Recap

• Other heap allocations
• realloc
Plan for Today

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

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture plan

• **Overview:** Generics
  • Generic Swap
  • Generics Pitfalls
  • Generic Array Swap
COMP201 Topic 5: 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.
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?
Lecture Plan

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

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;
}
```
Swap

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;
}
```
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;
}
```
Swap

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;
}
```

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff14</td>
<td>5</td>
</tr>
<tr>
<td>0xff10</td>
<td>2</td>
</tr>
</tbody>
</table>

\[...\]
Swap

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;
}
```

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff14</td>
<td>5</td>
</tr>
<tr>
<td>0xff10</td>
<td>2</td>
</tr>
</tbody>
</table>

---

1. **Stack Addresses**

   - `x` is stored at `0xff14` with a value of `5`.
   - `y` is stored at `0xff10` with a value of `2`.

2. **Variables in Main Function**

   - `x` is initialized to `2`.
   - `y` is initialized to `5`.
   - `swap_int(&x, &y)` swaps the values of `x` and `y`.
   - `printf("x = %d, y = %d\n", x, y);` prints `x = 5, y = 2`.

3. **Return Statement**

   - `return 0;` returns `0` to indicate successful execution.
Swap

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?”
Swap

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;
}
## Swap

```c
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;
}
```

![Address and Value Diagram](image.png)
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;
}
Swap

```c
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;
}
```

Address | Value
---|---
main() | x 0xff18 0xe
       | y 0xff10 0xe
       | a 0xff10 0xff18
       | b 0xf18 0xff10
temp | 0xf08 0xc
     | 0xf 0'
     | 0xe '5'
     | 0xdf '
     | 0xc '2'
DATA SEGMENT
Swap

```c
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;
}
```

Address  Value

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff18</td>
<td>0xe</td>
</tr>
<tr>
<td>0xff10</td>
<td>0xc</td>
</tr>
<tr>
<td>0xf08</td>
<td>0xc</td>
</tr>
<tr>
<td>0xf10</td>
<td>0xff18</td>
</tr>
<tr>
<td>0xf18</td>
<td>0xff10</td>
</tr>
</tbody>
</table>

<data_segment>

- DATA SEGMENT
  - 0xff08: '2'
  - 0xff0d: '5'
  - 0xff0f: '\0'
</data_segment>
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?”
“Awesome! Thanks. We also have 20 custom struct types. Could you write swap for those too?”

A user-defined structured data type in C (will be covered next week)
Generic Swap

What if we could write one function to swap two values of any single type?

```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) { ... }
...
Generic Swap

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;
}

All 3:
• Take pointers to values to swap
• Create temporary storage to store one of the values
• Move data at b into where a points
• Move data in temporary storage into where b points
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(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
}

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
}

Problem: each type needs to copy a different amount of data!
Generic Swap

```c
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;
```

4 bytes

```
*data2ptr = temp;
```

2 bytes

```
*data2ptr = temp;
```

8 bytes

**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?
Generic Swap

```c
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

```c
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

```c
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
}
Generic Swap

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
}

Let’s start by making space to store the temporary value. How can we make \texttt{nbytes} of temp space?
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 `nbytes` of 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
}

temp is nbytes of memory, since each char is 1 byte!
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
}

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
}

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

`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);
```

`const` is a type qualifier which indicates that the data is read only (will be covered next week).
memcpy

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));  // like x = y
```
**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`).
memmove

When might memmove be useful?

1 2 3 4 5 6 7

4 5 6 7 5 6 7
Generic Swap

```c
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
    temp = *data1ptr; /* temp = *data1ptr */
    // copy data2 to location of data1
    // copy data in temporary storage to location of data2
}

How can `memcpy` or `memmove` help us here?

void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
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.
Generic Swap

```c
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?
Generic Swap

```c
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
}
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);
}

int x = 2;
int y = 5;
swap(&x, &y, sizeof(x));
Generic Swap

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));
Generic Swap

```c
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);
}
```

```c
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));
C Generics

• We can use `void *` and `memcpy` to handle memory as generic bytes.
• If 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);
}
```
Lecture Plan

• **Overview:** Generics
• Generic Swap
• Generics Pitfalls
• Generic Array Swap
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!
Mid-Lecture Check-In

We can now answer the following questions:

1. What variable type represents a “generic pointer”?
2. What variable type can we use to create a specific number of bytes of space on the stack?
3. How can we copy generic memory from one location to another?
4. What is the difference between \texttt{memcpy} and \texttt{memmove}?
5. What are the benefits of generic functions in C? What are the challenges?
Lecture Plan

• **Overview:** Generics
• Generic Swap
• Generics Pitfalls
• Generic Array Swap
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) {
    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
", nums[0], nums[4]);
    return 0;
}
```

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

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?
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!
Swap Ends

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.
Swap Ends

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}?
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...

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

\textbf{Short?}
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...

\begin{itemize}
  \item \textbf{Int}: adds 3 \texttt{places} to \texttt{arr}, and \(3 \times \text{sizeof(int)} = 12\) bytes
  \item \textbf{Short}: adds 3 \texttt{places} to \texttt{arr}, and \(3 \times \text{sizeof(short)} = 6\) bytes
\end{itemize}
Pointer Arithmetic

arr + nelems - 1

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:** adds 3 places to arr, and 3 * sizeof(short) = 6 bytes

**Char **:* adds 3 places to arr, and 3 * 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?

\[(\text{nelems} - 1) \times \text{elem_bytes}\]
Swap Ends

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]));
```
Swap Ends

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
char *strs[] = {"Hi", "Hello", "Howdy"};
size_t nelems = sizeof(strs) / sizeof(strs[0]);
swap_ends(strs, nelems, sizeof(strs[0]));
```
Swap Ends

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

swap_ends.c
Recap

- **void** * is a variable type that represents a generic pointer “to something”.
- We cannot perform pointer arithmetic with or dereference a **void** *.
- We can use **memcpy** or **memmove** to copy data from one memory location to another.
- To do pointer arithmetic with a **void** *, we must first cast it to a **char** *.
- **void** * and generics are powerful but dangerous because of the lack of type checking, so we must be extra careful when working with generic memory.
Recap

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

**Next time:** Function Pointers