CS107, Lecture 8
C Generics – Void *

CS107 Topic 4: How can we use our knowledge of memory and data representation to write code that works with any data type?

Learning Goals

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

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect08 .
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect08 .
```

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


```
cp -r /afs/ir/class/cs107/lecture-code/lect08 .
```

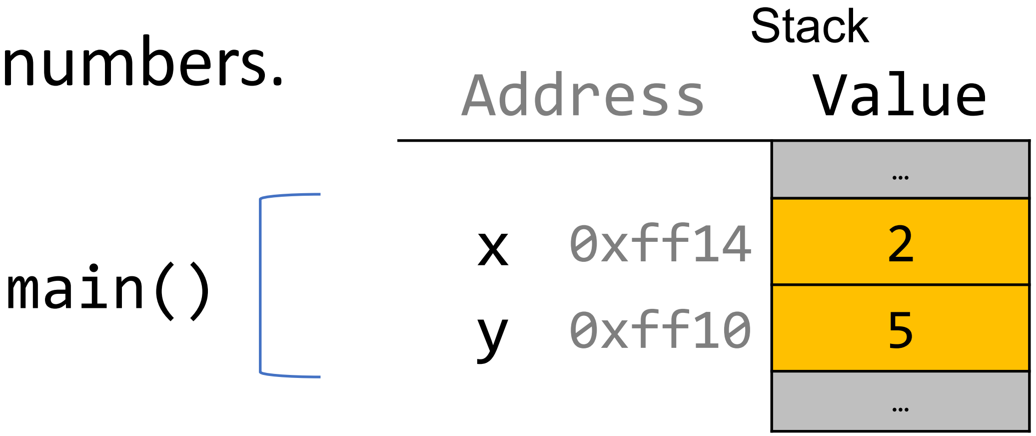
Swap

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.

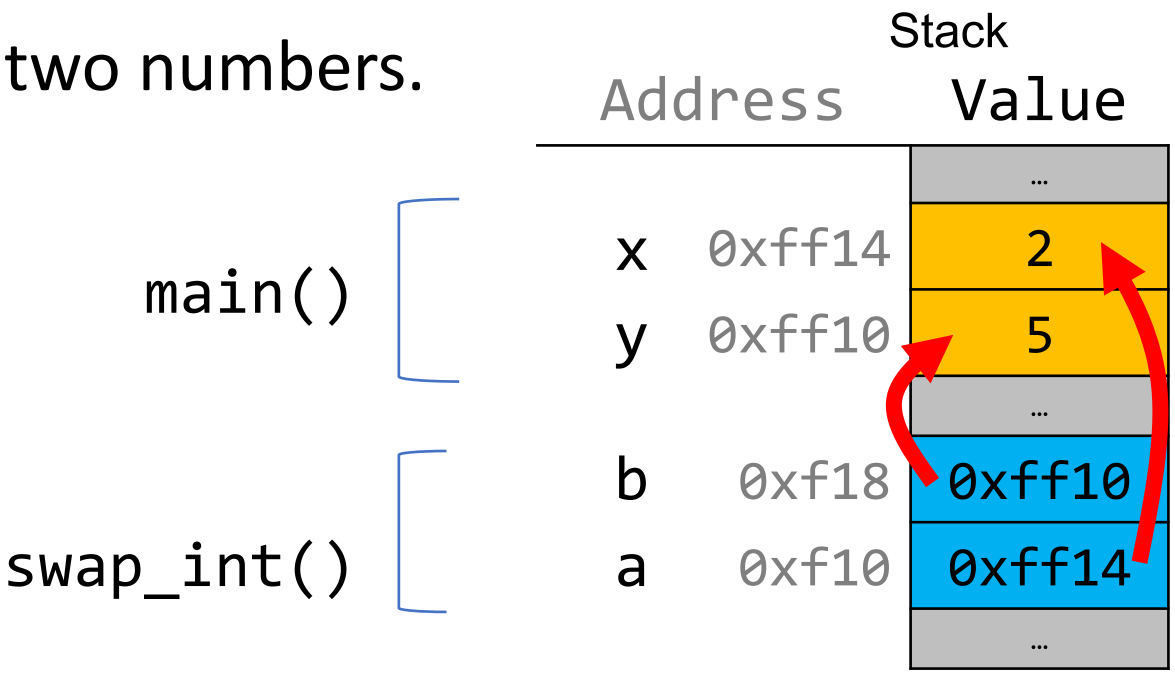



```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```

Swap

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

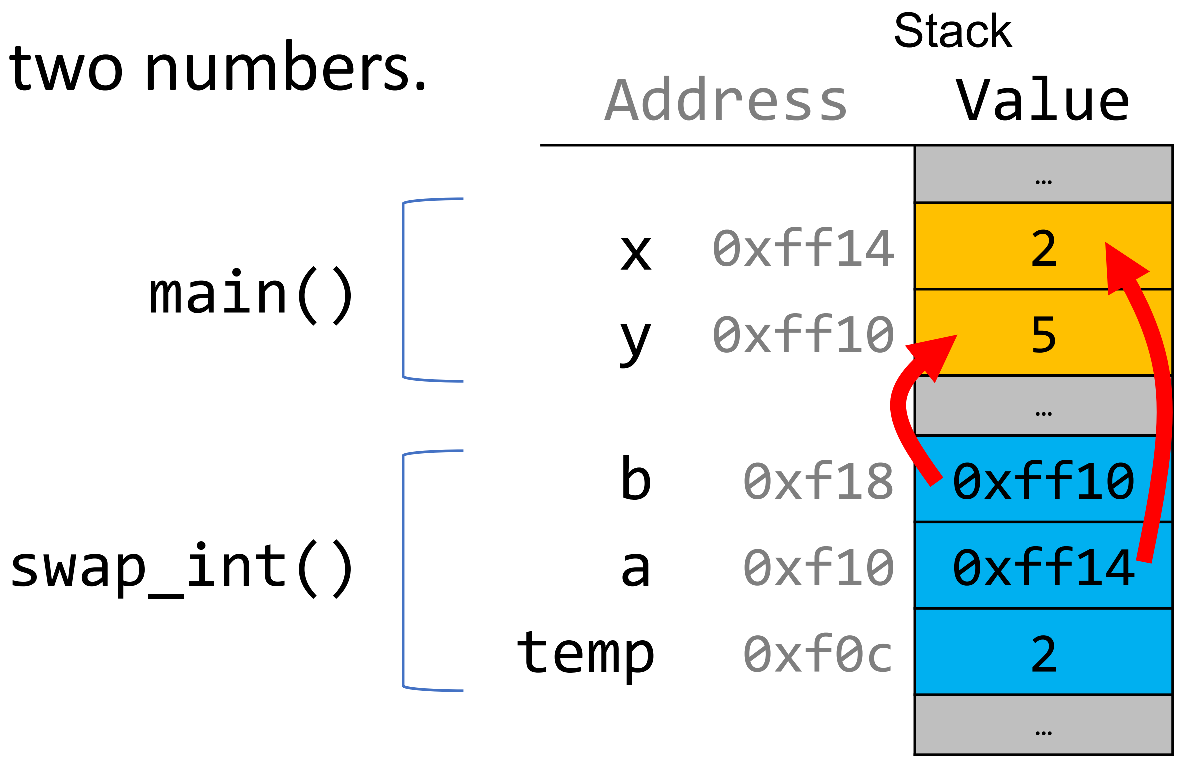
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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.

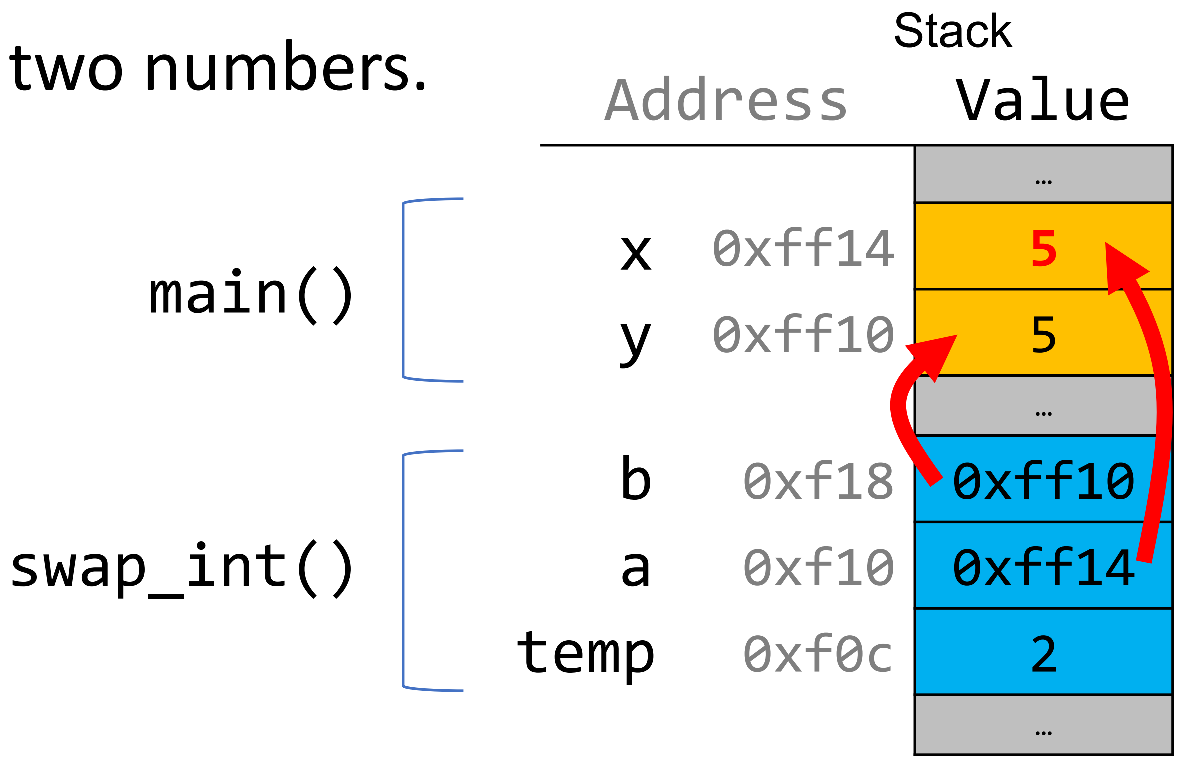
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```



Swap

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

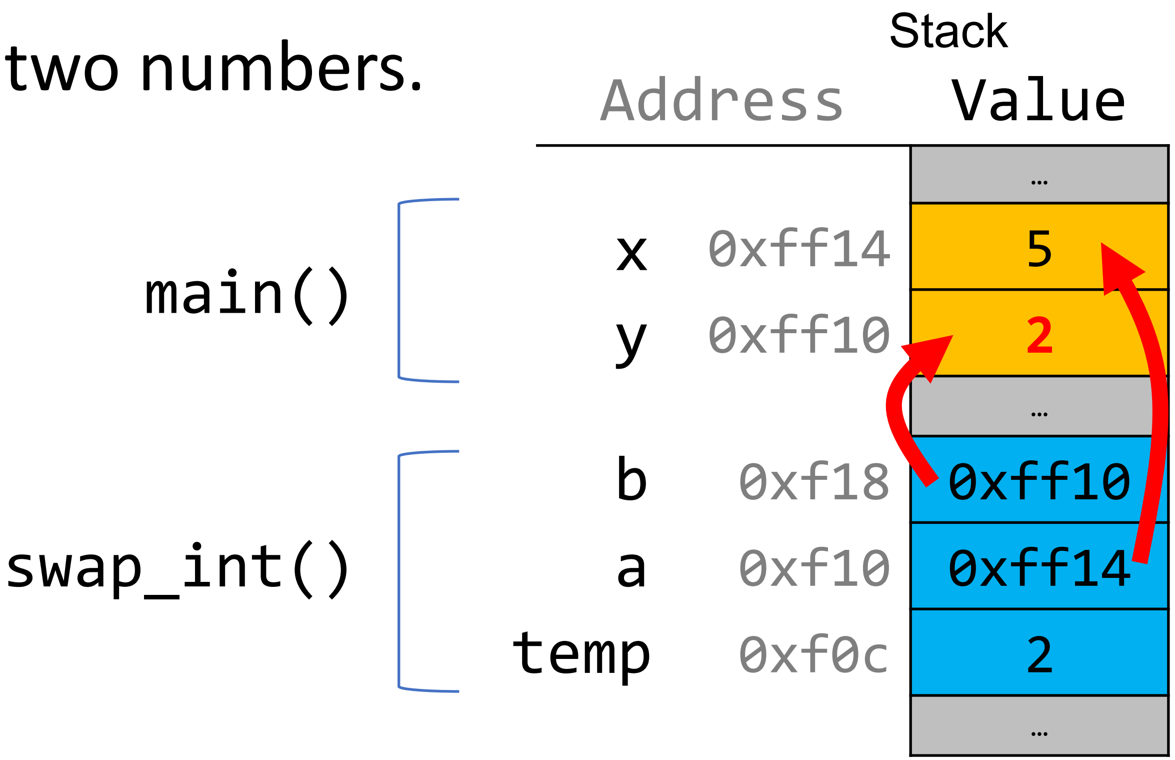
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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.

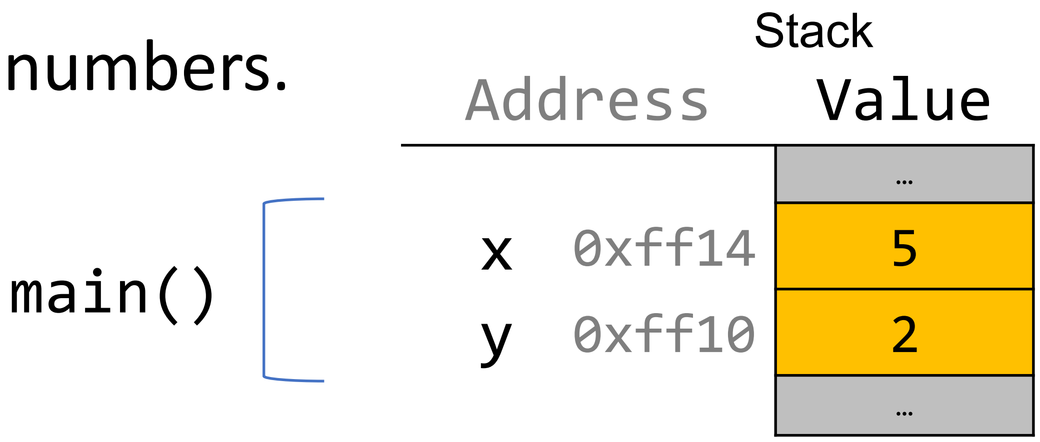
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```



Swap

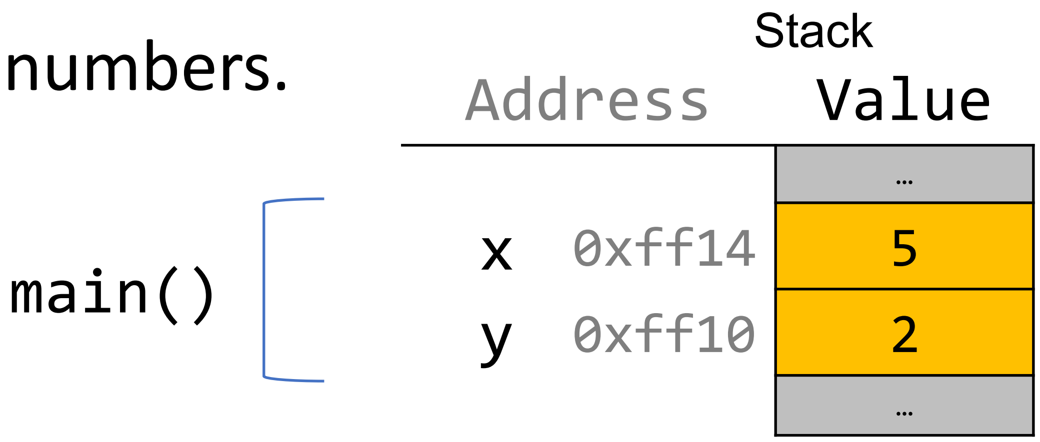
You're asked to write a function that swaps two numbers.

```
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.

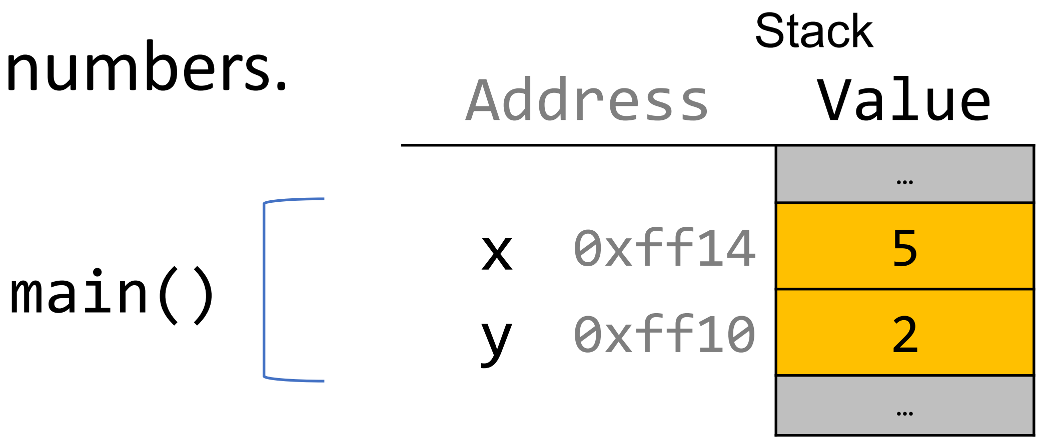



```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```

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;
}
```



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

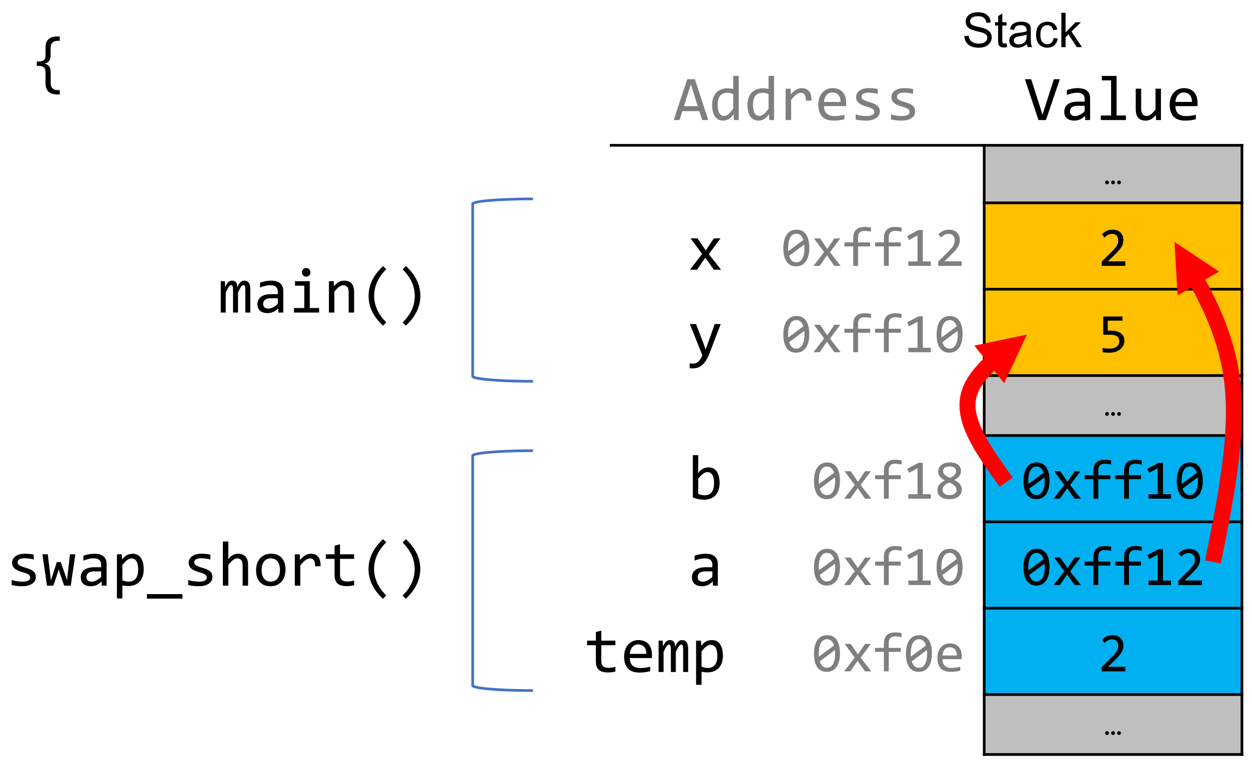
😑

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;
}
```

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;
}
```



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

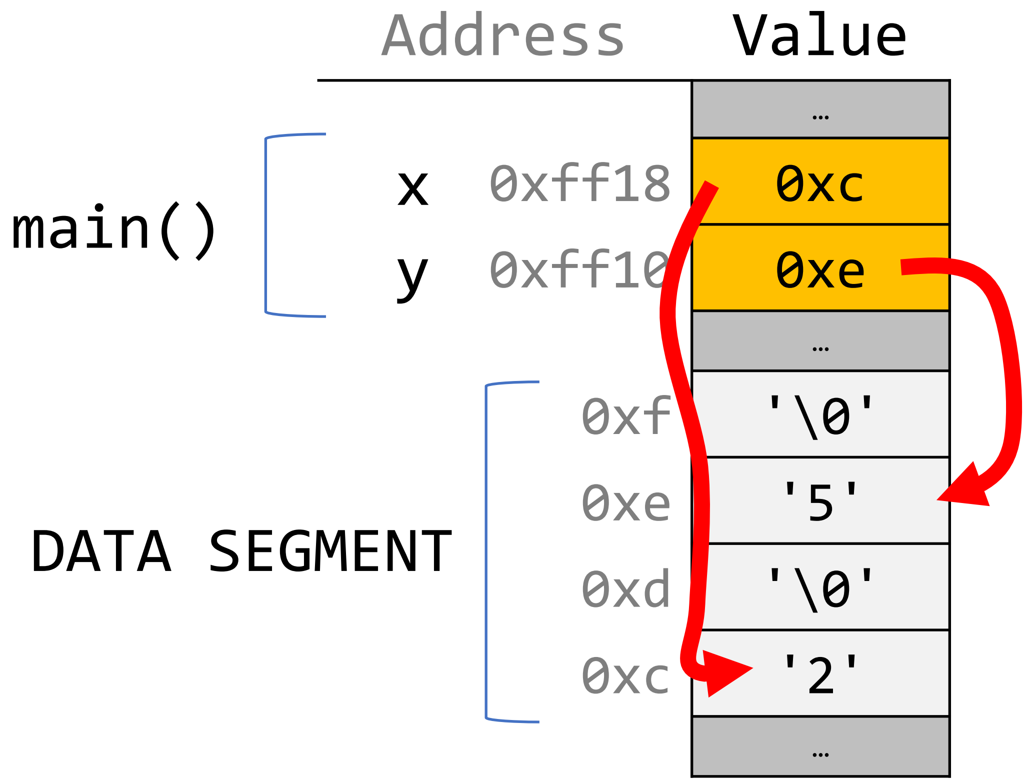
😤

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;
}
```

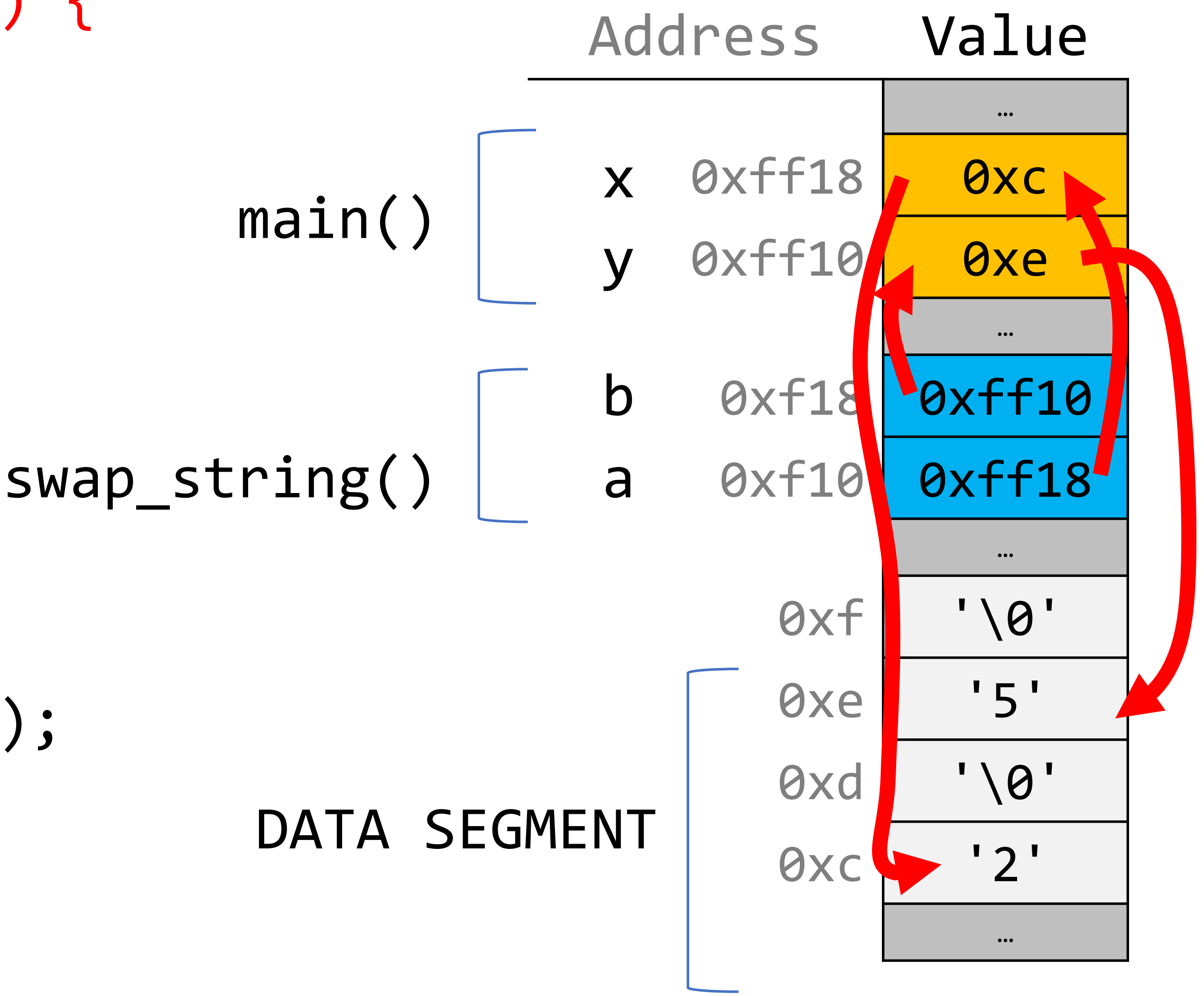
Swap

```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;
}
```



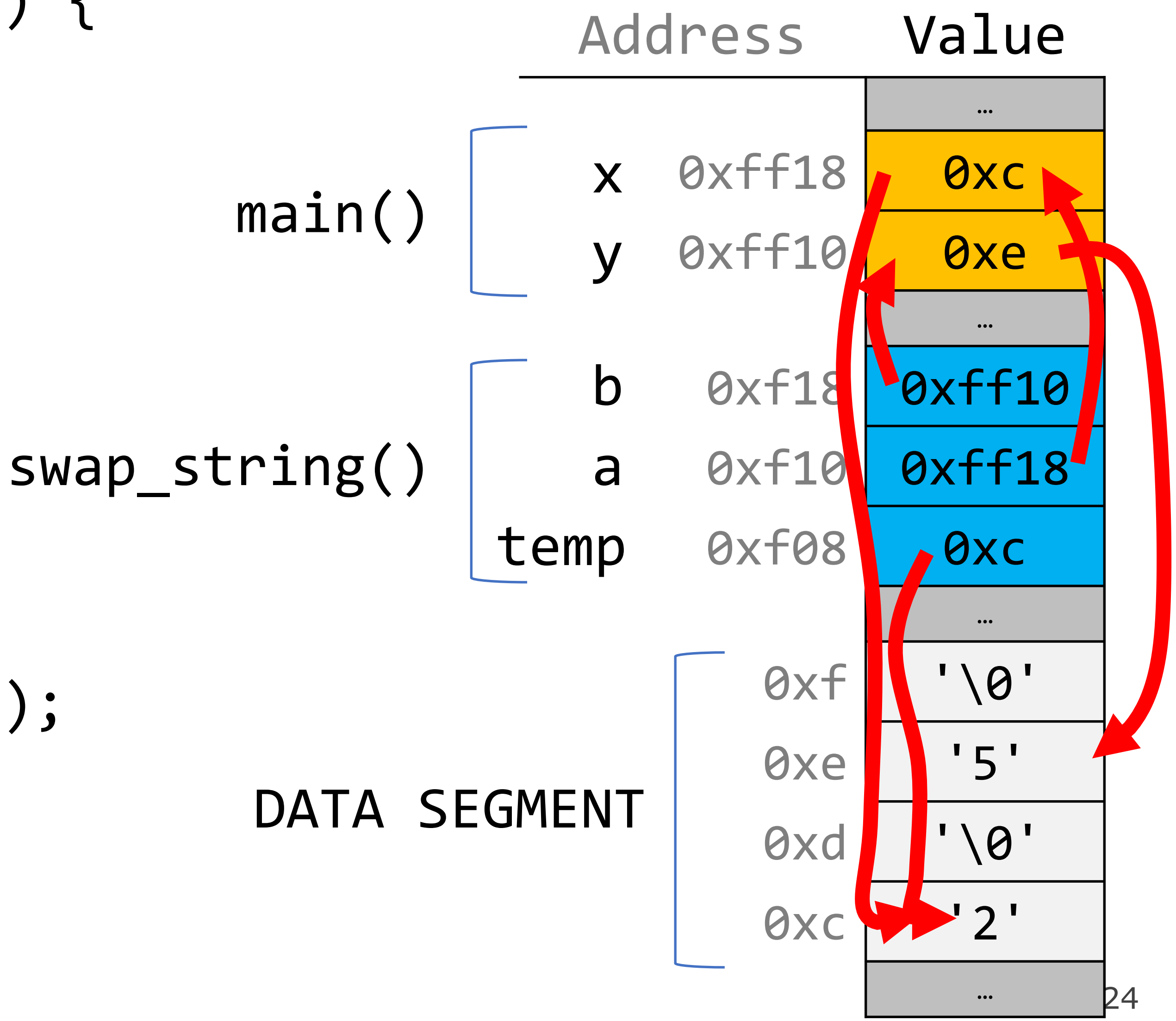
Swap

```
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;
}
```



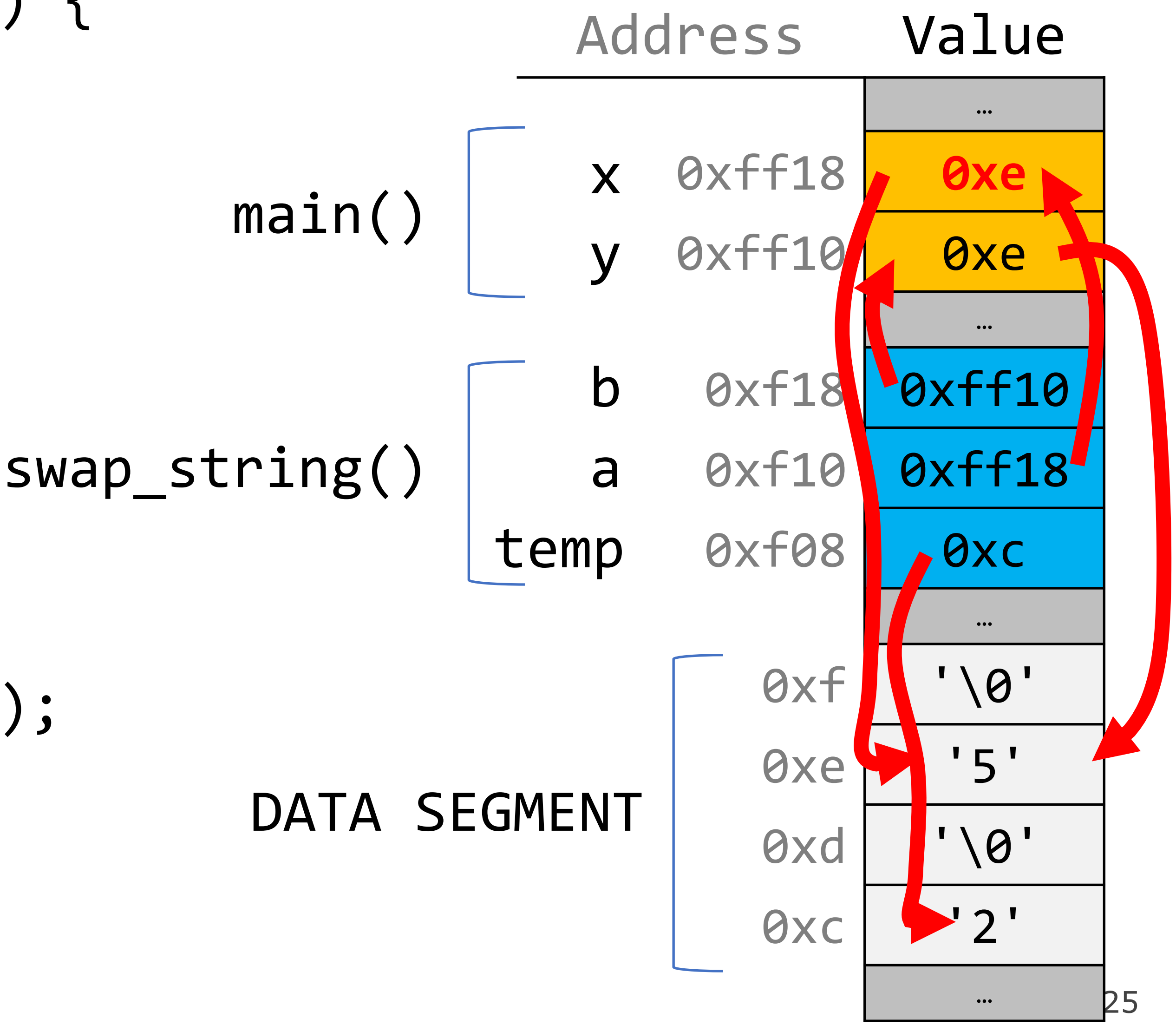
Swap

```
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;
}
```



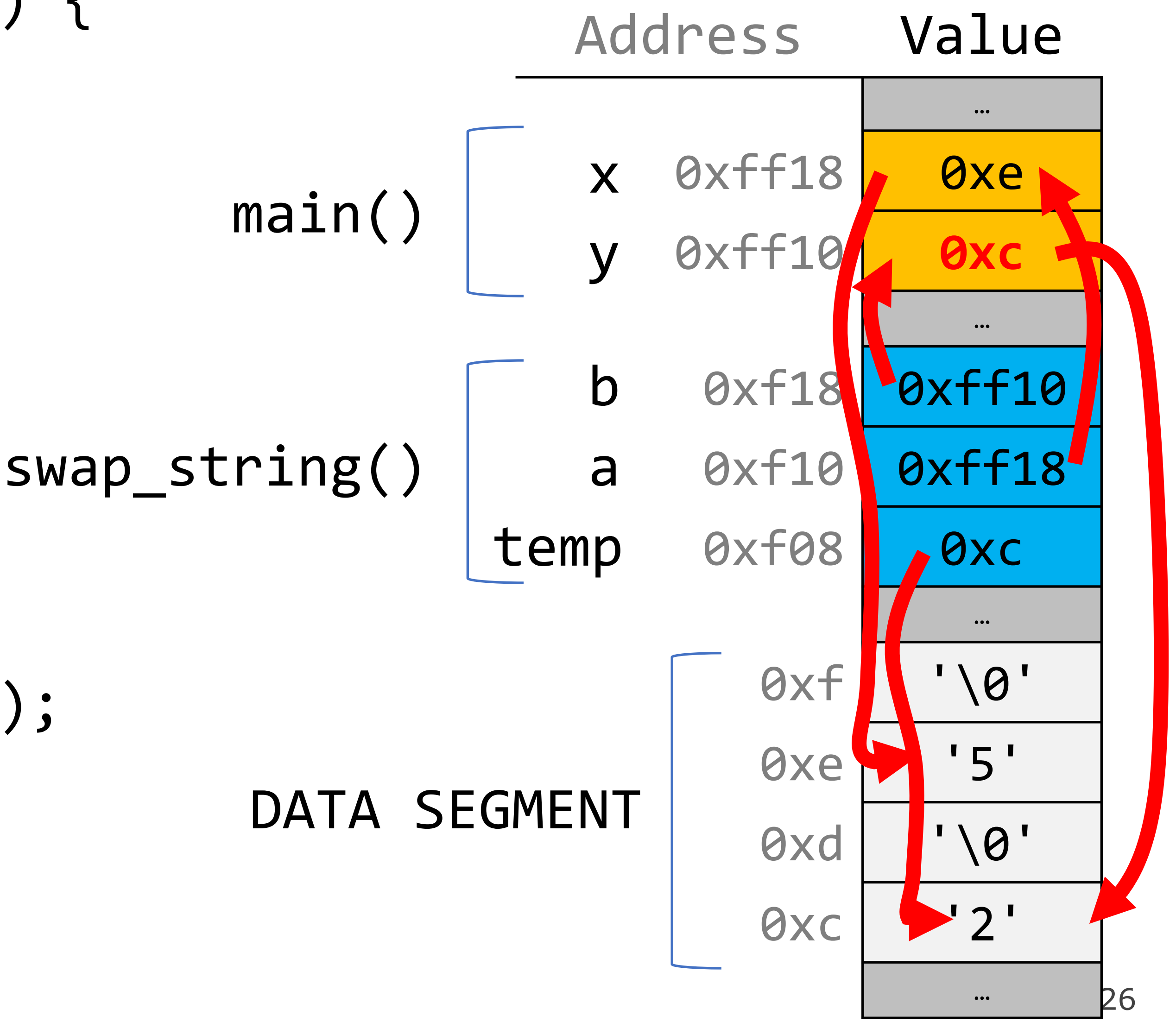
Swap

```
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;
}
```



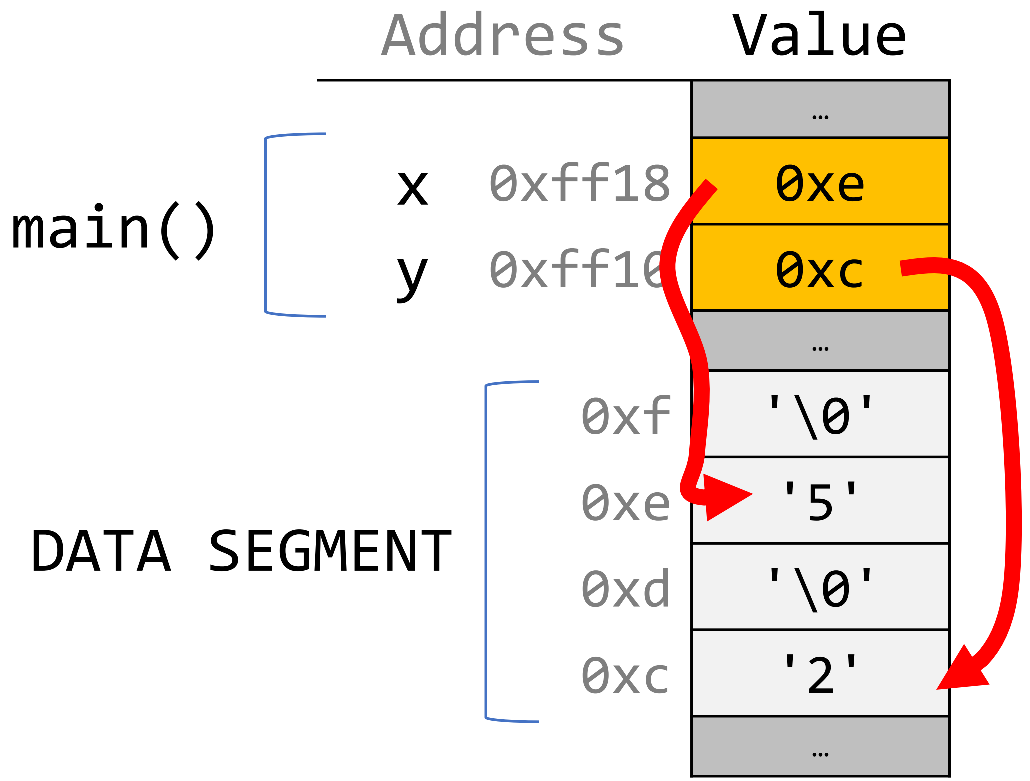
Swap

```
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;
}
```



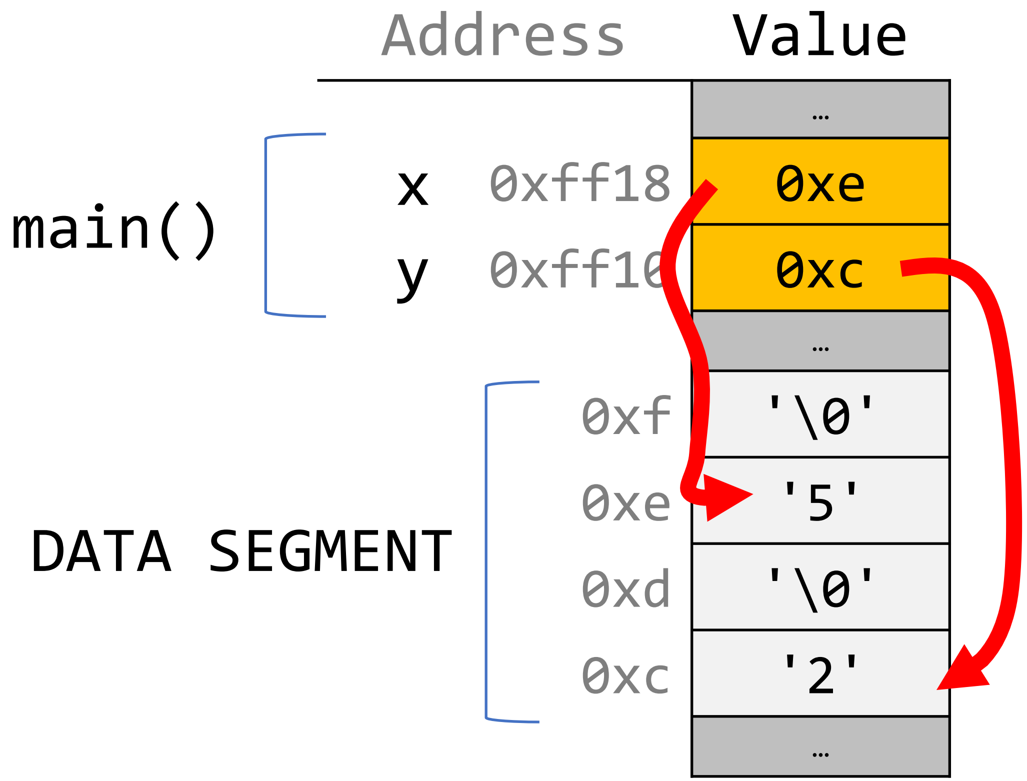
Swap

```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;
}
```



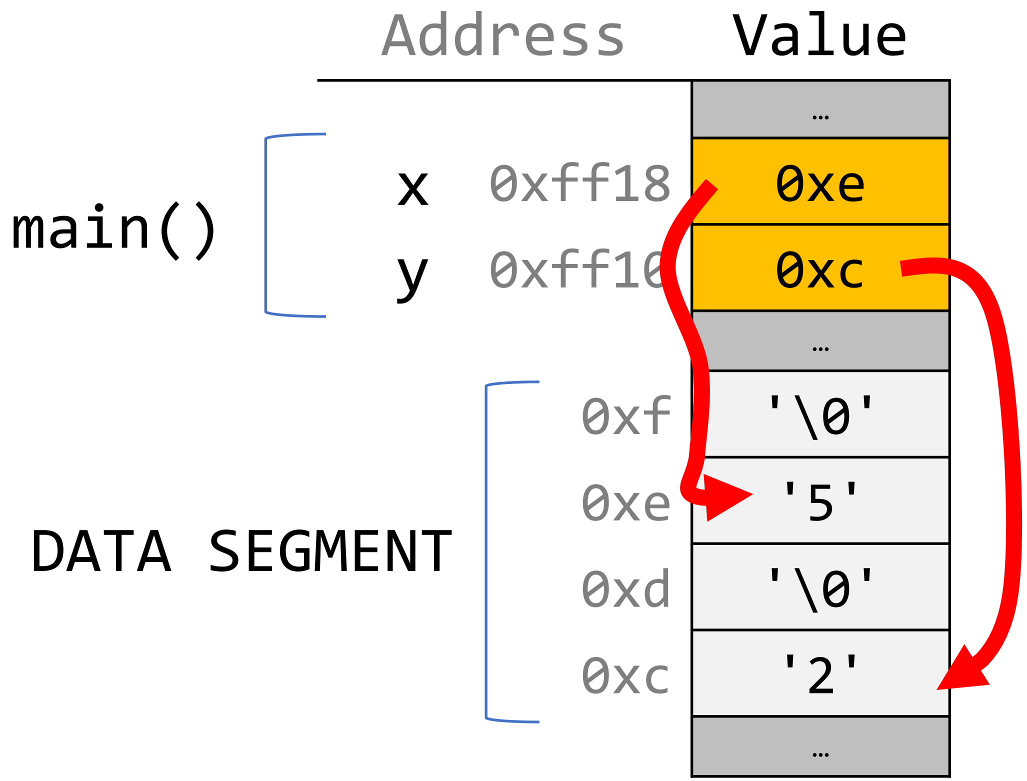
Swap

```
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.

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

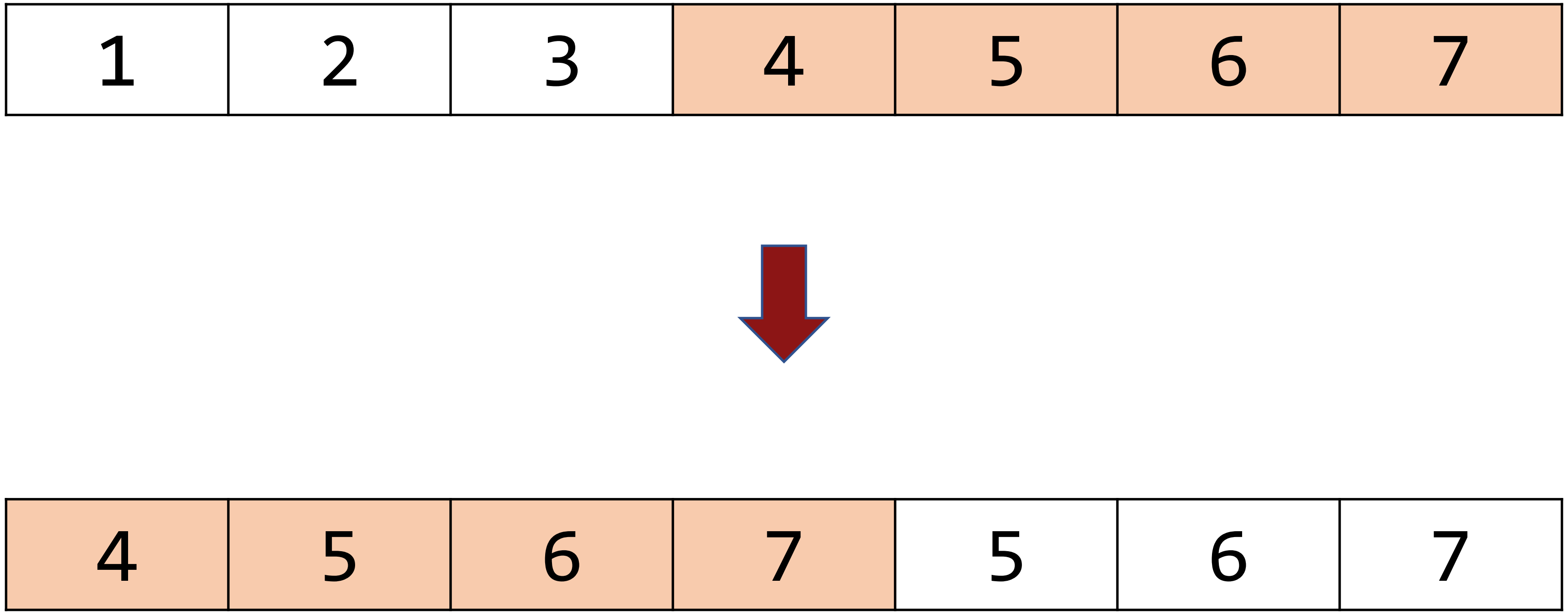
memmove is the same as memcpy, but supports overlapping regions of memory. (Unlike its name implies, it still “copies”).

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

It copies the next n bytes that src <u>points to</u> to the location contained in dest. (It also returns **dest**).

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);
}
```

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect08 .
```

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

swap.c

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

Lecture Plan


```
cp -r /afs/ir/class/cs107/lecture-code/lect08 .
```

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);
    // 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.

```
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 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 * 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…

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

Short?

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

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.

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 <u>anything</u>!

```
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]));
```

Demo: Void *s Gone Wrong

swap_ends.c

Lecture Plan

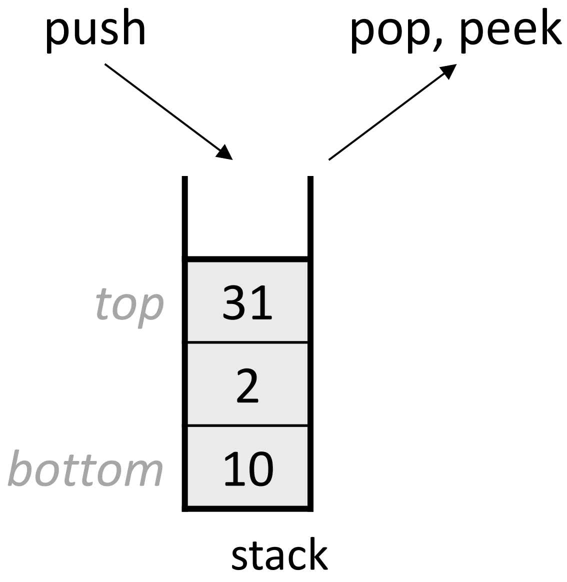

```
cp -r /afs/ir/class/cs107/lecture-code/lect08 .
```

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

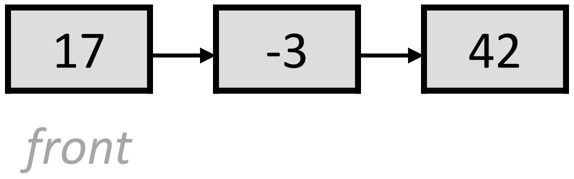


Refresher: Stacks

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

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

```
Stack&lt;int&gt; s;
s.push(42);
s.push(-3);
s.push(17);
```



Problem: C is not object-oriented! We can't call methods on variables.

Demo: Int Stack

`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

```c
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.

Demo: 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.

Recap

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

Next time: More Generics, and Function Pointers

Live Session Slides

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss

Lecture 8 takeaway: We can use **void ***, **memcpy** and **memmove** to manipulate data even if we don't know its type. We can cast **void ***s to perform pointer arithmetic. **void ***s have no type checking, so we must be vigilant!

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!

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

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);
}
```

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);

Pointer Arithmetic

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);
}
```

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

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;
}

...

stack *numStack = stack_create(sizeof(int));
```

Stack	Heap

```
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 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++;
}
...
int x = 2;
stack_push(numStack, &2);
```

Stack	Heap

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

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

Generic stack_pop

```
void stack_pop(stack *s, void *addr) {
    node *n = s->top;
    memcpy(addr, n->data,
        s->elem_size_bytes);
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
}
...
int num;
stack_pop(numStack, &num);
printf("%d\n", num);
```

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

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

Stack | Heap

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss

Lecture 8 takeaway: We can use **void ***, **memcpy** and **memmove** to manipulate data even if we don't know its type. We can cast **void** *s to perform pointer arithmetic. **void** *s have no type checking, so we must be vigilant!

Tips: C to English

- Translate C into English (function/variable declarations):

  https://cdecl.org/

- Pointer arithmetic: `(char *)` cast means byte address.
  What is the value of elt in the below (intentionally convoluted) code?

```
int arr[] = {1, 2, 3, 4};
void *ptr = arr;
int elt = *(int *)((char *) ptr + sizeof(int));
```

Code clarity: Consider breaking the last line into two lines! (1) pointer arithmetic, (2) int cast + dereference.

Exercise: Array Rotation

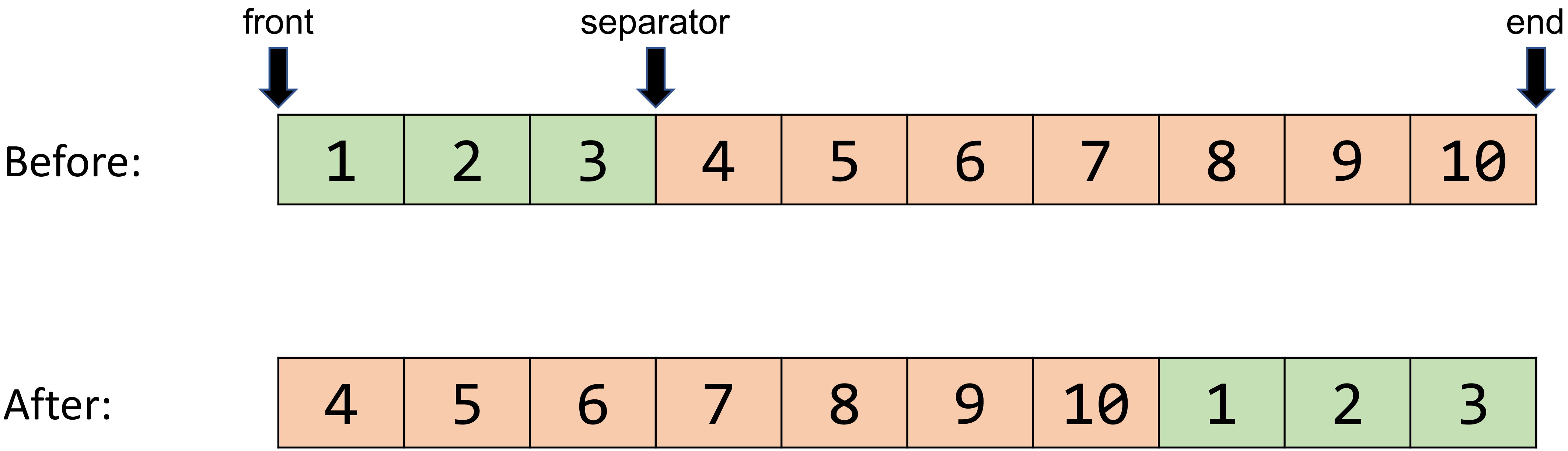
Exercise: You're asked to provide an implementation for a function called **`rotate`** with the following prototype:

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



Exercise: Array Rotation

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

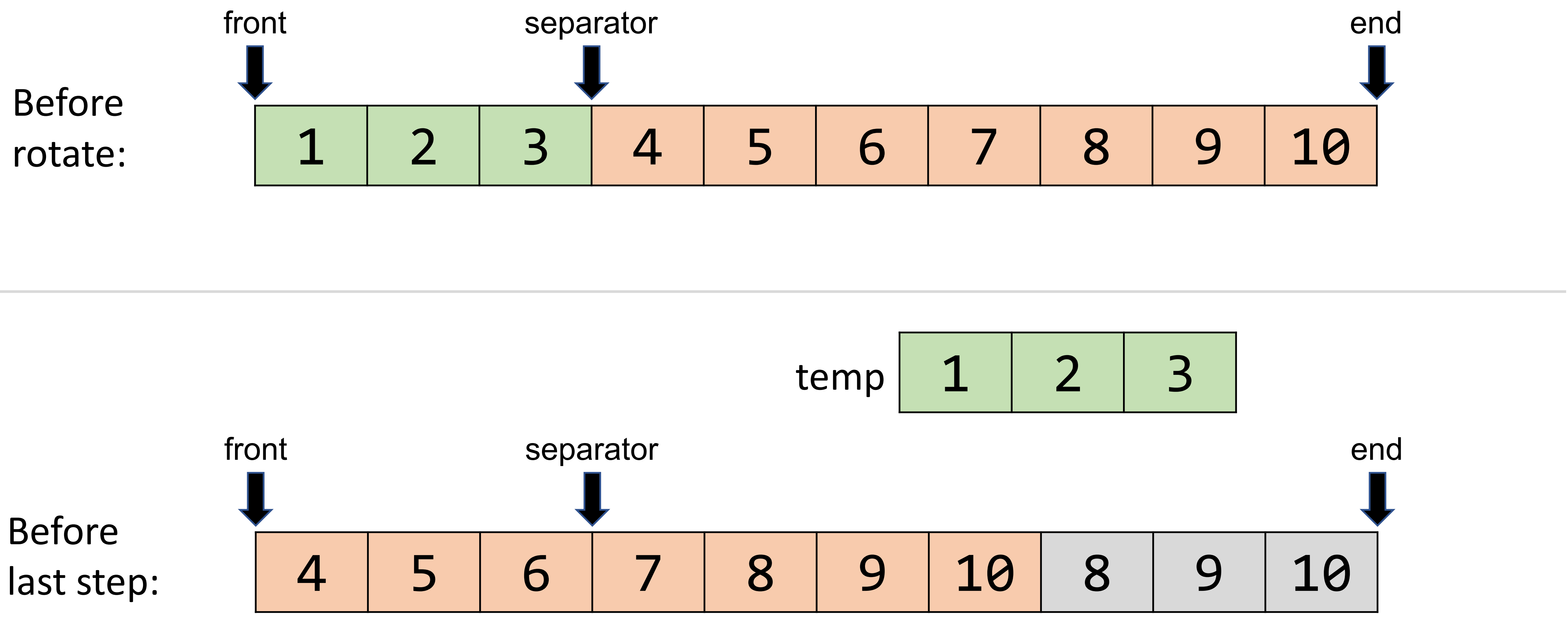
```
int main(int argc, char *argv[]) {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print_int_array(array, 10); // intuit implementation ☺
    rotate(array, array + 5, array + 10);
    print_int_array(array, 10);
    rotate(array, array + 1, array + 10);
    print_int_array(array, 10);
 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

Demo: Array Rotation

rotate.c

The inner workings of rotate



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);
}
```

stack_delete

- Lecture stack: No explicit delete stack; had to pop everything off in main
- How would we implement such a function?

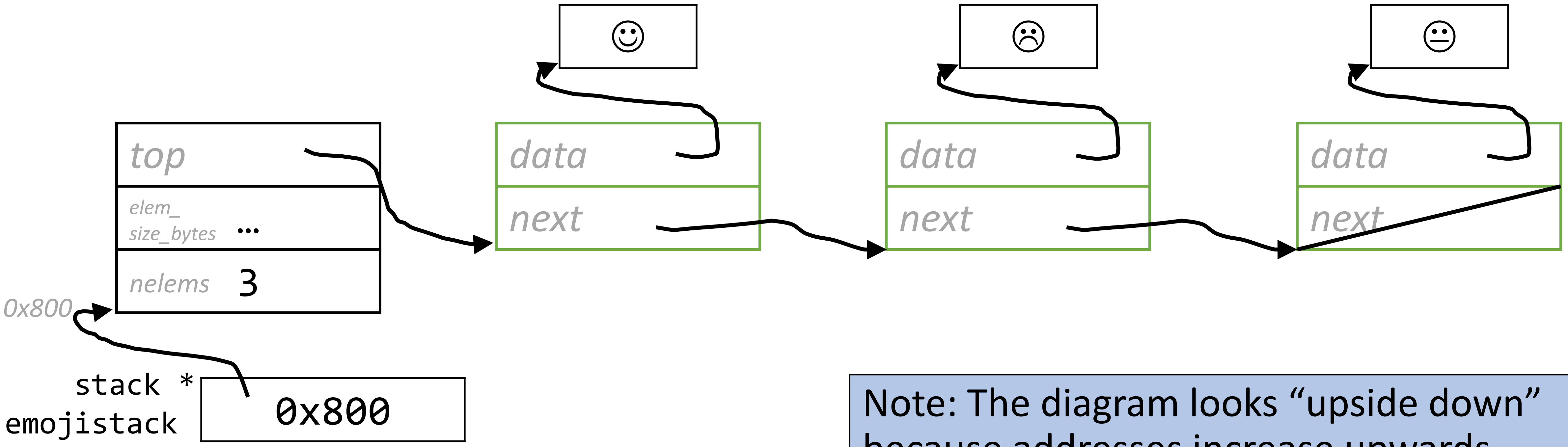
generic stack 2

These are extra slides that were not reviewed, to provide extra practice on another way to implement a generic stack. You can find code for this example in the lecture code folder (**live_stack.c**).

Lecture video stack

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

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

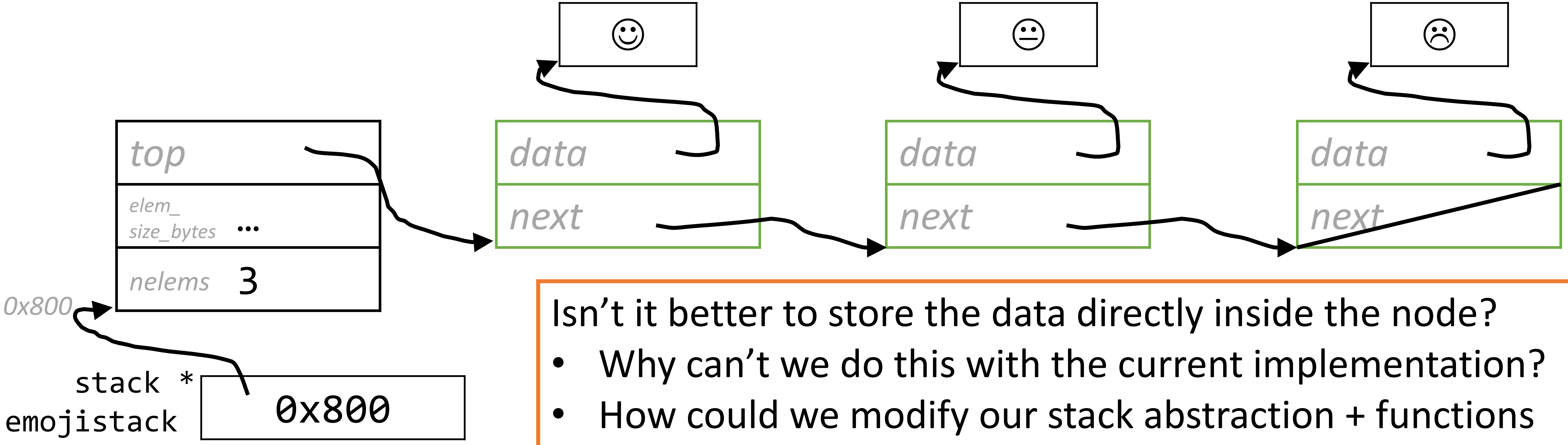


Note: The diagram looks "upside down" because addresses increase upwards.

Lecture video stack

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

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

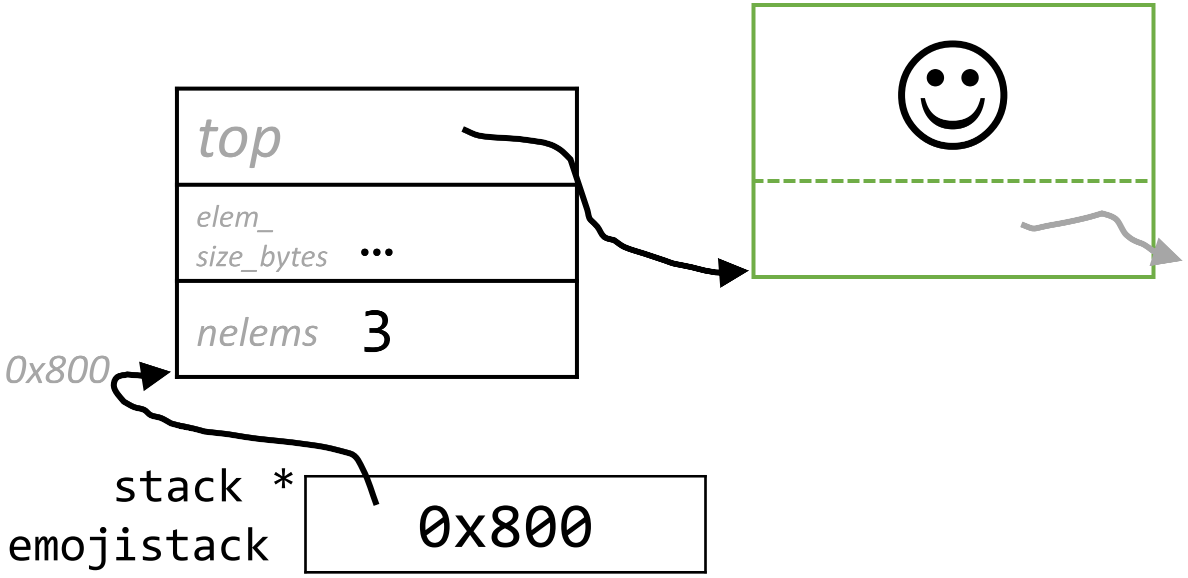


Isn't it better to store the data directly inside the node?

- Why can't we do this with the current implementation?
- How could we modify our stack abstraction + functions to do this?

A more efficient stack

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



If we *remove* the compile-time, 16B node struct:

- We create nodes that are elem_size_bytes + 8B and ***directly*** store the data into our node.
- A "node" just becomes contiguous bytes of memory storing (1) address of next node, and (2) data
- ⚠️ **Tricky!** We will be working with `sizeof(void *)` and `(void **)`!!

🌟 live_stack goals 🌟

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

- Rewrite our `generic_stack.c` code without the node struct
- Rewrite (as needed):
  - `stack_create`
  - `stack_push`
  - `stack_pop`
- (Don't touch main—a user of our stack should not know the difference)

stack_create

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

```
stack *stack_create(size_t elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

✅ No nodes touched, nothing to change

Old stack_push

```c
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++;
}
```

What do we have to change from the old function? Check all functionality:

1. Allocate a node
2. Copy in data
3. Set new node’s next to be top of stack
4. Set top of stack to be new node
5. Increment element count

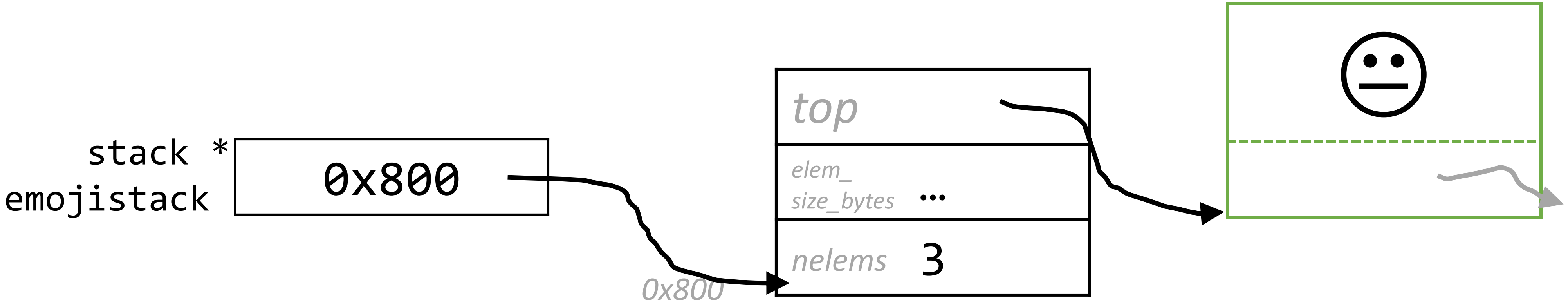
(we’ll go over each step next)

1. Allocate a node

In `stack_push`, we had: `node *new_node = malloc(sizeof(node));`

- We no longer have a typedef struct node!
  Our node is now just **contiguous bytes on the heap**.
- How do we **rewrite** this line to handle
  our new node representation?

1. Allocate a node

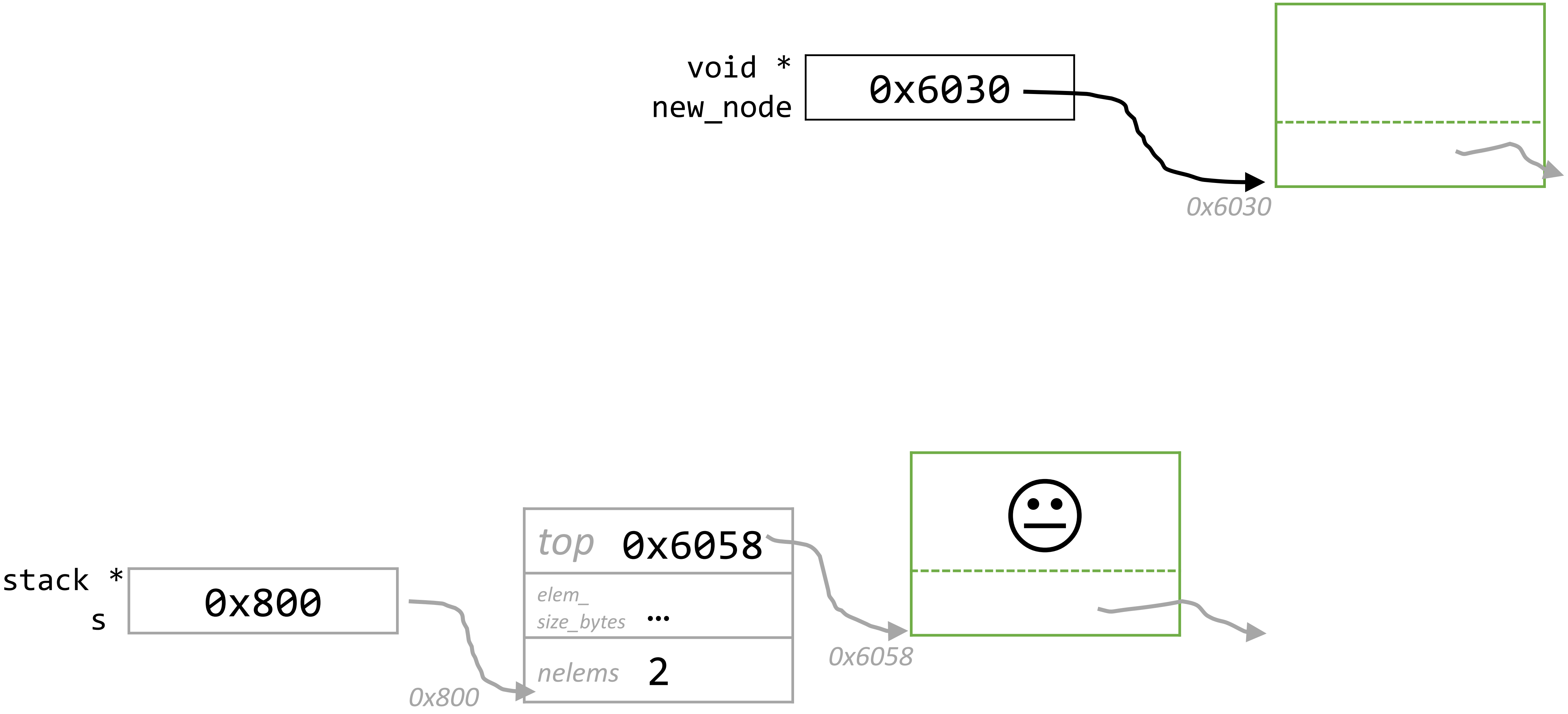


In stack_push, we had: `node *new_node = malloc(sizeof(node));`

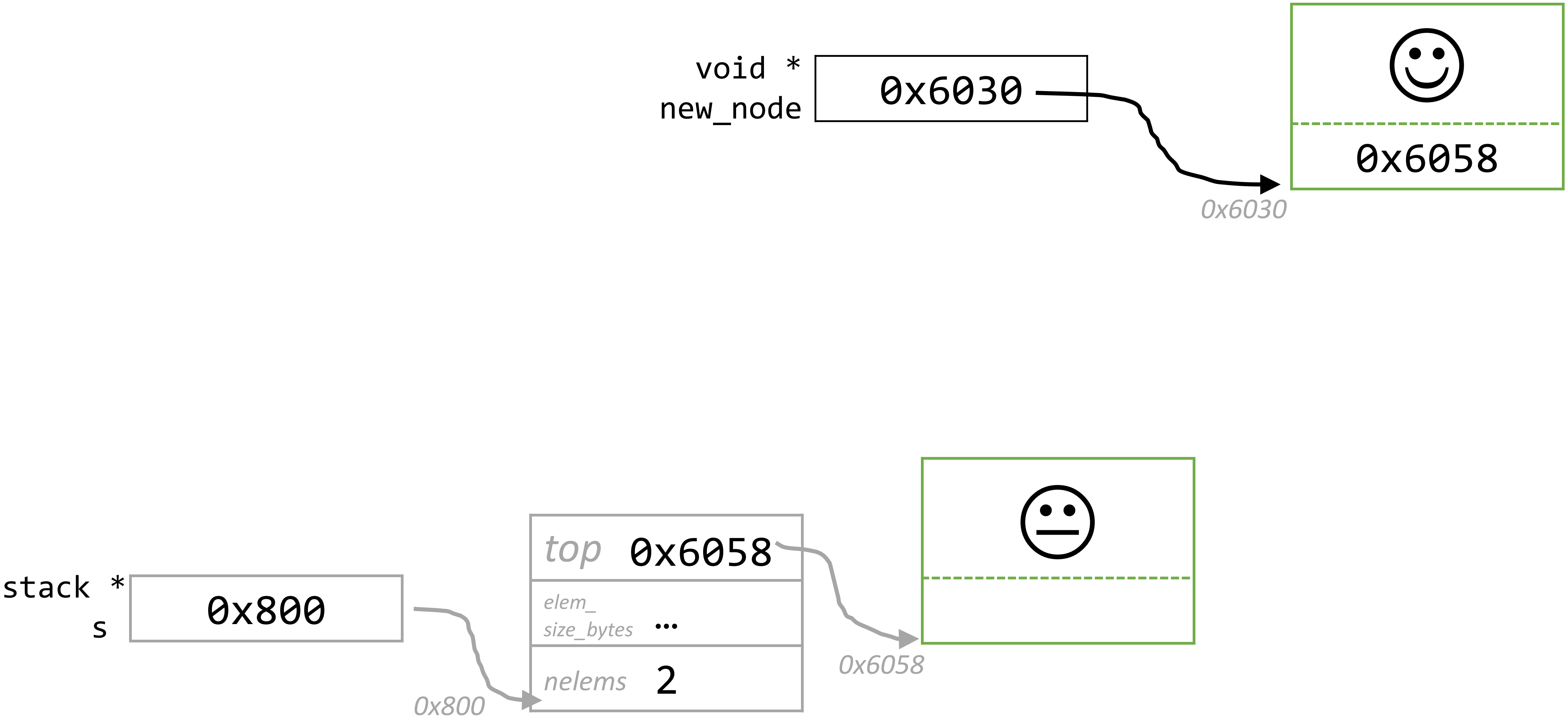
- We no longer have a typedef struct node!
  Our node is now just **contiguous bytes on the heap**.
- How do we **rewrite** this line to handle
  our new node representation?

```
void *new_node = malloc(sizeof(void *) + s->elem_size_bytes);
```

2, 3. Copy in data, set node next



4, 5. Update stack top, nelems



New stack_push

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

Check all functionality:

1. Allocate a node
2. Copy in data
3. Set new node’s next to be top of stack
4. Set top of stack to be new node
5. Increment element count

🌟 live_stack takeaways 🌟

- `sizeof(void *)` is the size of a pointer,
  which is always 8B (64-bit addresses)

- The dereference operation * (void **) ptr works!
  - `void * ptr = …;` Declaration: ptr stores an address, no idea what is at the address ptr
  - `(void **) ptr` Cast: at the address ptr, **there is an address**
  - `*(void **) ptr` Dereference: **get the address** stored at the address ptr

Old stack_pop

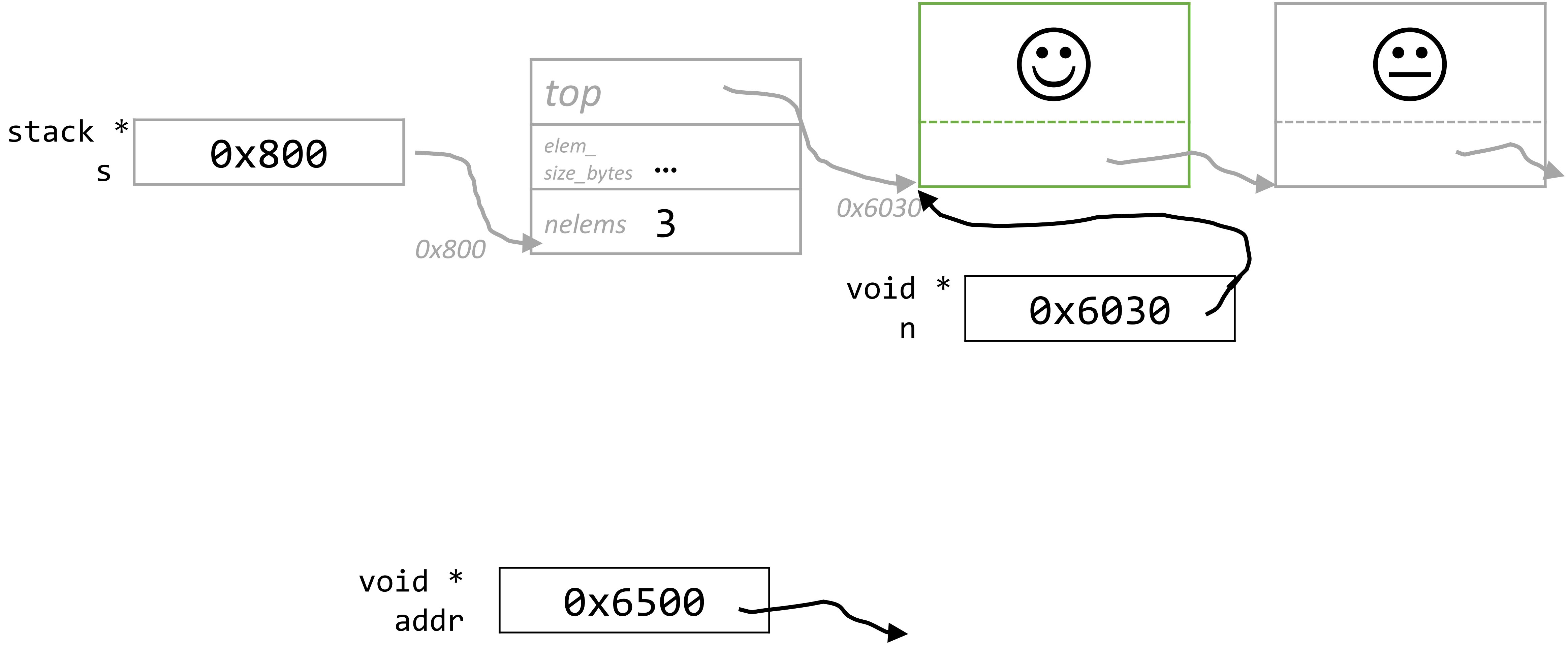
```c
void stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
}
```

What do we have to change from the old function? Check all functionality:

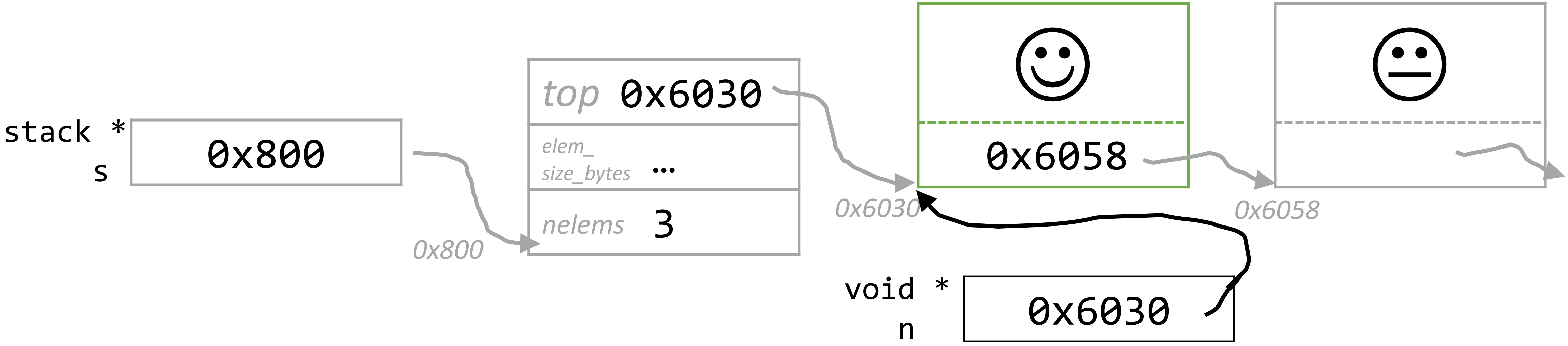
1. Copy top node’s data to addr buf
2. Set top of stack to top node’s next
3. Free old top node
4. Decrement element count

(we’ll go over each step next)

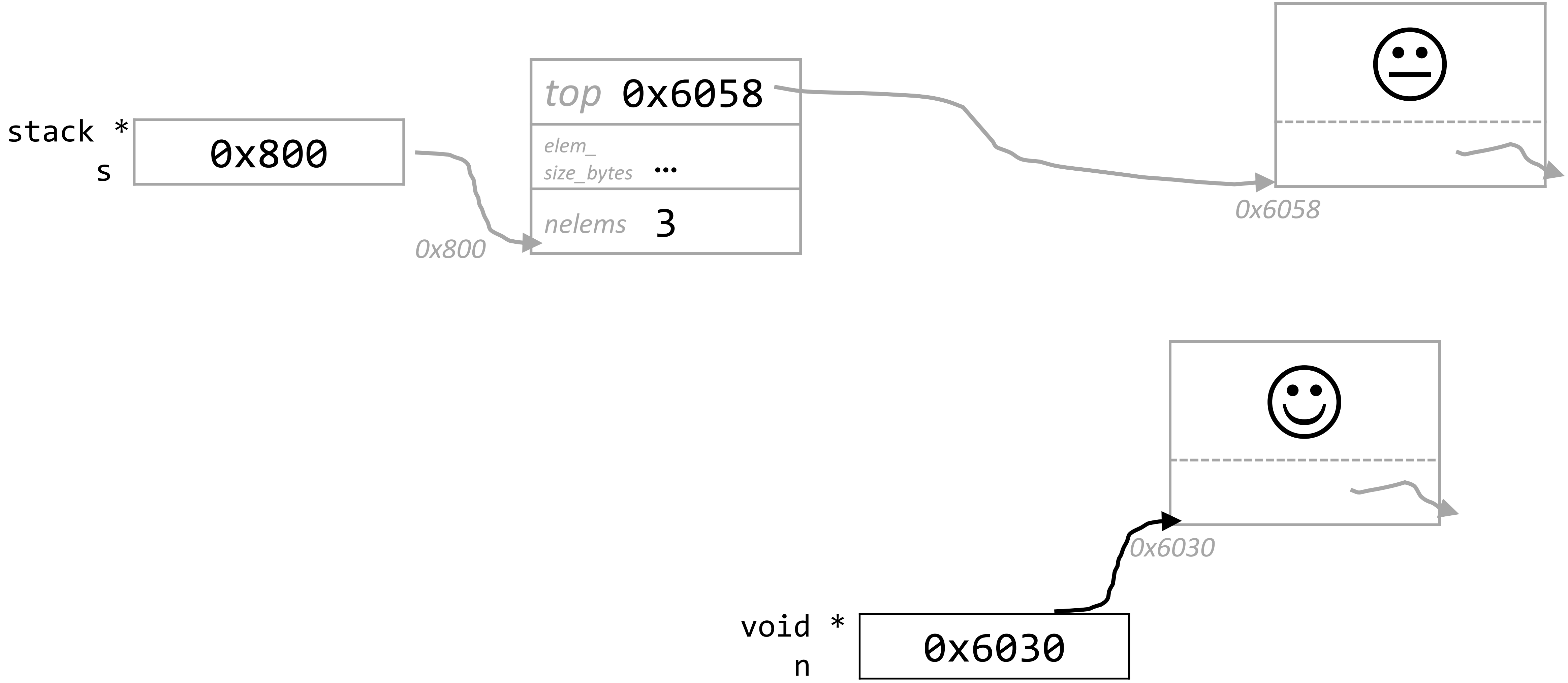
1. Copy node data to output buffer



2. Update stack top



3, 4. Free old top, update nelems



New stack_pop

```c
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, (char *) n + sizeof(void *), s->elem_size_bytes);
    s->top = *(void **) n;
    free(n);
    s->nelems--;
}
```

Check all functionality:

1. Copy top node's data to addr buf
2. Set top of stack to top node's next
3. Free old top node
4. Decrement element count

🌟 live_stack takeaways 🌟

- `sizeof(void *)` is the size of a pointer, which is always 8B (64-bit addresses)

- The dereference operation * (void **) ptr works!
  - `void * ptr = …;` Declaration: ptr stores an address, no idea what is at the address ptr
  - `(void **) ptr` Cast: at the address ptr, **there is an address**
  - `*(void **) ptr` Dereference: **get the address** stored at the address ptr