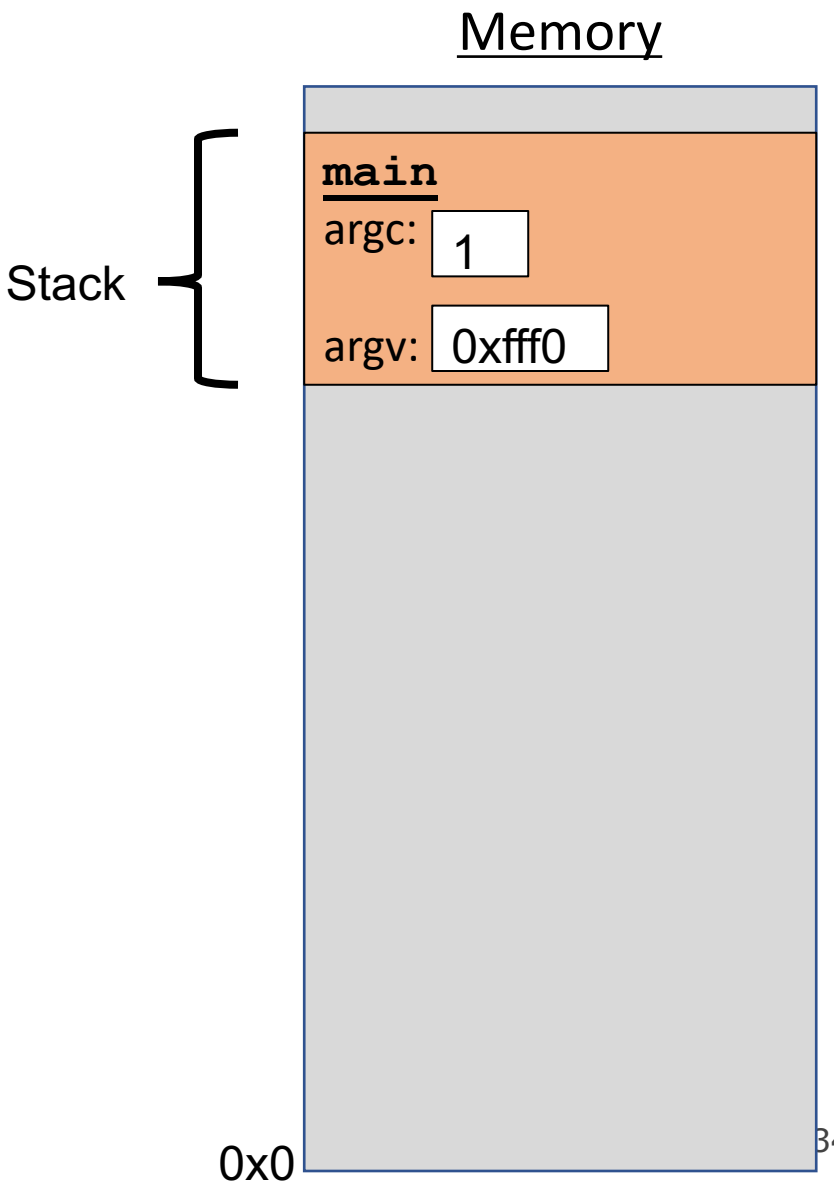
⚠️*Interesting and important fact:* C does not clear out memory when a function's frame is removed. Instead, it just marks that memory as usable for the next function call. This is more efficient!

The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c

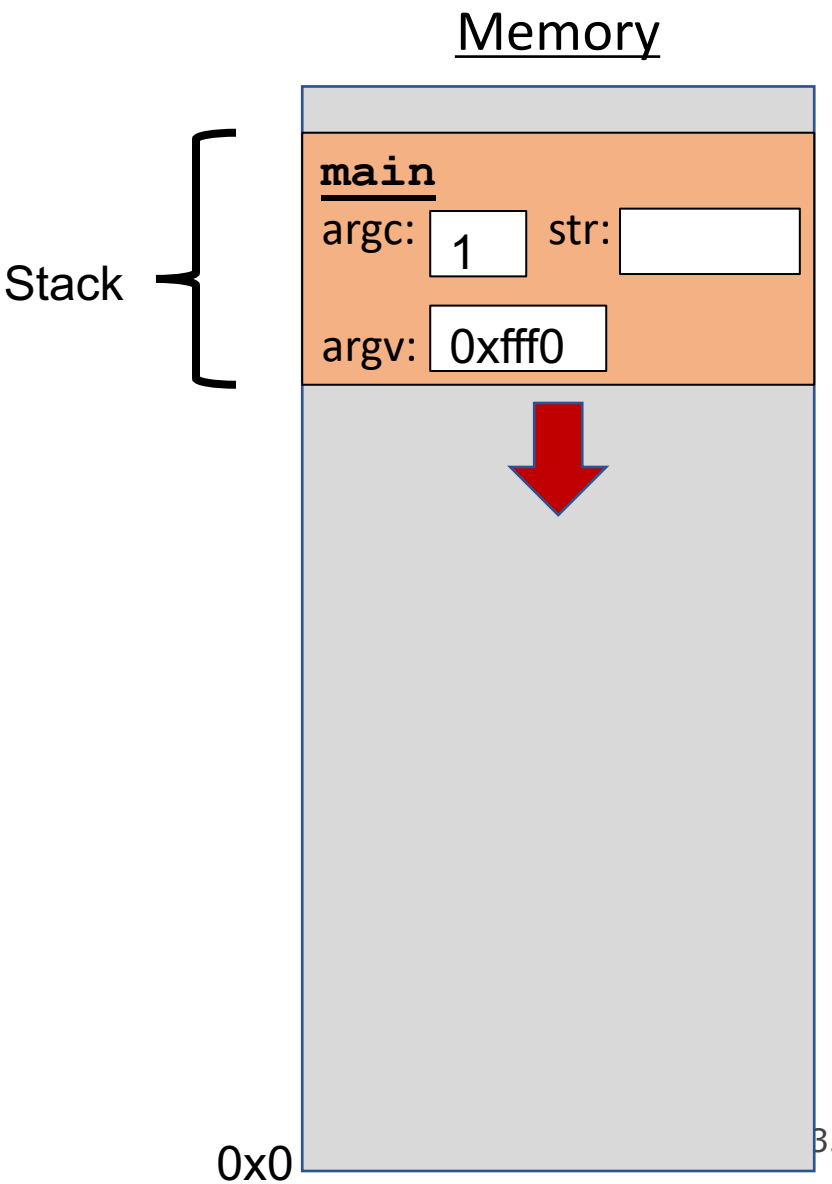


The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c

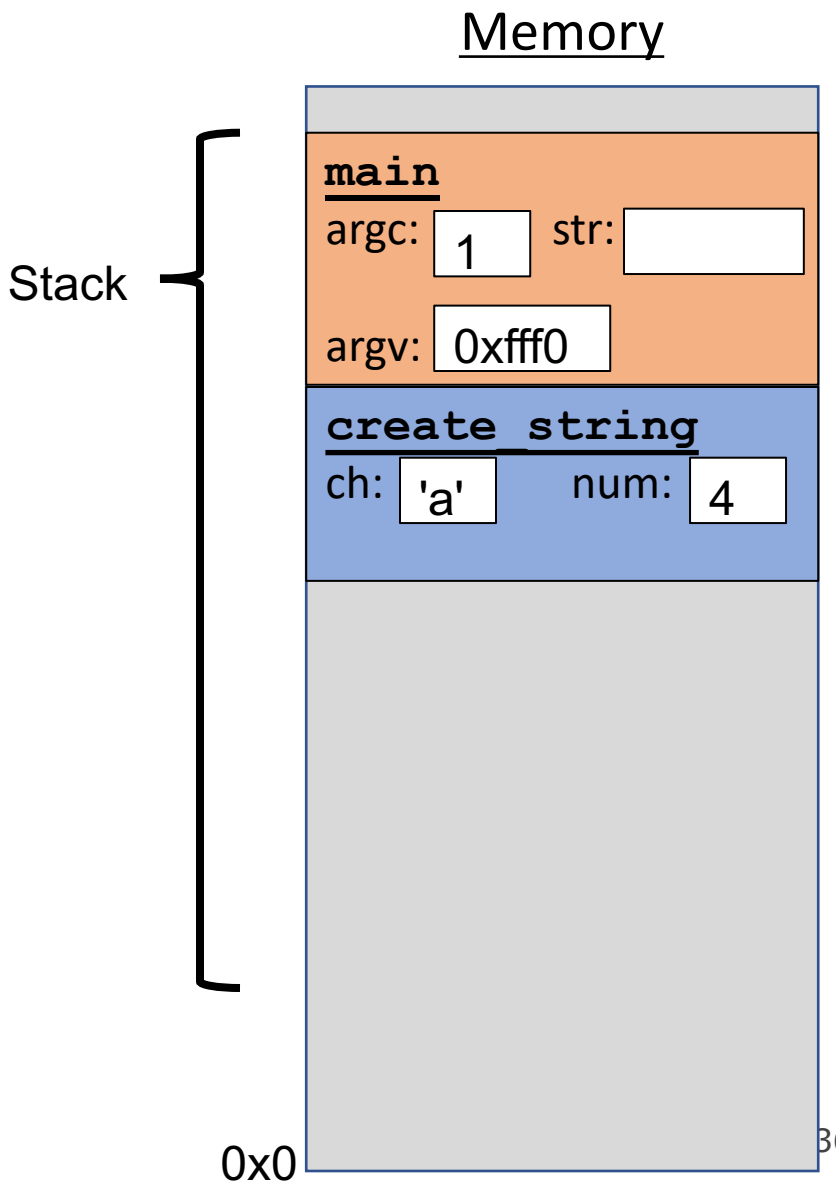


The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c

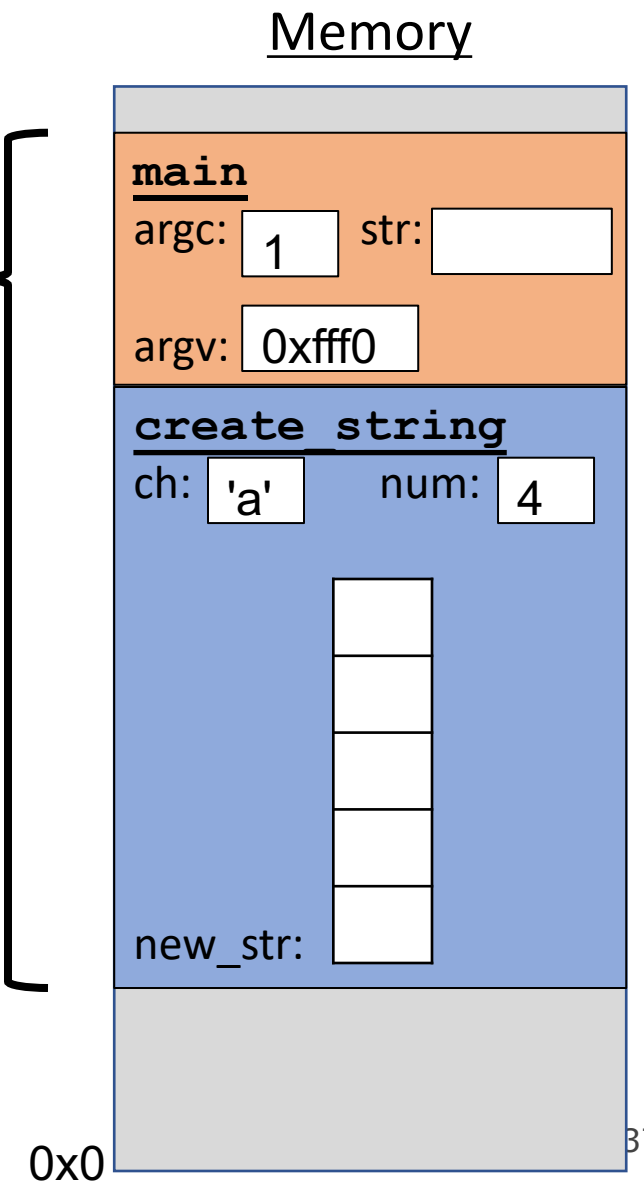


The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c

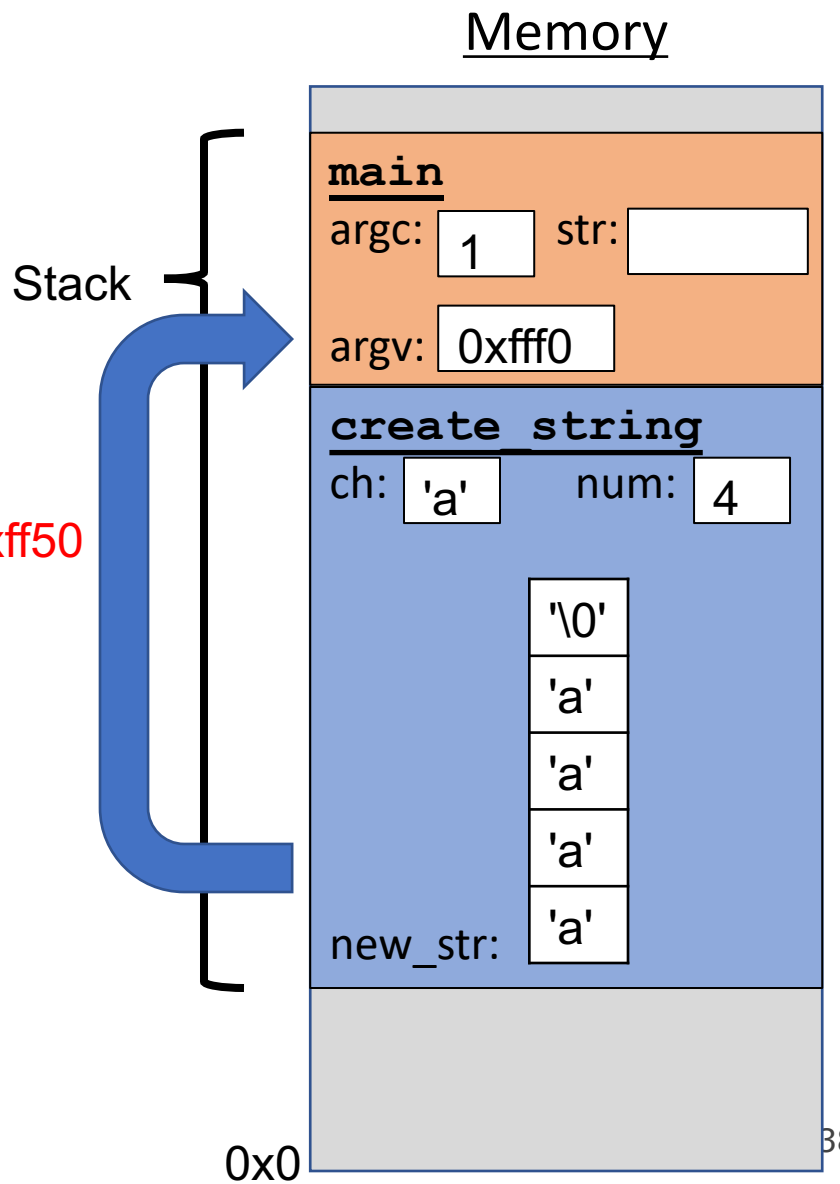


The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

Returns e.g. 0xff50



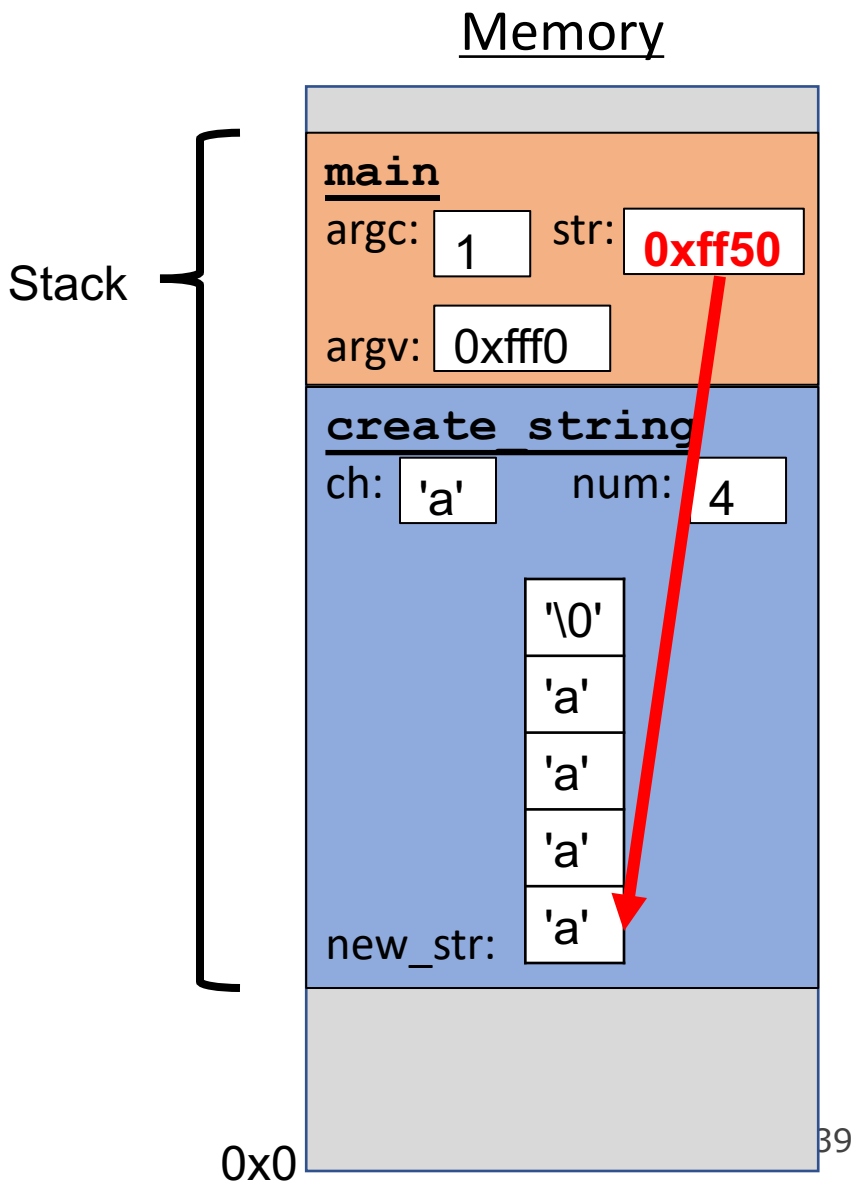
stack_chaos.c

The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c

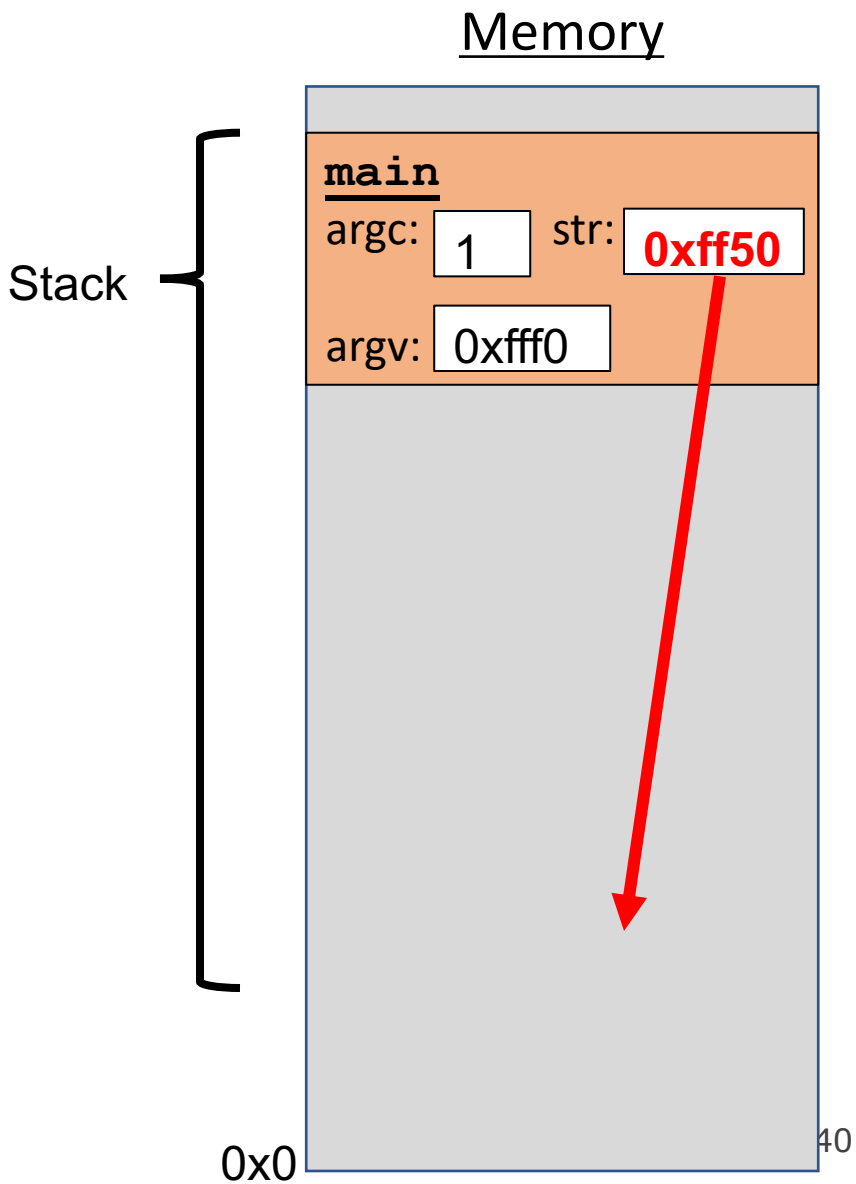


The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c

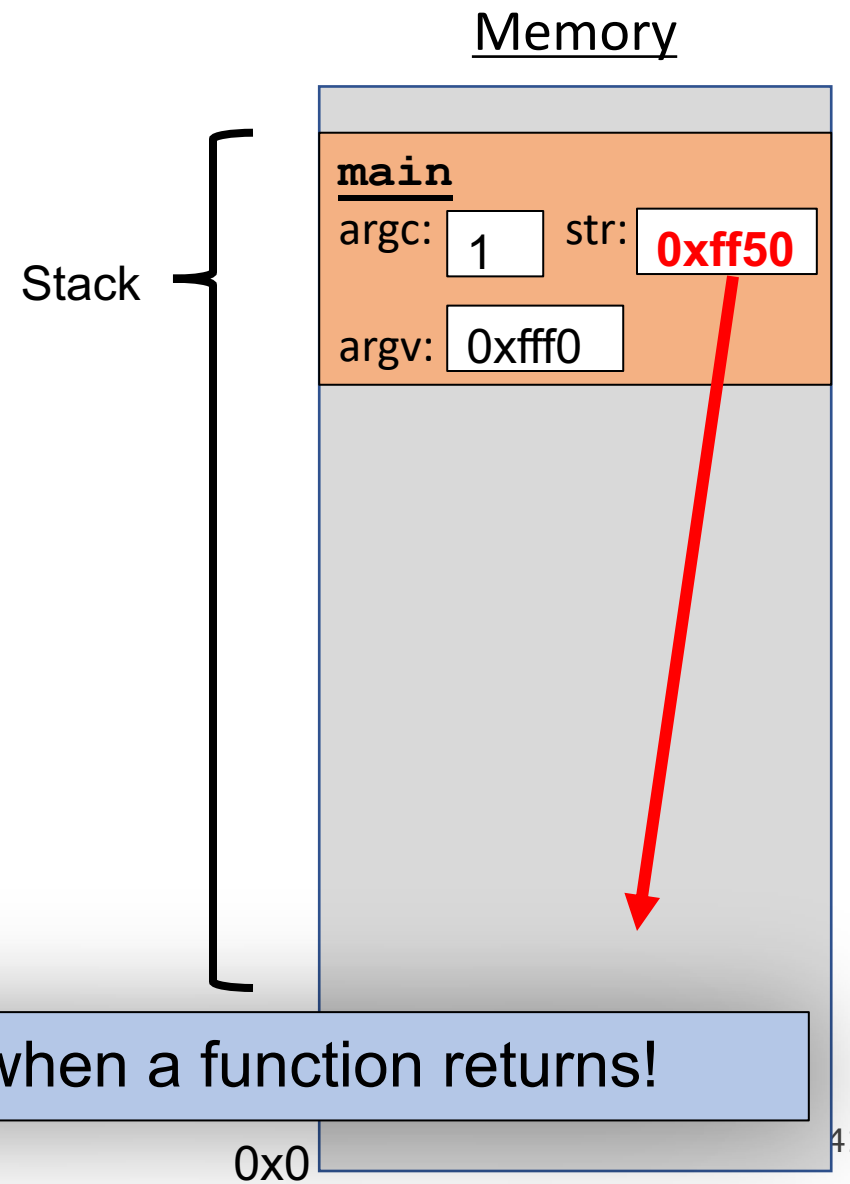


The Stack

```
1  char *create_string(char ch, int num) {
2      char new_str[num + 1];
3      for (int i = 0; i < num; i++) {
4          new_str[i] = ch;
5      }
6      new_str[num] = '\0';
7      return new_str;
8  }

1  int main(int argc, char *argv[]) {
2      char *str = create_string('a', 4);
3      printf("%s", str);   // want "aaaa"
4      return 0;
5  }
```

stack_chaos.c



⚠ Stack memory gets reused when a function returns!

Demo: Stack chaos

`stack_chaos.c`

Overflow

(for next time)

Stacked Against Us

How do we make sure memory content persists after a function exits?

If we're committed to stack-only memory: let the caller allocate memory and pass it to the callee.

(we'll learn heap memory next week)

Programming with the stack (1/2)

Recall that arrays passed in as parameters are converted to pointers.

Strategy #1: Allow callee to modify existing arrays.

```
void binky(char *ptr) {
    ptr[0] = 'B';
}

int main(int argc, char *argv[]) {
    char arr[] = "asdf";
    modify_array(arr);
    printf("%s", arr);      // Bsdf
    return 0;
}
```

Programming with the stack (2/2)

Recall that arrays passed in as parameters are converted to pointers.

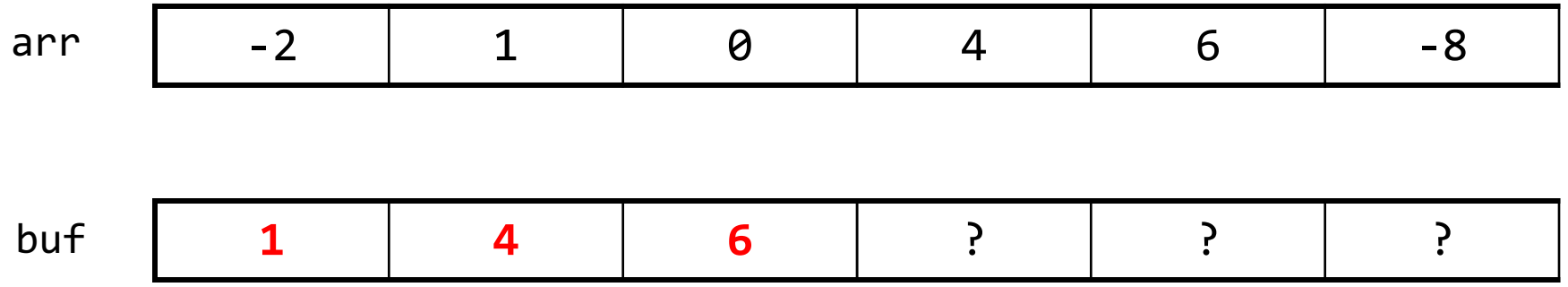
Strategy #2: Pass in a pre-allocated **buffer** that the caller can write to.

```
void pig_latin(char *out, char *in) {
    /* write changes to out */
}

int main(int argc, char *argv[]) {
    char str[] = "be";
    char converted[50];                  // output buffer
    pig_latin(converted, str);
    printf("%s %s", str, converted);     // be ebay
    return 0;
}
```

Practice: Copy positive ints

Write a function copy_pos that copies positive integers from in to out and returns the number of integers copied (i.e., the length of out).




```
 8 int main(int argc, char *argv[]) {
 9     int arr[] = {-2, 1, 0, 4, 6, -8};
10     int len = ______?______;                  // 6
11     int buf[len];
12     int buf_len = copy_pos(buf, arr, len); // returns 3
13     return 0;
14 }
```

Practice: Copy positive ints

Write a function copy_pos that copies positive integers from in to out and returns the number of integers copied (i.e., the length of out).

```
 1 int copy_pos(int *out, int *in, int len) {
 2     int num = 0;
 3     for (int i = 0; i < len; i++) {
 4         /* fill this part in */
 5     }
 6     return num;
 7 }
 8 int main(int argc, char *argv[]) {
 9     int arr[] = {-2, 1, 0, 4, 6, -8};
10     int len = ______?______;
11     int buf[len];
12     int buf_len = copy_pos(buf, arr, len);
13     return 0;
14 }
```

- Line 10: How do we find array length with sizeof?
- Line 6: Why do we return the length of out?
- Line 4: What goes in the for-loop? (multiple lines)

Practice: Copy positive integers

`int_array.c`

(lecture: with array indexing)
Bonus: Try writing this with pointer arithmetic!