Why We ❤ The Stack

• **It is fast.** Your program already has that memory reserved for it!

• **It is convenient.** Memory is handled automatically, and is fast because old memory is left in place and marked as usable for future function calls.

• **It is safe.** You specify variable types, and the compiler can therefore do checks on the data. We’ll see later this is not necessarily true on the heap.
Why We ❤ The Heap

• **It is plentiful.** The stack has at most 8MB by default. The heap can provide more on demand!

• **Allocations are resizable.** Unlike on the stack, if you allocate something (e.g. an array), you can change the size of it later using realloc.

• **Scope.** The memory is not cleaned up when its function exits; instead, you control when the memory is freed.
As a general rule of thumb, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.

Heap allocation is a necessity when:

- you have a very large allocation that could blow out the stack
- you need to control the memory lifetime, or memory must persist outside of a function call
- you need to resize memory after its initial allocation
CS107 Topic 4: How can we use our knowledge of memory and data representation to write code that works with any data type?
Learning Goals

• Learn how to write C code that works with any data type.
• Learn about how to use void * and avoid potential pitfalls.
Plan For Today

• **Overview**: Generics
• Generic Swap
• Generics Pitfalls
• **Announcements**
• Generic Array Swap
• Generic Stack
Plan For Today

• **Overview: Generics**
• Generic Swap
• Generics Pitfalls
• **Announcements**
• Generic Array Swap
• Generic Stack
Generics

• We always strive to write code that is as general-purpose as possible.
• Generic code reduces code duplication, and means you can make improvements and fix bugs in one place rather than many.
• Generics is used throughout C for functions to sort any array, search any array, free arbitrary memory, and more.
• How can we write generic code in C?
Plan For Today

• **Overview:** Generics
• **Generic Swap**
• Generics Pitfalls
• **Announcements**
• Generic Array Swap
• Generic Stack
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
You’re asked to write a function that swaps two numbers.

```c
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    int x = 2;
    int y = 5;
    swap_int(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```
“Oh, when I said ‘numbers’ I meant shorts, not ints.”
void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    short x = 2;
    short y = 5;
    swap_short(&x, &y);
    // want x = 5, y = 2
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
void swap_short(short *a, short *b) {
  short temp = *a;
  *a = *b;
  *b = temp;
}

int main(int argc, char *argv[]) {
  short x = 2;
  short y = 5;
  swap_short(&x, &y);
  // want x = 5, y = 2
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
“You know what, I goofed. We’re going to use strings. Could you write something to swap those?”
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
“Awesome! Thanks.”
“Awesome! Thanks. We also have 20 custom struct types. Could you write swap for those too?”
Wouldn’t it be nice if we could write one function that would work with any parameter type, instead of so many different versions?

```c
void swap_int(int *a, int *b) { ... }
void swap_float(float *a, float *b) { ... }
void swap_size_t(size_t *a, size_t *b) { ... }
void swap_double(double *a, double *b) { ... }
void swap_string(char **a, char **b) { ... }
void swap_mystruct(mystruct *a, mystruct *b) { ... }
...
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void swap_short(short *a, short *b) {
    short temp = *a;
    *a = *b;
    *b = temp;
}

void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}

int temp = *data1ptr;  // 4 bytes
short temp = *data1ptr;  // 2 bytes
char *temp = *data1ptr;  // 8 bytes

Problem: each type may need a different size temp!
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}

*data1Ptr = *data2ptr;  // 4 bytes
*data1Ptr = *data2ptr;  // 2 bytes
*data1Ptr = *data2ptr;  // 8 bytes

**Problem:** each type needs to copy a different amount of data!
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}

*data2ptr = temp;
*data2ptr = temp;
*data2ptr = temp;

Problem: each type needs to copy a different amount of data!
C knows the size of temp, and knows how many bytes to copy, because of the variable types.
Is there a way to make a version that doesn’t care about the variable types?
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
void swap(pointer to data1, pointer to data2) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}

Generic Swap
void swap(void *data1ptr, void *data2ptr) {
    store a copy of data1 in temporary storage
    copy data2 to location of data1
    copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
   // store a copy of data1 in temporary storage
   // copy data2 to location of data1
   // copy data in temporary storage to location of data2
}

If we don’t know the data type, we don’t know how many bytes it is. Let’s take that as another parameter.
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    void temp; ???
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

Let’s start by making space to store the temporary value. How can we make this temp space?
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

Now, how can we copy in what data1ptr points to into temp?
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; // ??
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; ???
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

We can’t dereference a void * (or set an array equal to something). C doesn’t know what it points to! Therefore, it doesn’t know how many bytes there it should be looking at.
memcpy is a function that copies a specified amount of bytes at one address to another address.

```c
void *memcpy(void *dest, const void *src, size_t n);
```

It copies the next n bytes that `src` points to to the location contained in `dest`. (It also returns `dest`). It does **not** support regions of memory that overlap.

```c
int x = 5;
int y = 4;
memcpy(&x, &y, sizeof(x));   // x = y
```
memcpy is a function that copies a specified amount of bytes at one address to another address.

```c
void *memcpy(void *dest, const void *src, size_t n);
```

It copies the next n bytes that src points to to the location contained in dest. (It also returns dest). It does not support regions of memory that overlap.

```c
int x = 5;
int y = 4;
memcpy(&x, &y, sizeof(x));
```

memcpy must take pointers to the bytes to work with to know where they live and where they should be copied to.
memmove

**memmove** is the same as `memcpy`, but supports overlapping regions of memory. (Unlike its name implies, it still “copies”).

```c
void *memmove(void *dest, const void *src, size_t n);
```

It copies the next n bytes that *src* points to to the location contained in *dest*. (It also returns *dest*).
When might `memmove` be useful?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

| 4 | 5 | 6 | 7 | 5 | 6 | 7 |
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    temp = *data1ptr; //
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

We can’t dereference a **void** *. C doesn’t know what it points to! Therefore, it doesn’t know how many bytes there it should be looking at.
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

We can copy the bytes ourselves into temp! This is equivalent to temp = *data1ptr in non-generic versions, but this works for any type of any size.
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

How can we copy data2 to the location of data1?
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    *data1ptr = *data2ptr; ???
    // copy data in temporary storage to location of data2
}

How can we copy data2 to the location of data1?
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
}

How can we copy data2 to the location of data1?
memcpy!
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
}

How can we copy temp’s data to the location of data2?
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}

How can we copy temp’s data to the location of data2? **memcpy**!
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}

ing int x = 2;
ing int y = 5;
swap(&x, &y, sizeof(x));
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}

short x = 2;
short y = 5;
swap(&x, &y, sizeof(x));
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}

char *x = "2";
char *y = "5";
swap(&x, &y, sizeof(x));
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    // store a copy of data1 in temporary storage
    memcpy(temp, data1ptr, nbytes);
    // copy data2 to location of data1
    memcpy(data1ptr, data2ptr, nbytes);
    // copy data in temporary storage to location of data2
    memcpy(data2ptr, temp, nbytes);
}

mystruct x = {...};
mystruct y = {...};
swap(&x, &y, sizeof(x));
• We can use `void *` and `memcpy` to handle memory as generic bytes.
• As long as we are given where the data of importance is, and how big it is, we can handle it!

```c
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
  char temp[nbytes];
  memcpy(temp, data1ptr, nbytes);
  memcpy(data1ptr, data2ptr, nbytes);
  memcpy(data2ptr, temp, nbytes);
}
```
Plan For Today

• Overview: Generics
• Generic Swap
• **Generics Pitfalls**
• Announcements
• Generic Array Swap
• Generic Stack
Void * Pitfalls

• **void** *s are powerful, but dangerous - C cannot do as much checking!

• E.g. with **int**, C would never let you swap *half* of an int. With **void** *s*, this can happen! *(How? Let’s find out!)*
Demo: Void *'s Gone Wrong
Void * Pitfalls

- Void * has more room for error because it manipulates arbitrary bytes without knowing what they represent. This can result in some strange memory Frankenstein's!
Plan For Today

• **Overview:** Generics
• Generic Swap
• Generics Pitfalls

• **Announcements**
• Generic Array Swap
• Generic Stack
Announcements

• New office hours added: Thurs. 7-9PM PST

• assign3
  • We’ve added a new test file, colors_no_end_newline, without a \n at the end of the file
  • Remember to git add any custom files you make for your custom_tests
  • You do not have to worry about memory leaks if a heap error occurs
Plan For Today

- **Overview**: Generics
- Generic Swap
- Generics Pitfalls
- **Announcements**
- **Generic Array Swap**
- Generic Stack
You’re asked to write a function that swaps the first and last elements in an array of numbers.

```c
void swap_ends_int(int *arr, size_t nelems) {
    int tmp = arr[0];
    arr[0] = arr[nelems - 1];
    arr[nelems - 1] = tmp;
}

int main(int argc, char *argv[]) {
    int nums[] = {5, 2, 3, 4, 1};
    size_t nelems = sizeof(nums) / sizeof(nums[0]);
    swap_ends_int(nums, nelems);
    // want nums[0] = 1, nums[4] = 5
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);
    return 0;
}
```

Wait – we just wrote a generic swap function. Let’s use that!
Swap Ends

You’re asked to write a function that swaps the first and last elements in an array of numbers.

```c
void swap_ends_int(int *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

int main(int argc, char *argv[]) {
    int nums[] = {5, 2, 3, 4, 1};
    size_t nelems = sizeof(nums) / sizeof(nums[0]);
    swap_ends_int(nums, nelems);
    // want nums[0] = 1, nums[4] = 5
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);
    return 0;
}
```

Wait – we just wrote a generic swap function. Let’s use that!
Let’s write out what some other versions would look like (just in case).

```c
void swap_ends_int(int *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

void swap_ends_short(short *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

void swap_ends_string(char **arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}

void swap_ends_float(float *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

The code seems to be the same regardless of the type!
Let’s write a version of swap_ends that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

Is this generic? Does this work?

Unfortunately not. First, we no longer know the element size. Second, pointer arithmetic depends on the type of data being pointed to. With a void *, we lose that information!
Let’s write a version of `swap_ends` that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

Is this generic? Does this work?

**Unfortunately not.** First, we no longer know the element size. Second, pointer arithmetic depends on the type of data being pointed to. With a `void *`, we lose that information!
Let’s write a version of swap_ends that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(*arr));
}
```

We need to know the element size, so let’s add a parameter.
Let’s write a version of swap_ends that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + nelems - 1, elem_bytes);
}
```

We need to know the element size, so let’s add a parameter.
Pointer Arithmetic

\[ \text{arr} + \text{nelems} - 1 \]

Let’s say \( \text{nelems} = 4 \). How many bytes beyond \( \text{arr} \) is this?

If it’s an array of...

\textbf{Int}?
Let’s say nelems = 4. How many bytes beyond arr is this?

If it’s an array of...

**Int:** adds 3 places to arr, and 3 * sizeof(int) = 12 bytes
Let’s say nelems = 4. How many bytes beyond arr is this?

If it’s an array of...

**Int:** adds 3 places to arr, and 3 * sizeof(int) = 12 bytes

**Short?**
Let’s say `nelems = 4`. How many bytes beyond `arr` is this?

If it’s an array of...

**Int:** adds 3 places to arr, and $3 \times \text{sizeof(int)} = 12$ bytes

**Short:** adds 3 places to arr, and $3 \times \text{sizeof(short)} = 6$ bytes
Let’s say $\text{nelems} = 4$. How many bytes beyond $\text{arr}$ is this?

If it’s an array of...

**Int:** adds 3 places to $\text{arr}$, and $3 \times \text{sizeof(int)} = 12$ bytes

**Short:** adds 3 places to $\text{arr}$, and $3 \times \text{sizeof(short)} = 6$ bytes

**Char *:** adds 3 places to $\text{arr}$, and $3 \times \text{sizeof(char *)} = 24$ bytes

In each case, we need to know the element size to do the arithmetic.
Let’s write a version of swap_ends that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + nelems - 1, elem_bytes);
}
```

How many bytes past arr should we go to get to the last element?

\[(\text{nelems} - 1) \times \text{elem}_\text{bytes}\]
Let’s write a version of `swap_ends` that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

How many bytes past `arr` should we go to get to the last element?

`(nelems - 1) * elem_bytes`
Let’s write a version of swap_ends that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

But C still can’t do arithmetic with a void*. We need to tell it to not worry about it, and just add bytes. How can we do this?
Let’s write a version of swap_ends that works for any type of array.

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

But C still can’t do arithmetic with a void*. We need to tell it to not worry about it, and just add bytes. **How can we do this?**

char * pointers already add bytes!
You’re asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```
You’re asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```c
int nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```
You’re asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```c
short nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```
You’re asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of anything!

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```c
cchar *strs[] = {"Hi", "Hello", "Howdy"];
size_t nelems = sizeof(strs) / sizeof(strs[0]);
swap_ends(strs, nelems, sizeof(strs[0]));
```
You’re asked to write a function that swaps the first and last elements in an array of numbers. Well, now it can swap for an array of *anything*!

```c
void swap_ends(void *arr, size_t nelems, size_t elem_bytes) {
    swap(arr, (char *)arr + (nelems - 1) * elem_bytes, elem_bytes);
}
```

```c
mystruct structs[] = ...;
size_t nelems = ...;
swap_ends(structs, nelems, sizeof(structs[0]));
```
Demo: Void *s Gone Wrong
Plan For Today

- **Overview**: Generics
- Generic Swap
- Generics Pitfalls
- **Announcements**
- Generic Array Swap
- **Generic Stack**
Stacks

• C generics are particularly powerful in helping us create generic data structures.
• Let’s see how we might go about making a Stack in C.
A Stack is a data structure representing a stack of things.

Objects can be pushed on top of or popped from the top of the stack.

Only the top of the stack can be accessed; no other objects in the stack are visible.

Main operations:

- push(value): add an element to the top of the stack
- pop(): remove and return the top element in the stack
- peek(): return (but do not remove) the top element in the stack
A stack is often implemented using a **linked list** internally.

- "bottom" = tail of linked list
- "top" = head of linked list  \((why \ not \ the \ other \ way \ around?)\)

```c
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);
```

**Problem:** C is not object-oriented! We can’t call methods on variables.
Demo: Int Stack
What modifications are necessary to make a generic stack?
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;

typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;

How might we modify the Stack data representation itself to be generic?
Each node can no longer store the data itself, because it could be any size! Instead, it stores a pointer to the data somewhere else.

```c
typedef struct node {
    struct node *next;
    void *data;
} node;

typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```
int_stack_create

int_stack *int_stack_create() {
    int_stack *s = malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}

How might we modify this function to be generic?

From previous slide:
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```c
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

How might we modify this function to be generic?

---

**From previous slide:**

```c
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;

typedef struct node {
    struct node *next;
    void *data;
} node;
```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

**Problem:** we can no longer pass the data itself as a parameter, because it could be any size! We also cannot copy the data into the node itself.
```c
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

**Solution:** pass a pointer to the data as a parameter instead, and make a heap-allocated copy of it that the node points to.
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data, data, s->elem_size_bytes);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

Solution: pass a pointer to the data as a parameter instead, and make a heap-allocated copy of it that the node points to.
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```c
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

**Problem:** we can no longer return the data itself, because it could be any size!
```c
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

**Solution:** have the caller pass a memory location as a parameter, and copy the data value to that location.
void stack_pop(stack *s, void *addr) {
  if (s->nelems == 0) {
    error(1, 0, "Cannot pop from empty stack");
  }
  node *n = s->top;
  memcpy(addr, n->data, s->elem_size_bytes);
  s->top = n->next;

  free(n->data);
  free(n);
  s->nelems--;
}

**Solution:** have the caller pass a memory location as a parameter, and copy the data value to that location.
Demo: Generic Stack
Plan For Today

- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- **Announcements**
- Generic Array Swap
- Generic Stack

Next time: More Generics, and Function Pointers