

# CS107, Lecture 8

## C Generics – Void \*

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

This document is copyright (C) Stanford Computer Science, Adam Kepler and Olayinka Adekola, licensed under Creative Commons Attribution 2.5 License. All rights reserved.  
Based on slides created by Nick Troccoli and Chris Gregg.



**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 using void \* and avoiding potential pitfalls.
- Before we dive in, remember void \* is just like any other type
  - This means it is just a way of saying:
    - This variable stores the address of an unspecified type of data
    - Information still lives in memory
  - Whatever information is there still has whatever structure it started with

# 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 are used throughout C for common functions like:
  - Sorting
  - Searching
  - Free-ing arbitrary memory
  - And more.
- How can we write generic code in C?



# Imagine

**Lets imagine we need to  
implement some swapping for our  
team or a client**

# Lecture Plan

- **Generic Swap**
- Generics Pitfalls
- Generic Swap Ends
- Generic Stack
- Array Rotation

# Swap

You're asked to write a function that swaps two numbers.

```
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.

```
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;  
}
```

main()



		Stack
		Value
Address		
		...
x	0xff14	2
y	0xff10	5
		...



# Swap

You're asked to write a function that swaps two numbers.

```
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;  
}
```

main()

swap\_int()

		Stack
Address		Value
		...
x	0xff14	2
y	0xff10	5
		...
b	0xf18	0xff10
a	0xf10	0xff14
		...

# Swap

You're asked to write a function that swaps two numbers.

```
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;  
}
```

main()

swap\_int()

		Stack
		Address
		Value
		...
x	0xff14	2
y	0xff10	5
		...
b	0xf18	0xff10
a	0xf10	0xff14
temp	0xf0c	2
		...

# Swap

You're asked to write a function that swaps two numbers.

```
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;
}
```

main()

swap\_int()

		Stack
		Address
		Value
		...
x	0xff14	5
y	0xff10	5
		...
b	0xf18	0xff10
a	0xf10	0xff14
temp	0xf0c	2
		...

# Swap

You're asked to write a function that swaps two numbers.

```
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;
}
```

main()

swap\_int()

		Stack
		Value
Address		
		...
x	0xff14	5
y	0xff10	2
		...
b	0xf18	0xff10
a	0xf10	0xff14
temp	0xf0c	2
		...

# Swap

You're asked to write a function that swaps two numbers.

```
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;  
}
```

main()



		Stack
		Value
Address		
		...
x	0xff14	5
y	0xff10	2
		...

**“Oh, when I said ‘numbers’  
I meant shorts, not ints.”**



# Swap

```
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;  
}
```

main()

swap\_short()

		Stack
		Address
		Value
		...
x	0xff12	2
y	0xff10	5
		...
b	0xf18	0xff10
a	0xf10	0xff12
temp	0xf0e	2
		...

The diagram illustrates the stack memory layout. The 'main()' function's stack frame contains variables x (address 0xff12, value 2) and y (address 0xff10, value 5). The 'swap\_short()' function's stack frame contains variables b (address 0xf18, value 0xff10), a (address 0xf10, value 0xff12), and temp (address 0xf0e, value 2). Red arrows show the swap of values between b and a, indicating that the values of x and y are being swapped via pointers.

**“You know what, I goofed.  
We’re going to use strings.  
Could you write something  
to swap those?”**

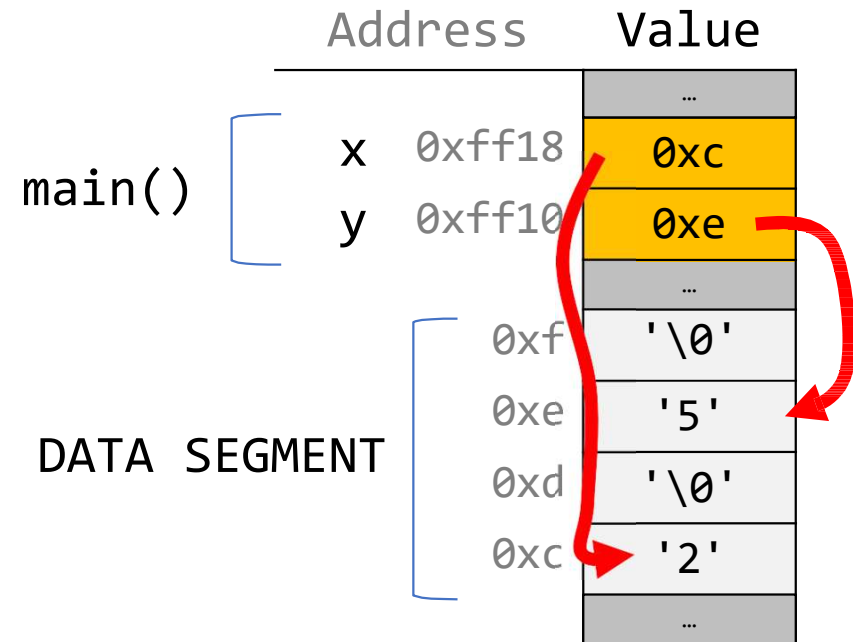




# Swap

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

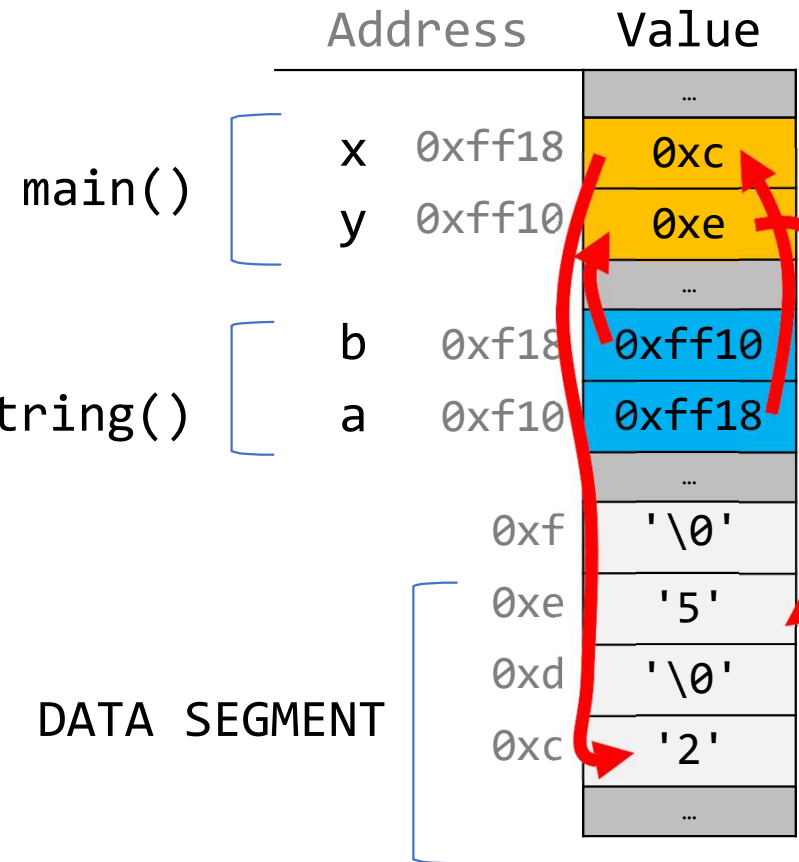
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



# Swap

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

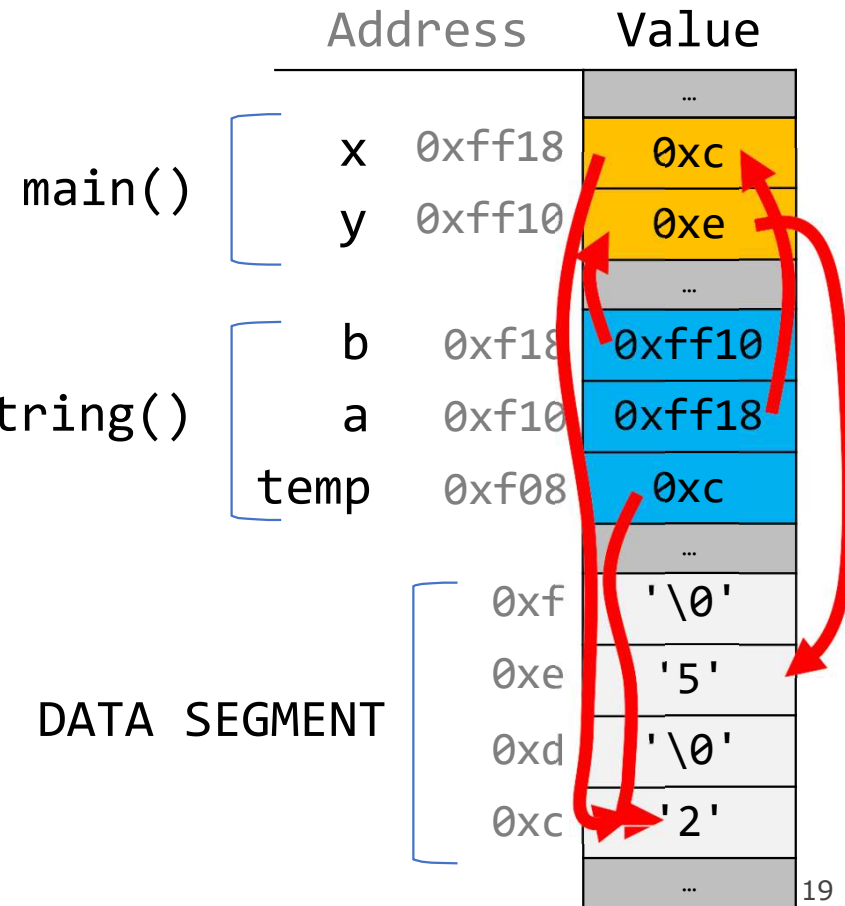
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



# Swap

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

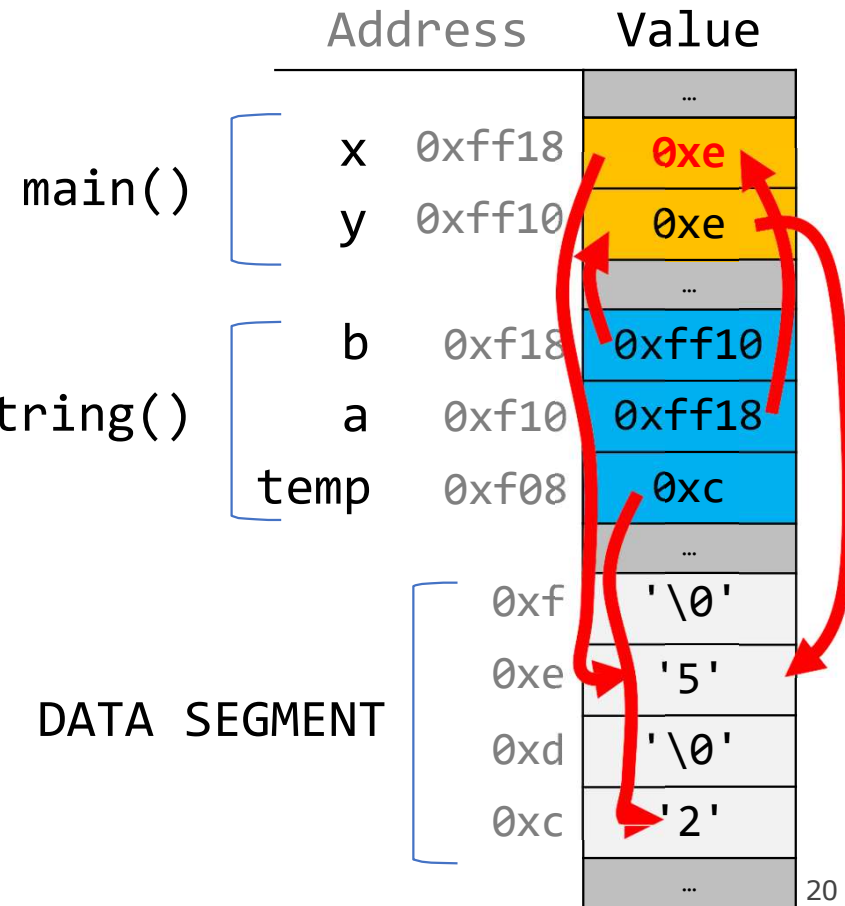
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



# Swap

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

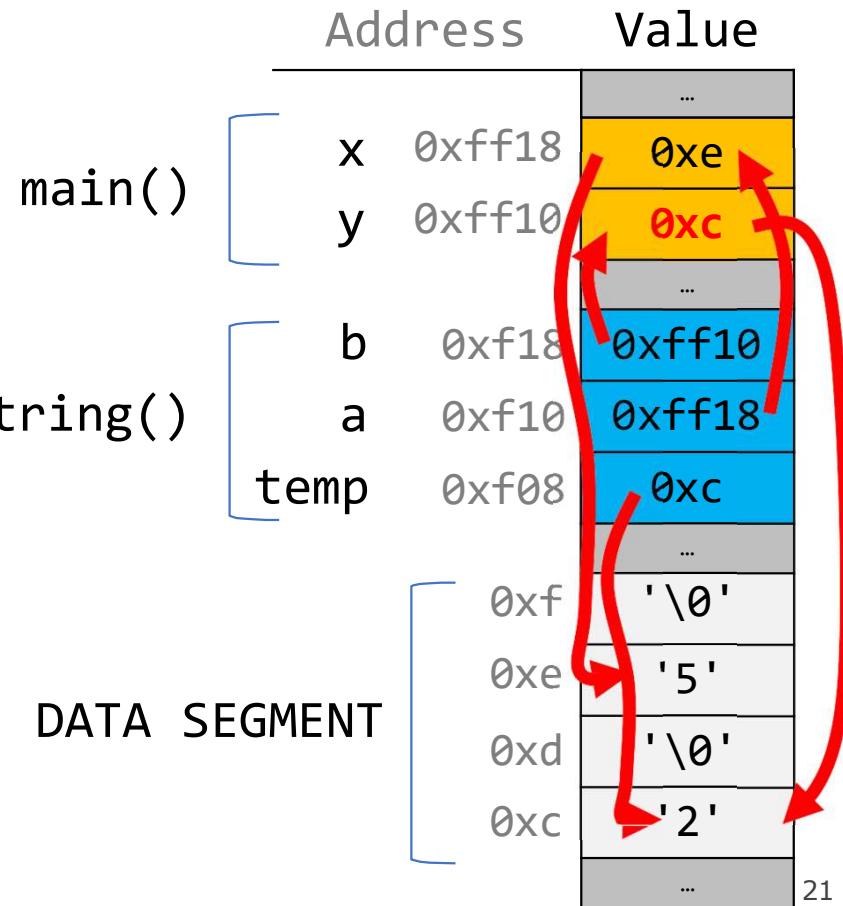
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



# Swap

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

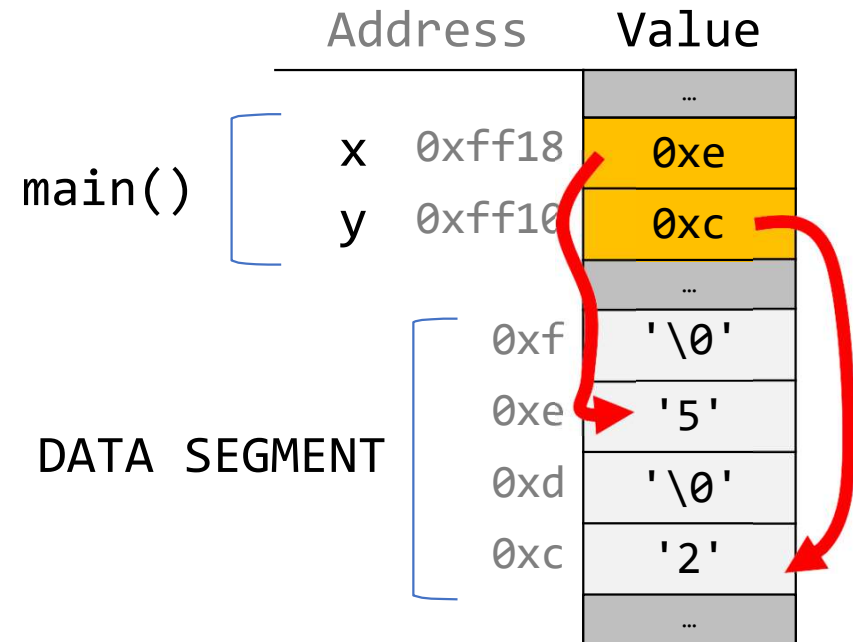
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



# Swap

```
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?”**





# Generic Swap

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

```
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;
}
```

# 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;  
}
```

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

# 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  
}
```

# 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  
}
```

```
int temp = *data1ptr;
```

4 bytes

```
short temp = *data1ptr;
```

2 bytes

```
char *temp = *data1ptr;
```

8 bytes

**Problem:** each type may need a different size temp!

# 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  
}
```

`*data1Ptr = *data2ptr;` 4 bytes

`*data1Ptr = *data2ptr;` 2 bytes

`*data1Ptr = *data2ptr;` 8 bytes

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

# 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  
}
```

```
*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

```
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  
}
```

# 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  
}
```

# 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  
}
```

# 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  
}
```

If we don't know the data type, we don't know how many bytes it is. Let's take that as another parameter.

# 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.

# 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  
}
```

Let's start by making space to store the temporary value. How can we make **nbytes** of temp space?



# Generic Swap

```
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?

# Generic Swap

```
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!

# Generic Swap

```
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**?

# Generic Swap

```
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**?

# Generic Swap

```
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.

```
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.

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

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

# memmove

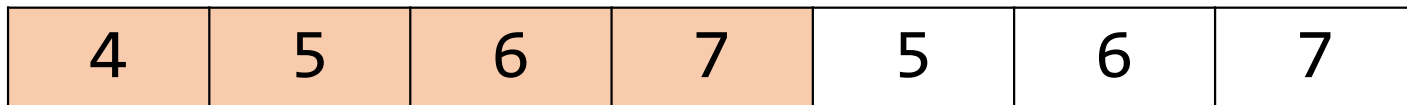
**memmove** is the same as `memcpy`, but supports overlapping regions of memory. (Unlike its name implies, it still “copies”).

```
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?





# void \*, memcpy, memmove

From a design standpoint, why does **memcpy** take **void \***s as parameters?

```
int x = 2;  
int y = 3;  
memcpy(&x, &y, sizeof(x)); // copy 3 into x
```

```
// why not this?
```

```
memcpy(x, y);
```

1. The first parameter must be a pointer so **memcpy** knows where to copy to.
2. The second parameter *could* be a non-pointer. But then there must be a version of **memcpy** for every possible type we would like to copy!

```
memcpy_i(void *, int); memcpy_c(void *, char); memcpy_d(void *, double);
```

# 4 Copies - Which To Choose?

## **strcpy (string copy):**

- Used for copying null-terminated strings
- Copies from source to destination until it encounters a null terminator
- Does not check buffer sizes, which can lead to buffer overflows if not used carefully

## **strncpy (string numbered copy):**

- Similar to strcpy, but requires specifying a maximum number of characters to copy
- If the source string is shorter than the specified length, it pads the destination with null characters
- Does not guarantee null-termination if the source string is longer than the specified length

## **memcpy (memory copy):**

- Used for copying a specified number of bytes from one memory location to another
- Can be used for any type of data, not just strings
- Does not check for overlap between source and destination memory regions

## **memmove (memory move):**

- Similar to memcpy, but designed to handle overlapping memory regions safely
- Copies data to a temporary buffer first if source and destination overlap
- Generally slower than memcpy but safer when dealing with overlapping memory

# Generic Swap

```
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.

# Generic Swap

```
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
}
```

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);
```

# 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  
    // copy data in temporary storage to location of data2  
}
```

# 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  
    // 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

```
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?

# 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  
    *data1ptr = *data2ptr; ???  
    // copy data in temporary storage to location of data2  
}
```

How can we copy data2 to the location of data1?



# 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  
}
```

How can we copy data2 to the location of data1?  
**memcpy!**

# 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  
}
```

How can we copy temp's data to the location of data2?

# 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);  
}
```

How can we copy temp's data to the location of data2? **memcpy!**

# 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);  
}
```

```
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

```
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));
```

# 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);  
}
```

```
mystruct x = {...};  
mystruct y = {...};  
swap(&x, &y, sizeof(x));
```

# void \*, memcpy, memmove

- 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!

```
// even more robust (handles overlapping swap pointers)
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {
    char temp[nbytes];
    memcpy(temp, data1ptr, nbytes);
    memmove(data1ptr, data2ptr, nbytes);
    memcpy(data2ptr, temp, nbytes);
}
```



# Lecture Plan

- Generic Swap
- **Generics Pitfalls**
- Generic Swap Ends
- Generic Stack
- Array Rotation

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

# 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!*)

# 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 Frankensteins!



<http://i.ytimg.com/vi/10gPoYjq3EA/hqdefault.jpg>

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

# Code Sample: Void \*s Gone Wrong



```
swap.c
```

# Lecture Plan

- Generic Swap
- Generics Pitfalls
- **Generic Swap Ends**
- Generic Stack
- Array Rotation

# Swap Ends

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

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

Wait – we just wrote a generic swap function. Let's use that!

```
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;  
}
```

# Swap Ends

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

```
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!

# Swap Ends

Let's write out what some other versions would look like (just in case).

```
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!



# Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

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

Is this generic? Does this work?

# Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
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.

```
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.

```
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

`arr + nelems - 1`

Let's say `nelems = 4`. How many bytes into/beyond `arr` is this?

If it's an array of...

**Int?**

# 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  $3 * \text{sizeof}(\text{int}) = 12$  bytes

# Pointer Arithmetic

`arr + nelems - 1`

Let's say `nelems = 4`. How many bytes into/beyond `arr` is this?

If it's an array of...

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

**Short?**

# Pointer Arithmetic

`arr + nelems - 1`

Let's say `nelems = 4`. How many bytes into/beyond `arr` is this?

If it's an array of...

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

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



# Pointer Arithmetic

`arr + nelems - 1`

Let's say `nelems = 4`. How many bytes into/beyond `arr` is this?

If it's an array of...

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

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

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

**In each case, we need to know the element size to do the arithmetic.**

# Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
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?

**`(nelems - 1) * elem_bytes`**

# Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
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`**

# Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
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?**

# Swap Ends

Let's write a version of `swap_ends` that works for any type of array.

```
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!

# Swap Ends

Well, now it can swap\_ends for an array of anything!

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

We can do pointer arithmetic with a **void \*** pointer by casting it.

# Swap Ends

Lets see some examples!

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

```
int nums[] = {5, 2, 3, 4, 1};  
size_t nelems = sizeof(nums) / sizeof(nums[0]);  
swap_ends(nums, nelems, sizeof(nums[0]));
```

# Swap Ends

Lets see some examples!

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

```
short nums[] = {5, 2, 3, 4, 1};  
size_t nelems = sizeof(nums) / sizeof(nums[0]);  
swap_ends(nums, nelems, sizeof(nums[0]));
```



# Swap Ends

Lets see some examples!

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

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

# Swap Ends

Lets see some examples!

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

```
mystruct structs[] = ...;  
size_t nelems = ...;  
swap_ends(structs, nelems, sizeof(structs[0]));
```

# Lecture Plan

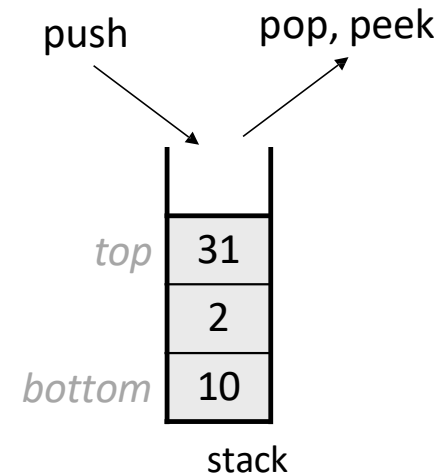
- Generic Swap
- Generics Pitfalls
- Generic Swap Ends
- **Generic Stack**
- Array Rotation

# 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.

# Refresher: Stacks

- 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

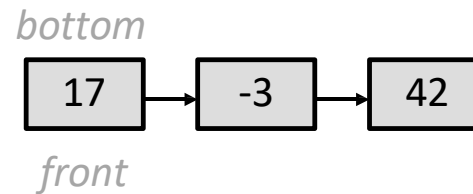


# Refresher: Stacks

A stack is often implemented using a **linked list** internally.

- "bottom" = tail of linked list
- "top" = head of linked list

```
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.

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

# Code Sample: Int Stack



```
int_stack.c
```



**What modifications are  
necessary to make a  
generic stack?**



# Stack Structs

```
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?

# Stack Structs

```
typedef struct int_node {  
    struct int_node *next;  
    int data;  
} int_node;
```

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

**Problem:** each node can no longer store the data itself, because it could be any size!

# Generic Stack Structs

```
typedef struct int_node {  
    struct int_node *next;  
    void *data;  
} int_node;
```

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

**Solution:** each node stores a pointer, which is always 8 bytes, to the data somewhere else. We must also store the data size in the Stack struct.

# Stack Functions

- **int\_stack\_create()**: creates a new stack on the heap and returns a pointer to it
- **int\_stack\_push(int\_stack \*s, int data)**: pushes data onto the stack
- **int\_stack\_pop(int\_stack \*s)**: pops and returns topmost stack element

# 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;
```

# Generic stack\_create

```
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;  
}
```

# int\_stack\_push

```
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:

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

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

# Generic stack\_push

```
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 1:** we can no longer pass the data itself as a parameter, because it could be any size!



# Generic stack\_push

```
void int_stack_push(int_stack *s, const void *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 1:** pass a pointer to the data as a parameter instead.

# Generic stack\_push

```
void int_stack_push(int_stack *s, const void *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 2:** we cannot copy the existing data pointer into new\_node. The data structure must manage its own copy that exists for its entire lifetime. The provided copy may go away!

# Generic stack\_push

```
int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
    // now stack stores pointer to invalid memory for 7!
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
```

# Generic stack\_push

```
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 2:** make a heap-allocated copy of the data that the node points to.

# int\_stack\_pop

```
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;
}
```

How might we modify this function to be generic?

From previous slide:

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

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

# Generic stack\_pop

```
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!

# Generic stack\_pop

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

While it's possible to return the heap address of the element, this means the client would be responsible for freeing it. Ideally, the data structure should manage its own memory here.

# Generic stack\_pop

```
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 to that location.



# Using Generic Stack

```
int_stack *intstack = int_stack_create();  
for (int i = 0; i < TEST_STACK_SIZE; i++) {  
    int_stack_push(intstack, i);  
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

# Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));  
for (int i = 0; i < TEST_STACK_SIZE; i++) {  
    stack_push(intstack, &i);  
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

# Using Generic Stack

```
int_stack *intstack = int_stack_create();  
int_stack_push(intstack, 7);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

# Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));  
int num = 7;  
stack_push(intstack, &num);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

# Using Generic Stack

```
// Pop off all elements
while (intstack->nelems > 0) {
    printf("%d\n", int_stack_pop(intstack));
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

# Using Generic Stack

```
// Pop off all elements
int popped_int;
while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
    printf("%d\n", popped_int);
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

# Code Sample: Generic Stack



```
generic_stack.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.



# Lecture Plan

- Generic Swap
- Generics Pitfalls
- Generic Swap Ends
- Generic Stack
- **Array Rotation**

# Exercise: Array Rotation

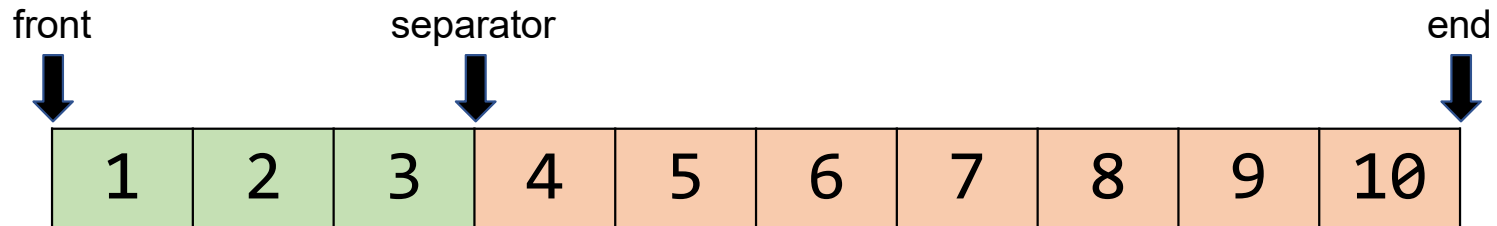
**Exercise:** You're asked to provide an implementation for a function called **rotate** with the following prototype:

```
void rotate(void *front, void *separator, void *end);
```

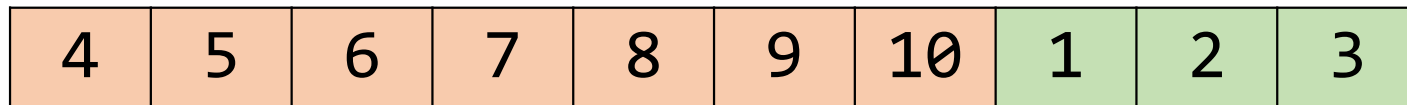
The expectation is that **front** is the base address of an array, **end** is the past-the-end address of the array, and **separator** is the address of some element in between. **rotate** moves all elements in between **front** and **separator** to the end of the array, and all elements between **separator** and **end** move to the front.

# Exercise: Array Rotation

```
int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
rotate(array, array + 3, array + 10);
```

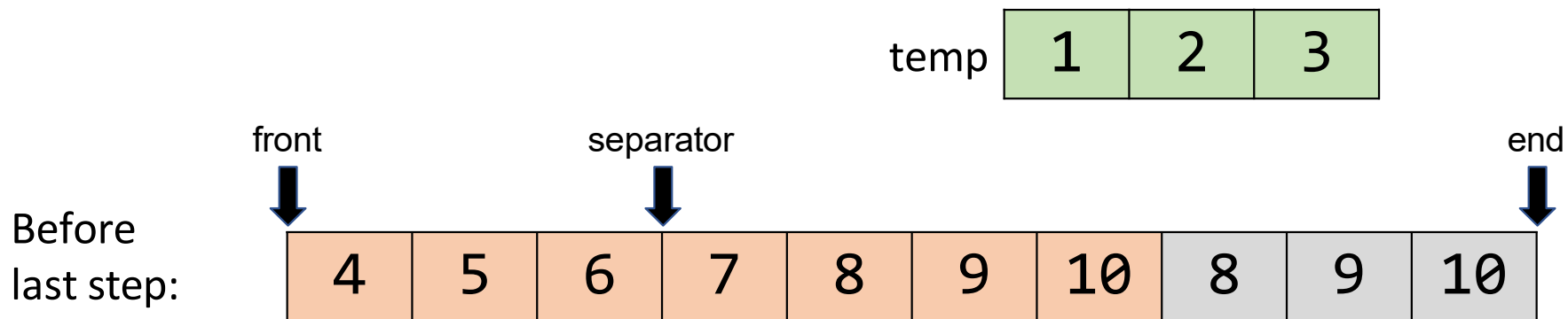
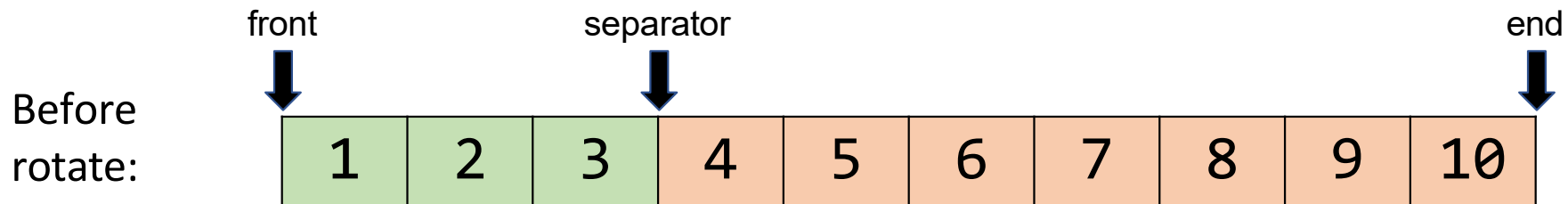


Before:



After:

# The inner workings of rotate



# Exercise: Array Rotation

**Exercise:** Implement `rotate` to generate the provided output.

```
int main(int argc, char *argv[]) {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print_int_array(array, 10); // intuit implementation 😊
    rotate(array, array + 5, array + 10);
    print_int_array(array, 10);
    rotate(array, array + 1, array + 10);
    print_int_array(array, 10);
    rotate(array + 4, array + 5, array + 6);
    print_int_array(array, 10);
    return 0;
}
```

Output:

```
myth52:~/lect8$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
myth52:~/lect8$
```

```
cp -r /afs/ir/class/cs107/lecture-code/lect8 .
```

# Code Sample: Array Rotation



```
rotate.c
```



# Exercise: Array Rotation

**Exercise:** A properly implemented **rotate** will prompt the following program to generate the provided output.

And here's that properly implemented function!

```
void rotate(void *front, void *separator, void *end) {
    int width = (char *)end - (char *)front;
    int prefix_width = (char *)separator - (char *)front;
    int suffix_width = width - prefix_width;

    char temp[prefix_width];
    memcpy(temp, front, prefix_width);
    memmove(front, separator, suffix_width);
    memcpy((char *)end - prefix_width, temp, prefix_width);
}
```