CS107 Lecture 8

Generics in C: Leveraging the `void *`

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

Why We ❤️ The Stack

- **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 fast**. The stack segment is fully configured before **`main`** is called.
- **It is safer**. Local variables are typed, so the compiler has better information about what operations are legal and which ones aren't.

Why We ❤️ The Heap

- **Memory abounds!** By default, the stack is relatively small—on the order of 8 MBs. The heap, however, can more easily grow to accommodate much larger allocation requests.
- **Allocations are resizable.** If a dynamically allocated figure turns out to be too small, that figure can be resized using **`realloc`**.
- **Scope.** The memory is not cleaned up when its function exits; instead, you control when the memory is freed.

Stack and Heap

- Rule of thumb: unless a situation truly benefits from dynamic allocation, you should bias towards the stack.
- Dynamic memory allocation is generally necessary when:
  - the amount of memory needed can’t be determined until the program is running
  - the memory must persist beyond the lifetime of the function that allocated it

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 successfully use the **`void *`** even though they have no type information about what they point to.

Plan For Today

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

```
cp -r /afs/ir/class/cs107/samples/lectures/lect8 .
```

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

Swapping `int`s

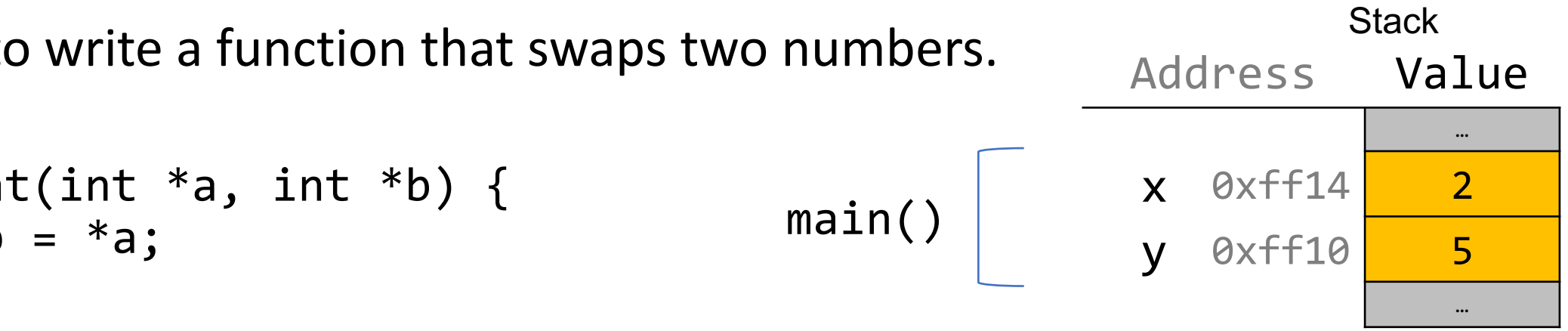
You're asked to write a function that swaps two numbers.

```
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);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

Swapping `int`s

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

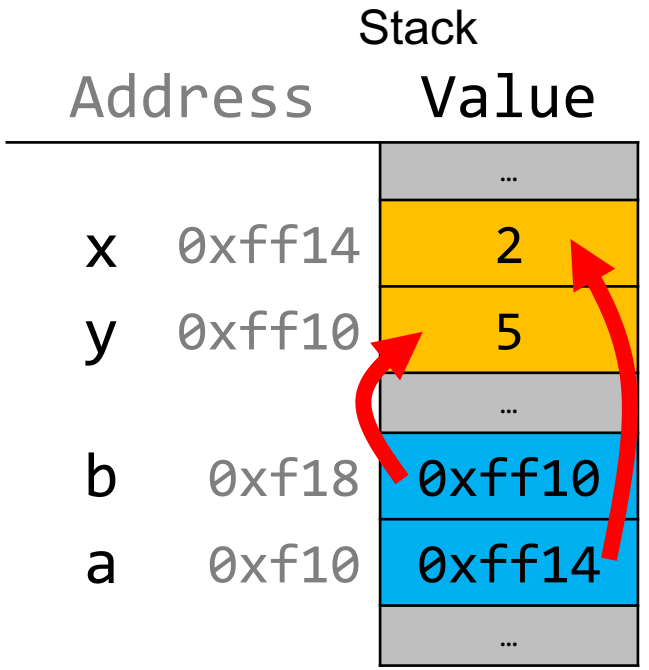
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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

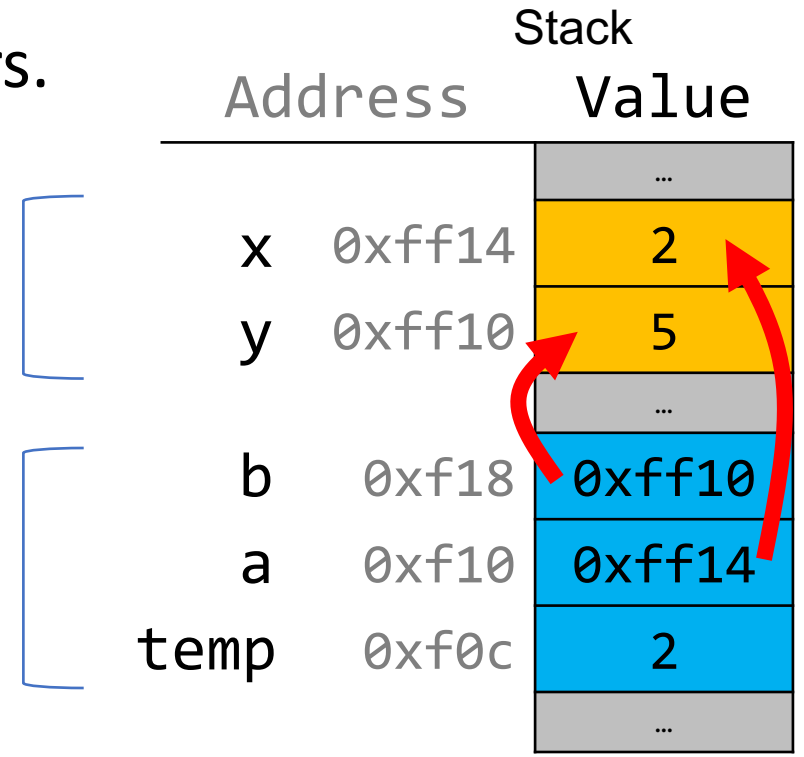
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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

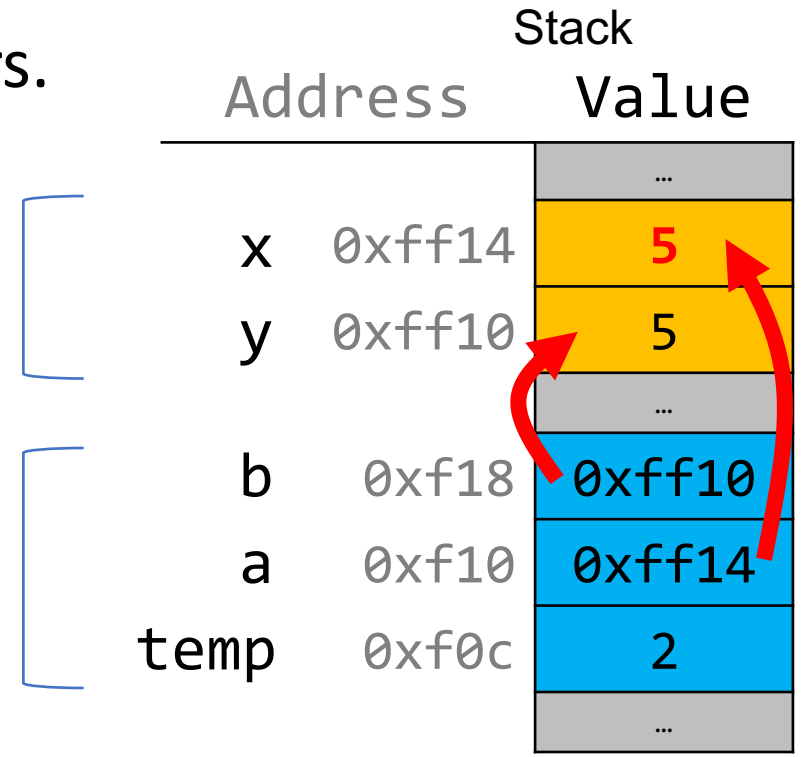
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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

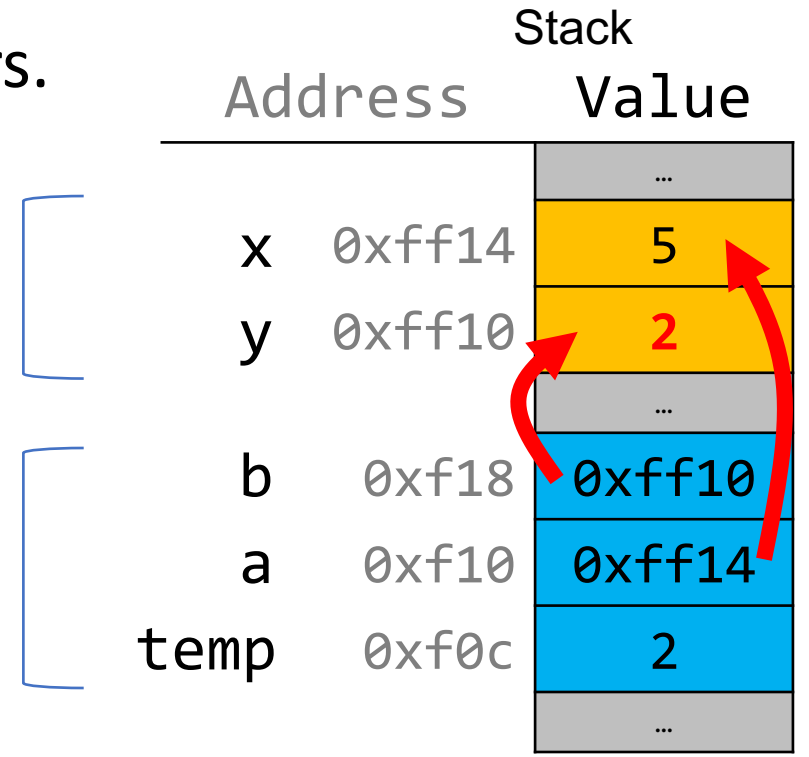
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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

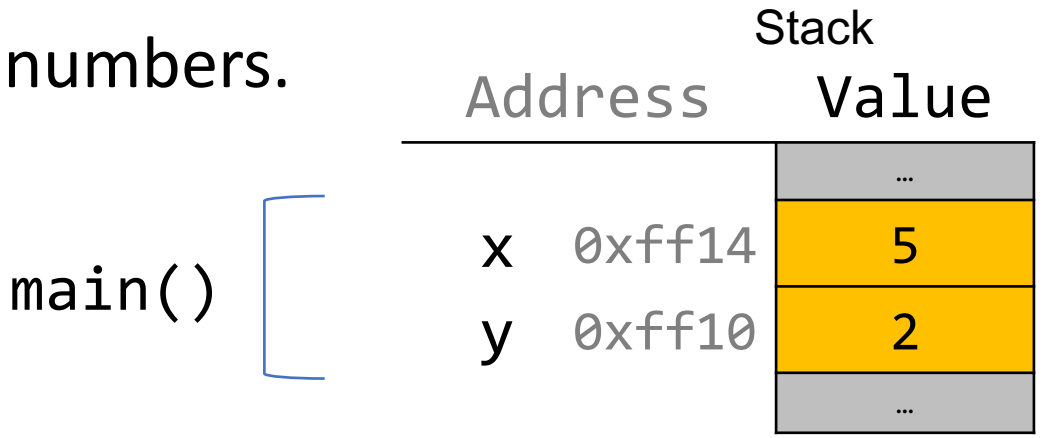
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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

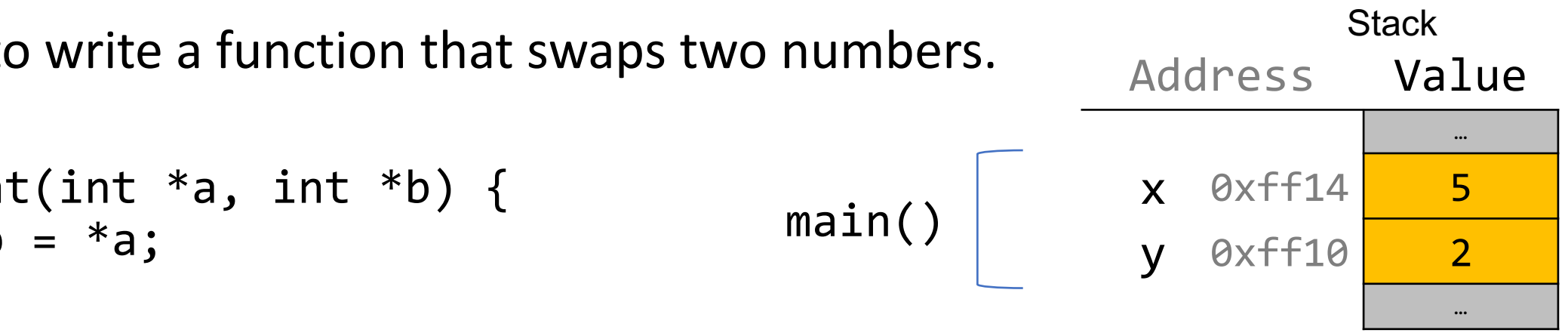
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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

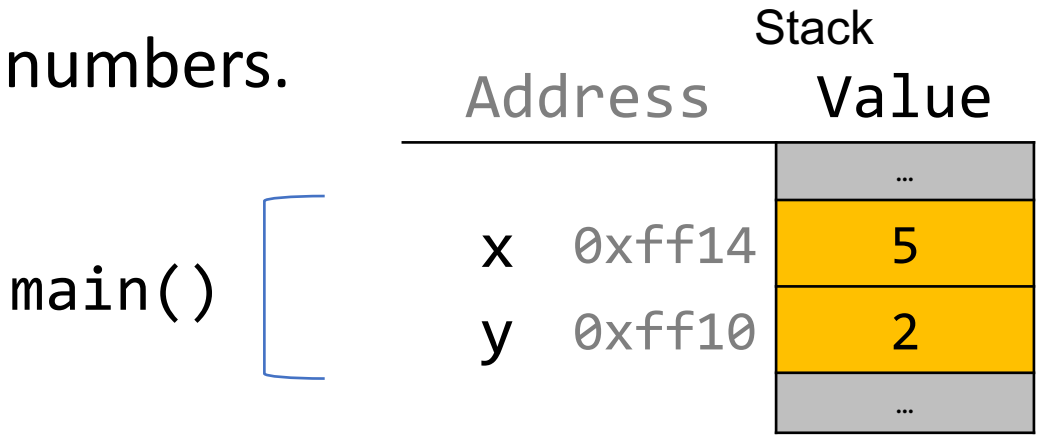
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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



Swapping `int`s

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);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

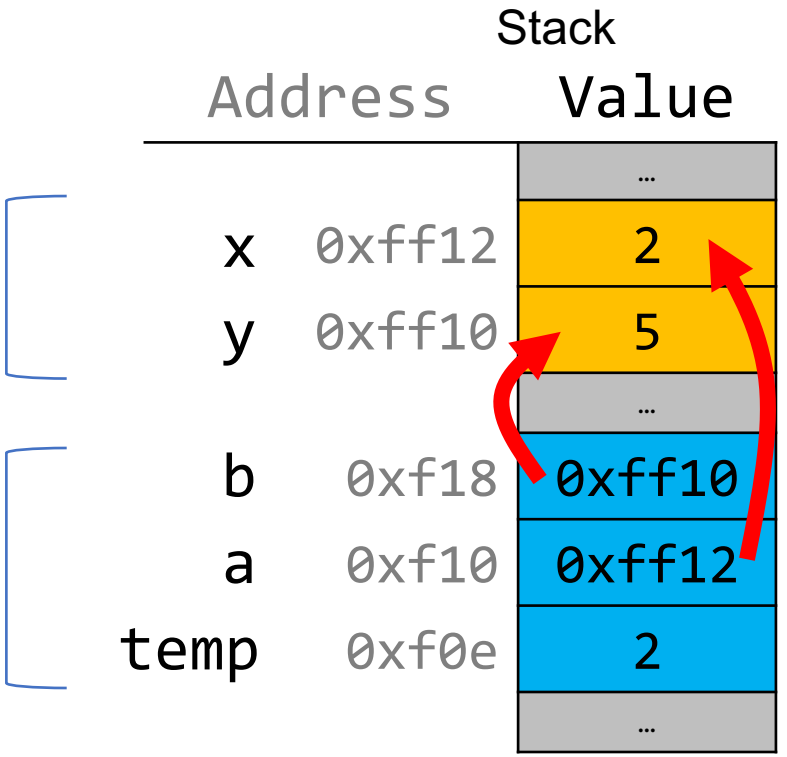


Swapping shorts

```
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);
    printf("x = %hd, y = %hd\n", x, y);
    return 0;
}
```

Swapping shorts

```
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);
    printf("x = %hd, y = %hd\n", x, y);
    return 0;
}
```

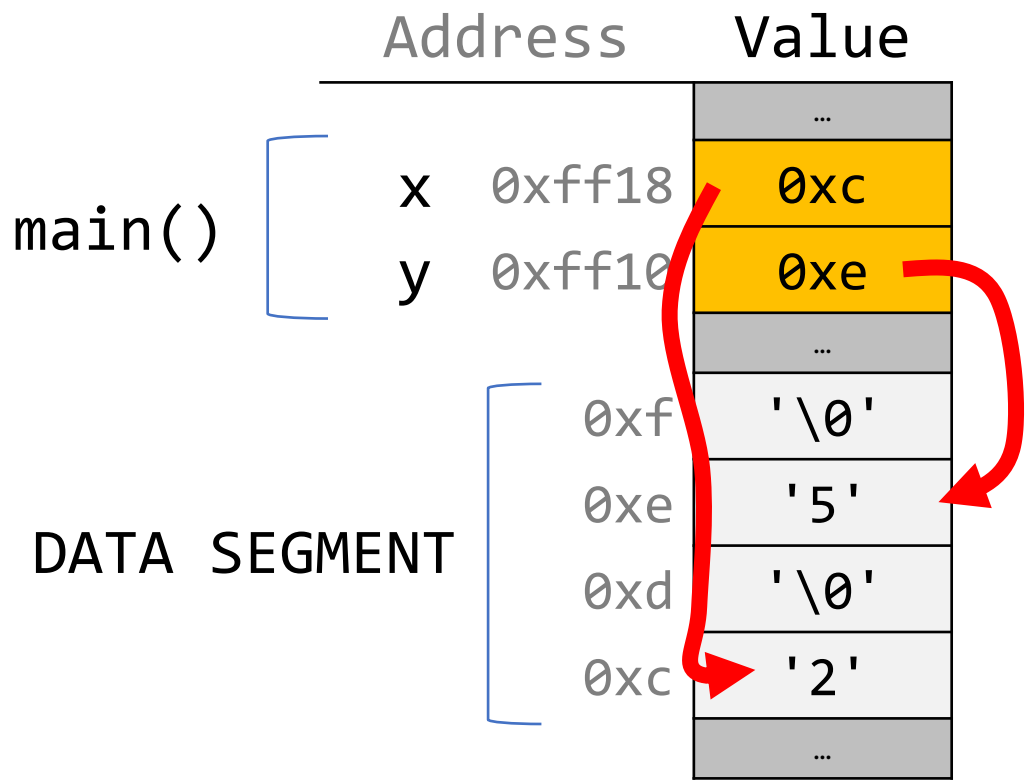


Swapping C Strings

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```

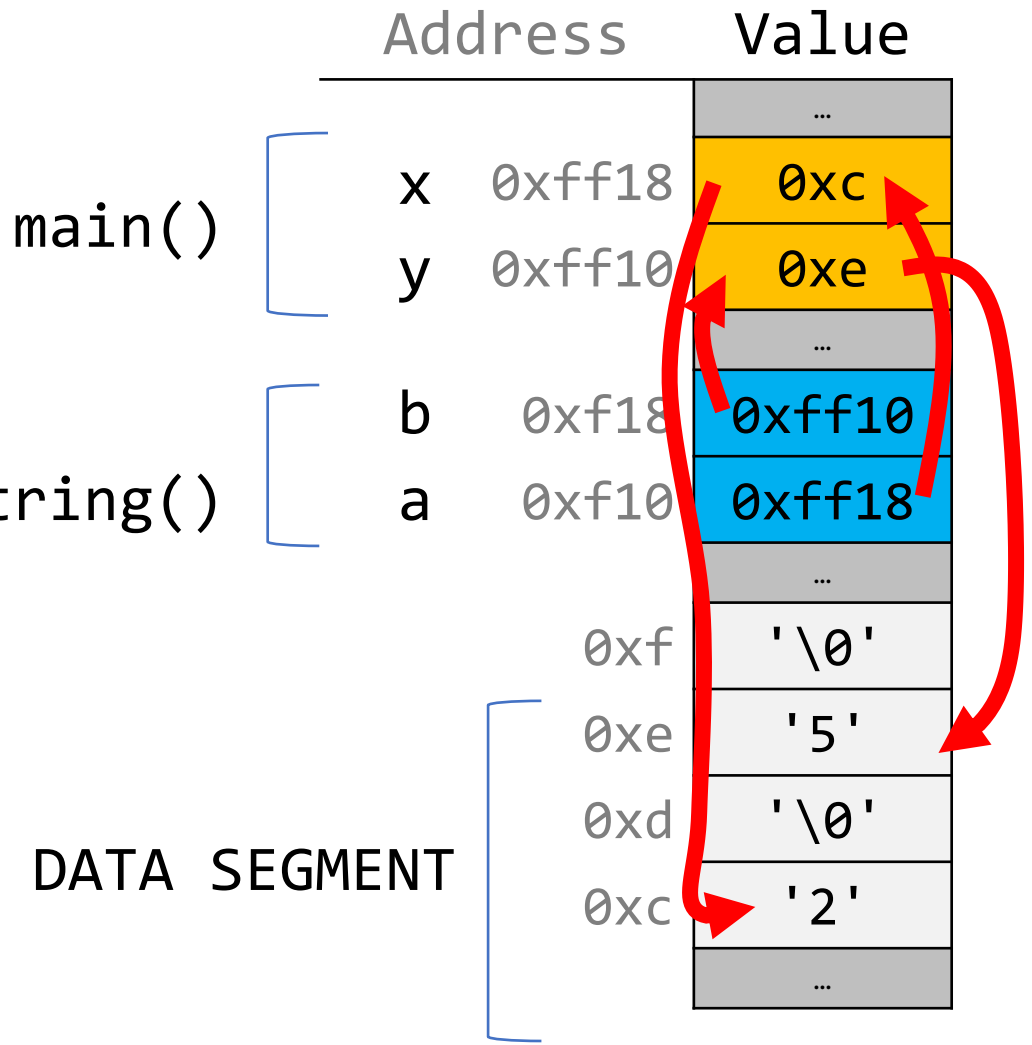
Swapping C Strings

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



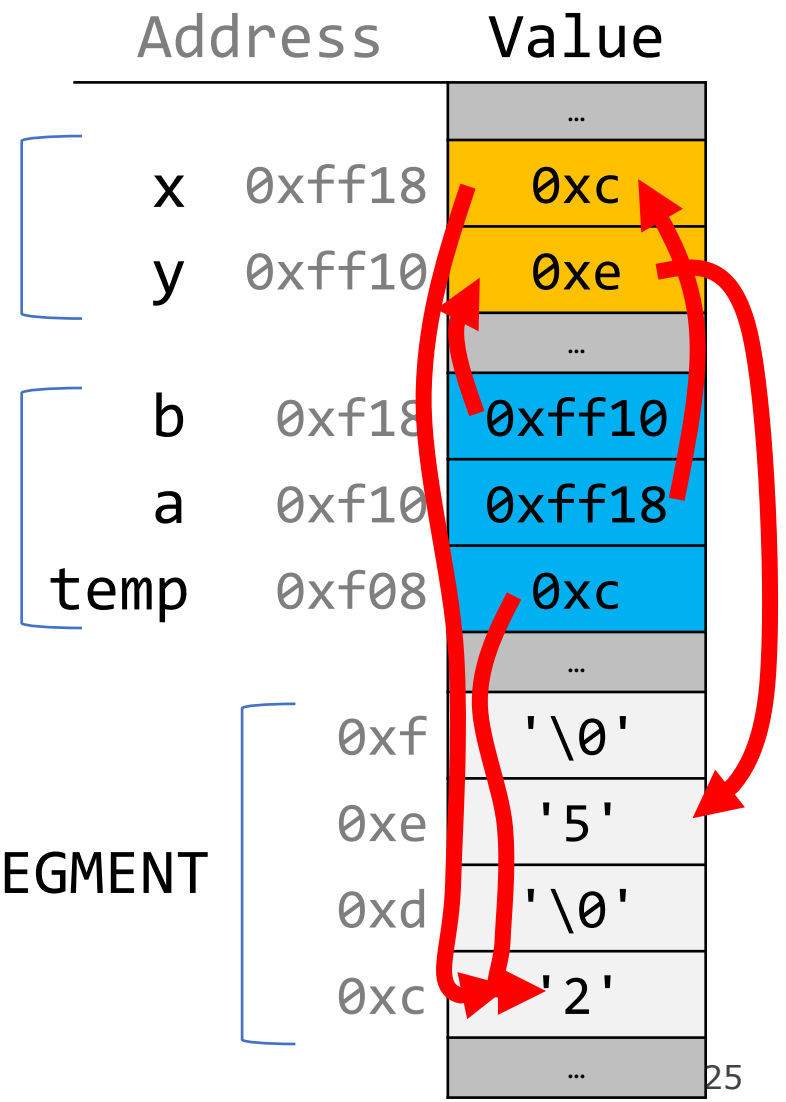
Swapping C Strings

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



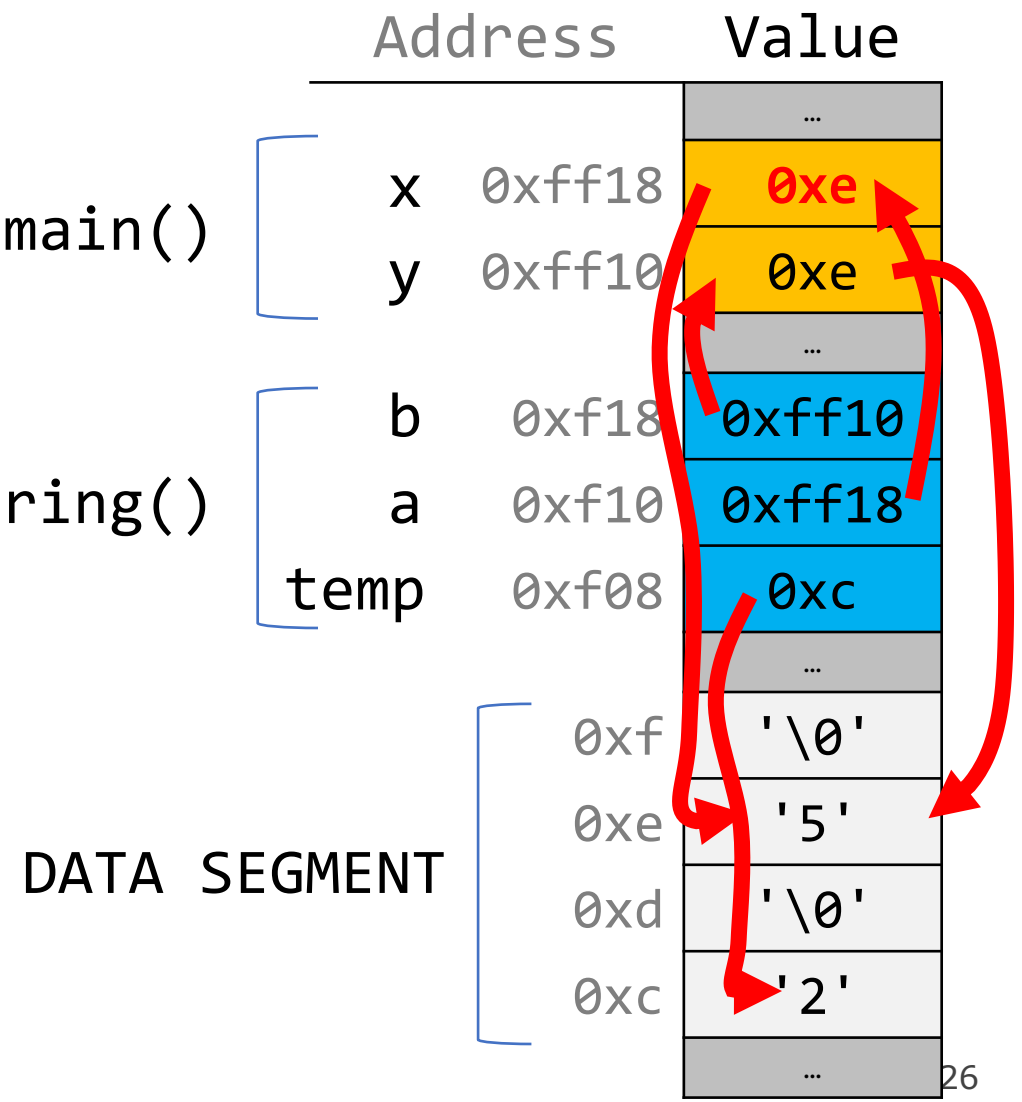
Swapping C Strings

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



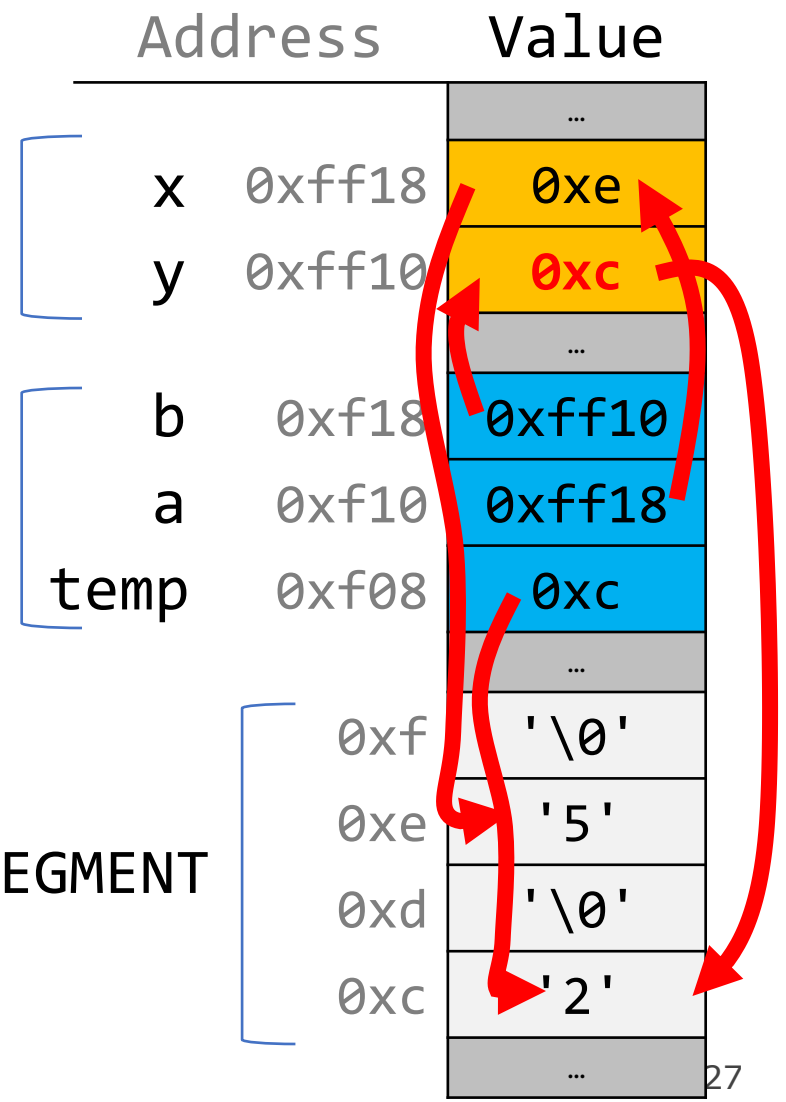
Swapping C Strings

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



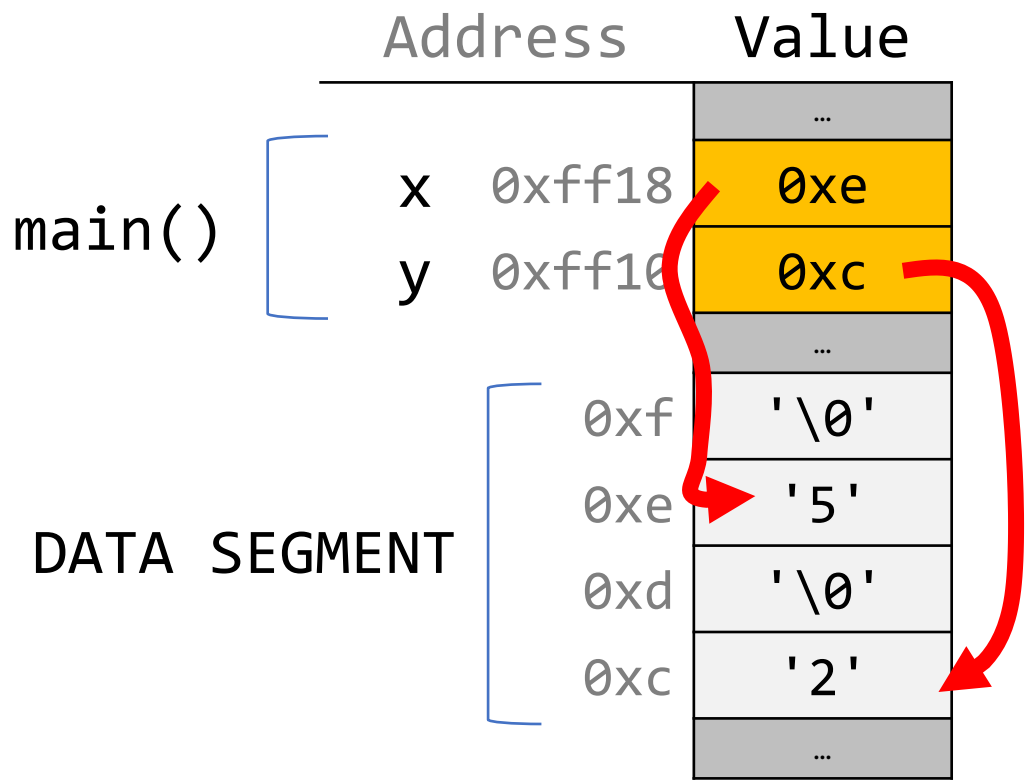
Swap

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



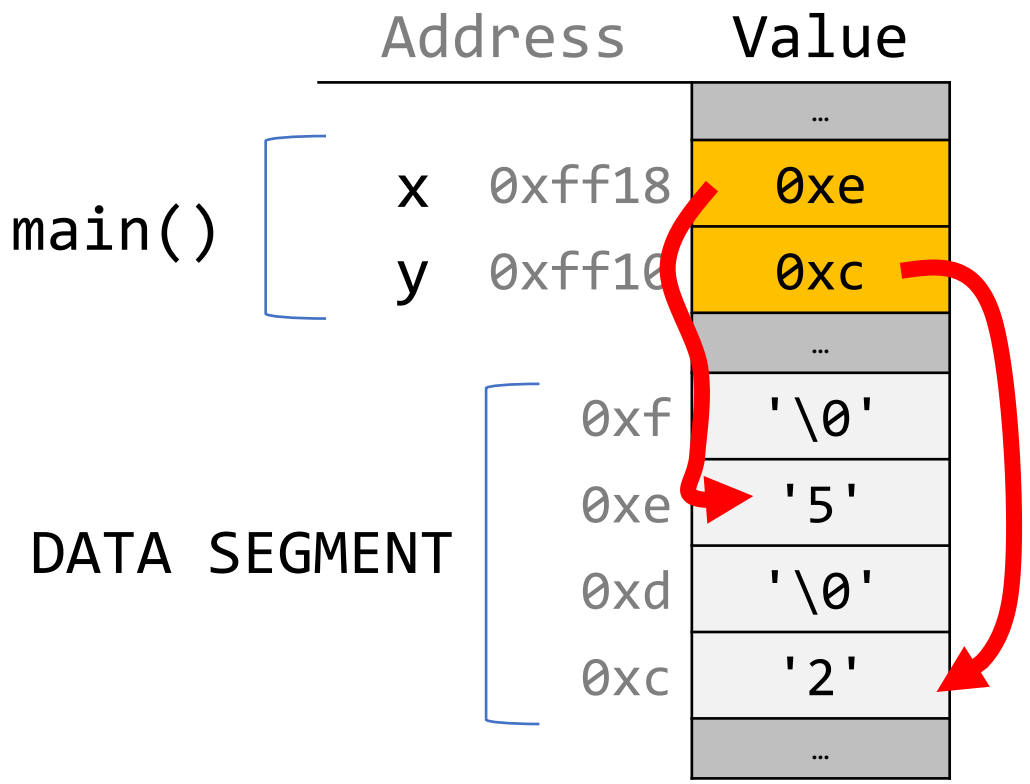
Swap

```
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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



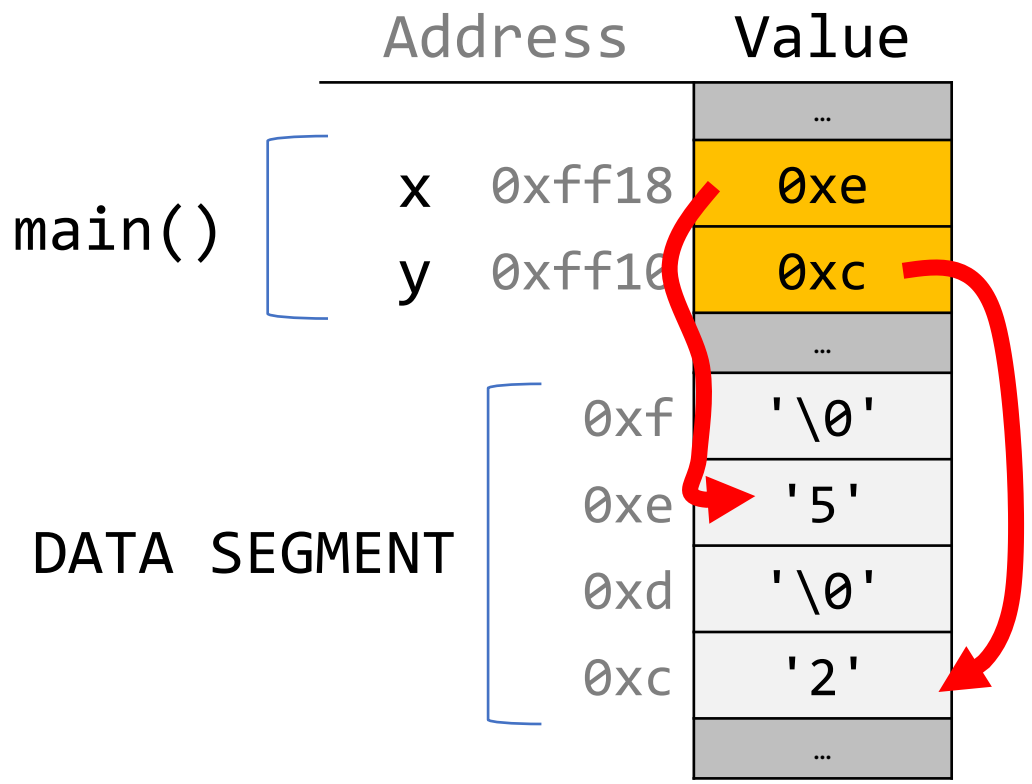
Swap

```
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);
    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);
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



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 the variables are strongly typed.

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. (It assumes the two figures don't overlap.)

```
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**).

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

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

memmove

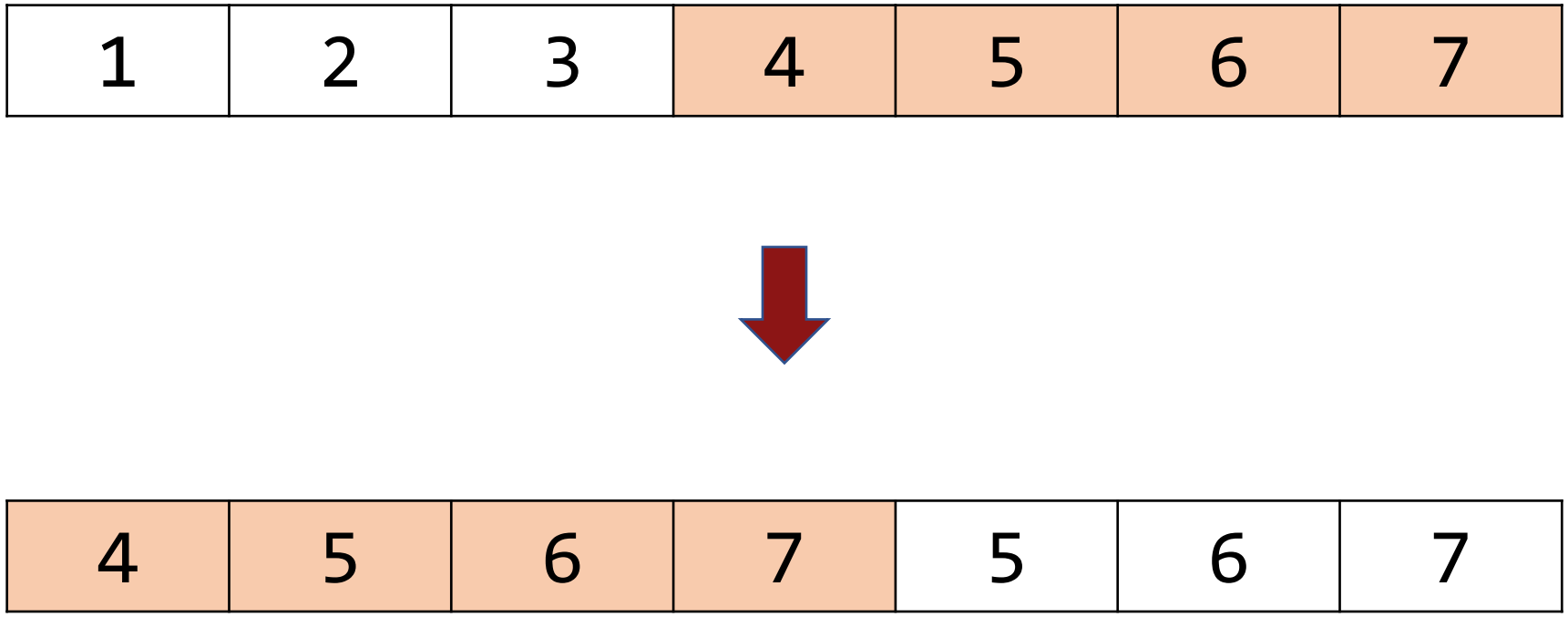
memmove is like **memcpy**, except that it supports overlapping memory regions.

```
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**, tough the return value is generally ignored.)

memmove

When might **memmove** be useful?



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));
```

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!

```
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 any type checking!
- E.g. with **int**, C would never let you swap *half* of an int. With **void *s**, this can happen!

Plan For Today

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

Announcements

- assign3
  - 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. Restated, if **`assert`** ends your program because it caught a legitimate runtime error, you're off the hook

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 **`memcpy`** and **`memmove`**?
5. What are the benefits of generic functions in C? What are the challenges?

Plan For Today

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

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;
}

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);
    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.

```
void swap_ends_int(int arr[], size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(arr[0]));
}

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);
    printf("nums[0] = %d, nums[4] = %d\n", nums[0], nums[4]);
    return 0;
}
```

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[0]));
}

void swap_ends_short(short arr[], size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(arr[0]));
}

void swap_ends_string(char *arr[], size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(arr[0]));
}

void swap_ends_float(float arr[], size_t nelems) {
    swap(arr, arr + nelems - 1, sizeof(arr[0]));
}
```

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. We don't know the element size. And pointer arithmetic depends on the type of data being addressed. With **`void`** `*`s, 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. This, however, is still not quite right yet.

Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

`ints`?

Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

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

Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

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

`shorts`?

Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

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

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

Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of…

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

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

`char *s`: 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.

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

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!

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

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!

```
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

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!

```
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

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!

```
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

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!

```
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]));
```

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

Exercise: A properly implemented **rotate** will prompt the following program 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$
```

Demo: 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) {
    size_t width = (char *) end - (char *) front;
    size_t prefix_width = (char *) separator - (char *) front;
    size_t 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);
}
```

Plan For Today

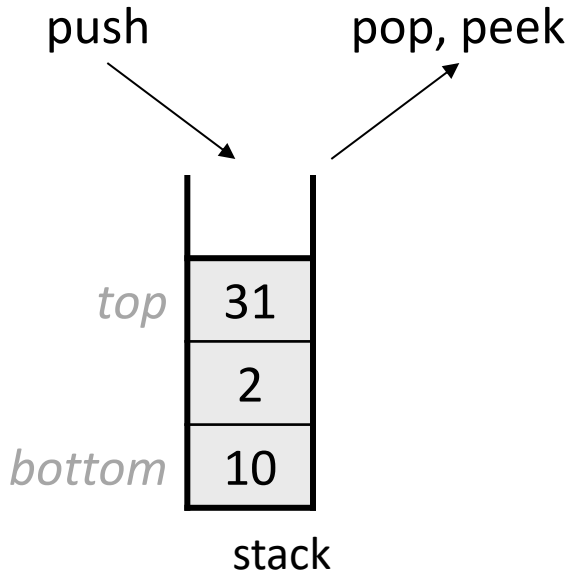
- **Overview:** Generics
- Generic Swap
- Generics Pitfalls
- **Announcements**
- Generic Array Swap
- **Generic Stack**

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.

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 items are immediately visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack

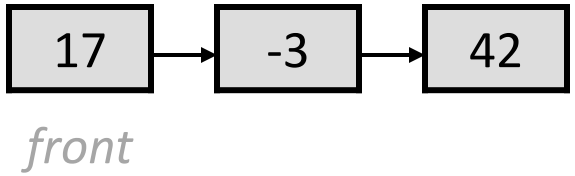


Refresher: Stacks

A stack is sometimes implemented using a **linked list**.

- "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, and we don't have templates.

Demo: Int Stack

`int_stack.c`

What modifications are necessary to make a generic stack?

Stack Structs

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


typedef struct stack {
    node *top;
    size_t nelems;
} stack;
```

How might we modify the Stack data representation itself to be generic?

Generic Stack Structs

The primary problem is that there's no single node type definition that can store data and a next pointer.

- An **`int`** stack requires nodes be **`sizeof(int) + sizeof(void *)`** in size
- A **`char *`** stack requires nodes be **`sizeof(char *) + sizeof(void *)`** in size
- A **`fraction`** stack? nodes are **`sizeof(fraction) + sizeof(void *)`** in size

```
typedef struct stack {
    void *top;
    size_t width;
    size_t nelems;
} stack;
```

Why is **`top`** of type **`void *`**? Because we know **`top`** stores the address of the first node in a list, but there's no data type that works for all nodes. Our nodes will instead be dynamically allocated figures whose size is **`width`** + **`sizeof(void *)`**. The first width bytes will store the data, and the rest will store the address of the next node.

stack_init

```
void stack_init(stack *s) {
    s->top = NULL;
    s->nelems = 0;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    void *top;
    size_t width;
    size_t nelems;
} stack;
```

Generic stack_init

```
void stack_init(stack *s, size_t width) {
    s->top = NULL;
    s->width = width;
    s->nelems = 0;
}
```

We should supply the element size at initialization time so the stack can track it and use it to know how many bytes need to be replicated on behalf of push and pop operations.

stack_push

```
void stack_push(stack *s, int data) {
    node *n = malloc(sizeof(node));
    n->data = data;
    n->next = s->top;
    s->top = n;
    s->nelems++;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    void *top;
    size_t width;
    size_t nelems;
} stack;
```

stack_push

```
void stack_push(stack *s, int data) {
    node *n = malloc(sizeof(node));
    n->data = data;
    n->next = s->top;
    s->top = n;
    s->nelems++;
}
```

Problem: we can no longer pass the data itself as a parameter because it could be of any type and any size.

stack_push

```
void stack_push(stack *s, int data) {
    node *n = malloc(sizeof(node));
    n->data = data;
    n->next = s->top;
    s->top = n;
    s->nelems++;
}
```

Solution: pass a pointer to the data as a parameter instead and **dynamically allocate just enough memory** to wedge in a copy of that data **aside** a pointer to the next node.

Generic stack_push

```
void stack_push(stack *s, const void *data) {
    void *n = malloc(s->width + sizeof(void *));
    memcpy(n, data, s->width);
    *(void **)((char *) n + s->width) = s->top;
    s->top = n;
    s->nelems++;
}
```

Solution: pass a pointer to the data as a parameter instead and **dynamically allocate exactly enough memory** to wedge in a copy of that data aside a pointer to the next node.

stack_pop

```
int stack_pop(stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    int val = n->data;

    // rewire
    s->top = n->next;
    free(n);
    s->nelems--;
    return val;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    void *top;
    size_t width;
    size_t nelems;
} stack;
```

stack_pop

```
int stack_pop(stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    int val = n->data;

    // rewire
    s->top = n->next;
    free(n);
    s->nelems--;
    return val;
}
```

Problem: we can no longer return the data itself, because it could be any size!

stack_pop

```
int stack_pop(stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    int val = n->data;

    // rewire
    s->top = n->next;
    free(n);
    s->nelems--;
    return val;
}
```

Solution: require the client supply the address of a figure where the data can be safely copied.

Generic stack_pop

```
void stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }

    void *n = s->top;
    memcpy(addr, n, s->width);
    s->top = *(void **)((char *)n + s->width);
    free(n);
    s->nelems--;
}
```

Solution: require the client supply the address of a figure where the data can be safely copied to.

Demo: Generic Stack

`generic_stack.c`

Recap

- **`void *`** is a variable type that represents a generic pointer to *something*.
- We cannot do pointer arithmetic on a **`void *`**. And you can't dereference of **`void *`** either.
- We can use **`memcpy`** or **`memmove`** to replicate data from one memory location to another.
- To do any type of generic pointer math using a **`void *`**, we very often cast it to a **`char *`**.
- **`void *`** and generics are lean and powerful, but they're dangerous. By going generic, we sedate the compile to ignore data types and whatever type checking would normally apply to those data types.

Plan For Today

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

Next time: More Generics, and Function Pointers