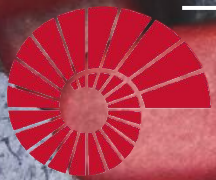


COMP201

Computer Systems & Programming

Lecture #10 – C Generics – void *



KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Spring 2026

Recap: Heap allocation interface:

```
void *malloc(size_t size);  
void *calloc(size_t nmemb, size_t size);  
void *realloc(void *ptr, size_t size);  
char *strdup(char *s);  
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- NULL on failure, so check with assert
- Memory is contiguous; it is not recycled unless you call free
- realloc preserves existing data
- calloc zero-initializes bytes, malloc and realloc do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after free, or if free is called twice on a location.
- If you realloc/free non-heap address

Recap: The Stack vs The Heap

Stack (“local variables”)

- **Fast**
Fast to allocate/deallocate; okay to oversize
- **Convenient.**
Automatic allocation/ deallocation;
declare/initialize in one step
- **Reasonable type safety**
Thanks to the compiler
- ⚠ **Not especially plentiful**
Total stack size fixed, default 8MB
- ⚠ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated
by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful.**
Can provide more memory on demand!
- **Very flexible.**
Runtime decisions about how much/when to
allocate, can resize easily with realloc
- **Scope under programmer control**
Can precisely determine lifetime
- ⚠ **Lots of opportunity for error**
Low type safety, forget to allocate/free
before done, allocate wrong size, etc.,
Memory leaks (much less critical)

COMP201 Topic 5: How can we use our knowledge of memory and data representation to write code that works with any data type?

Learning Goals

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

Plan for Today

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

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

Lecture plan

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

Generics

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

Lecture Plan

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

Swap

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

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

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

Swap

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

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

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

main()



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

Swap

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

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

main()
swap_int()

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

Swap

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

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

main()

swap_int()

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

The diagram illustrates the stack layout during the execution of the swap function. The stack grows downwards in memory. The variables x and y are located at addresses 0xff14 and 0xff10, respectively, and contain the values 2 and 5. The variables b and a are located at addresses 0xf18 and 0xf10, respectively, and contain the addresses 0xff10 and 0xff14. The variable temp is located at address 0xf0c and contains the value 2. Red arrows indicate the flow of data: from x to temp, from y to b, and from temp to a.

Swap

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

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

main()

swap_int()

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

The diagram illustrates the stack layout during the execution of the swap function. The stack grows downwards. The variables x and y in main() are located at addresses 0xff14 and 0xff10, respectively, and both contain the value 5. The variables b and a in swap_int() are located at addresses 0xf18 and 0xf10, respectively, and contain the addresses 0xff10 and 0xff14. The variable temp in swap_int() is located at address 0xf0c and contains the value 2. Red arrows show the flow of data: from x to b, from y to a, and from temp to x and y.

Swap

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

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

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

main()
swap_int()

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

Swap

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

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

main()



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

Swap

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

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

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

main()



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

Swap

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

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

main()



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

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



Swap

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

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

Swap

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

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

main()

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

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



Swap

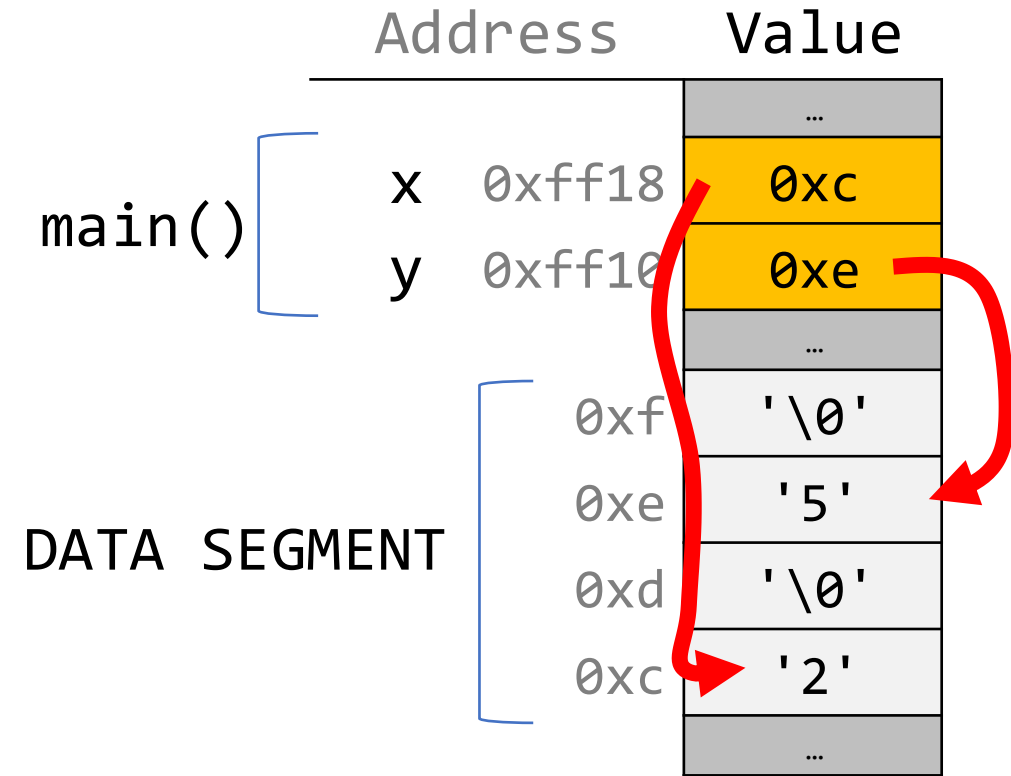
```
void swap_string(char **a, char **b) {  
    char *temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

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

Swap

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

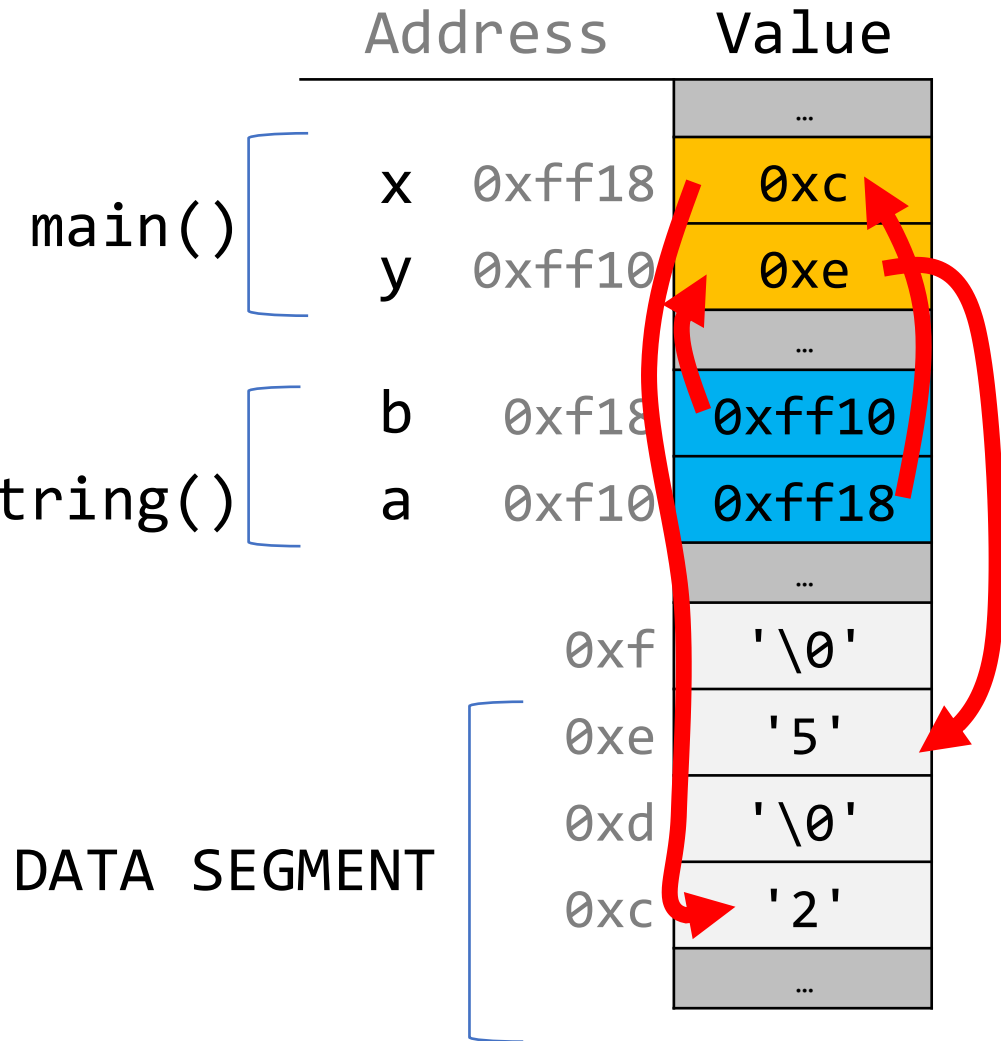
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

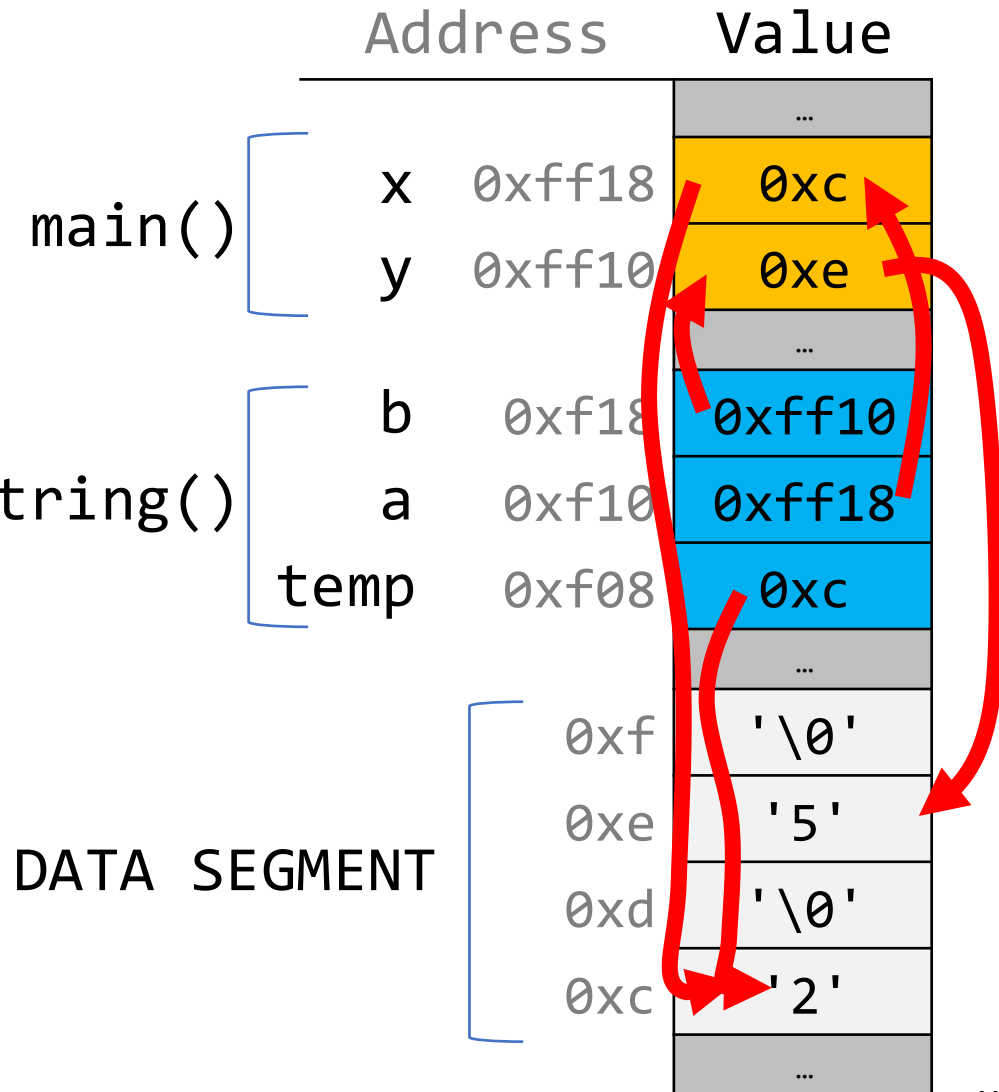
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

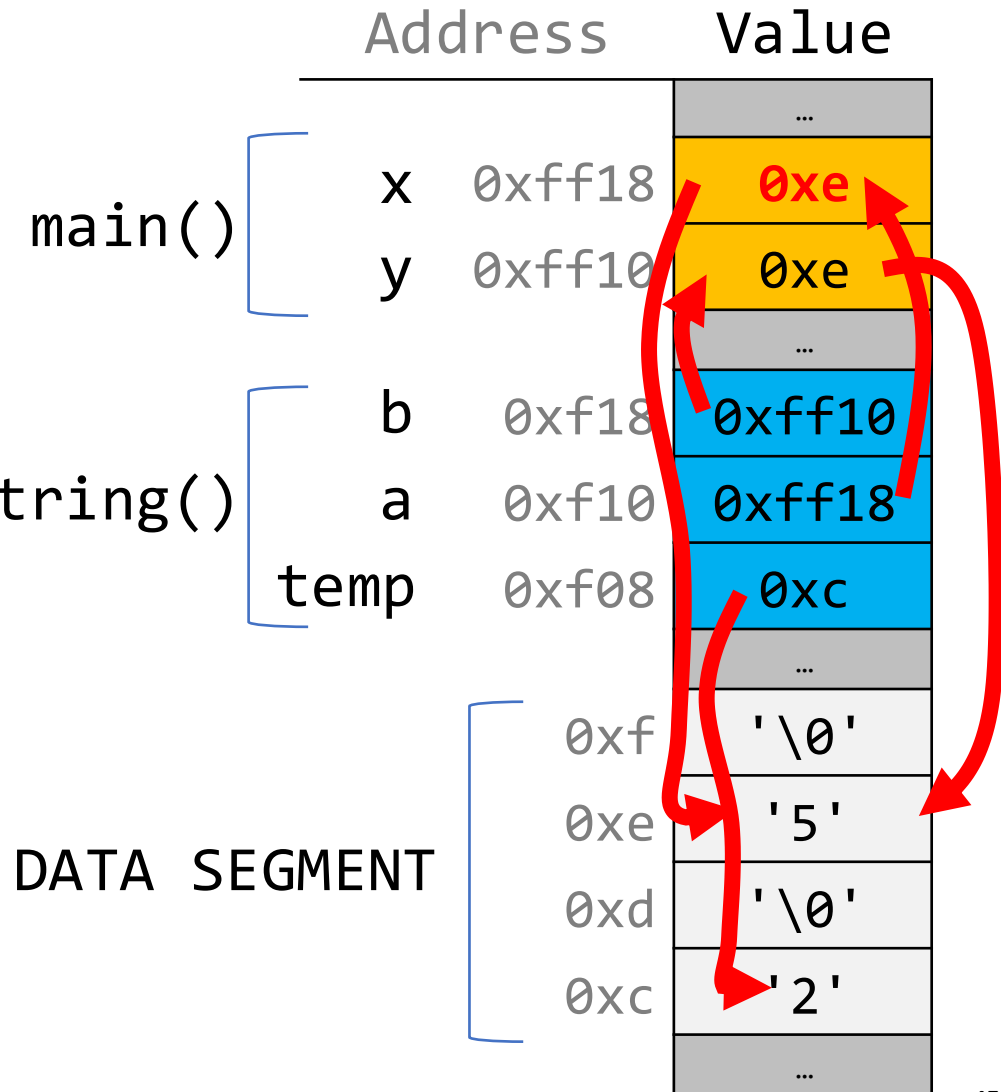
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

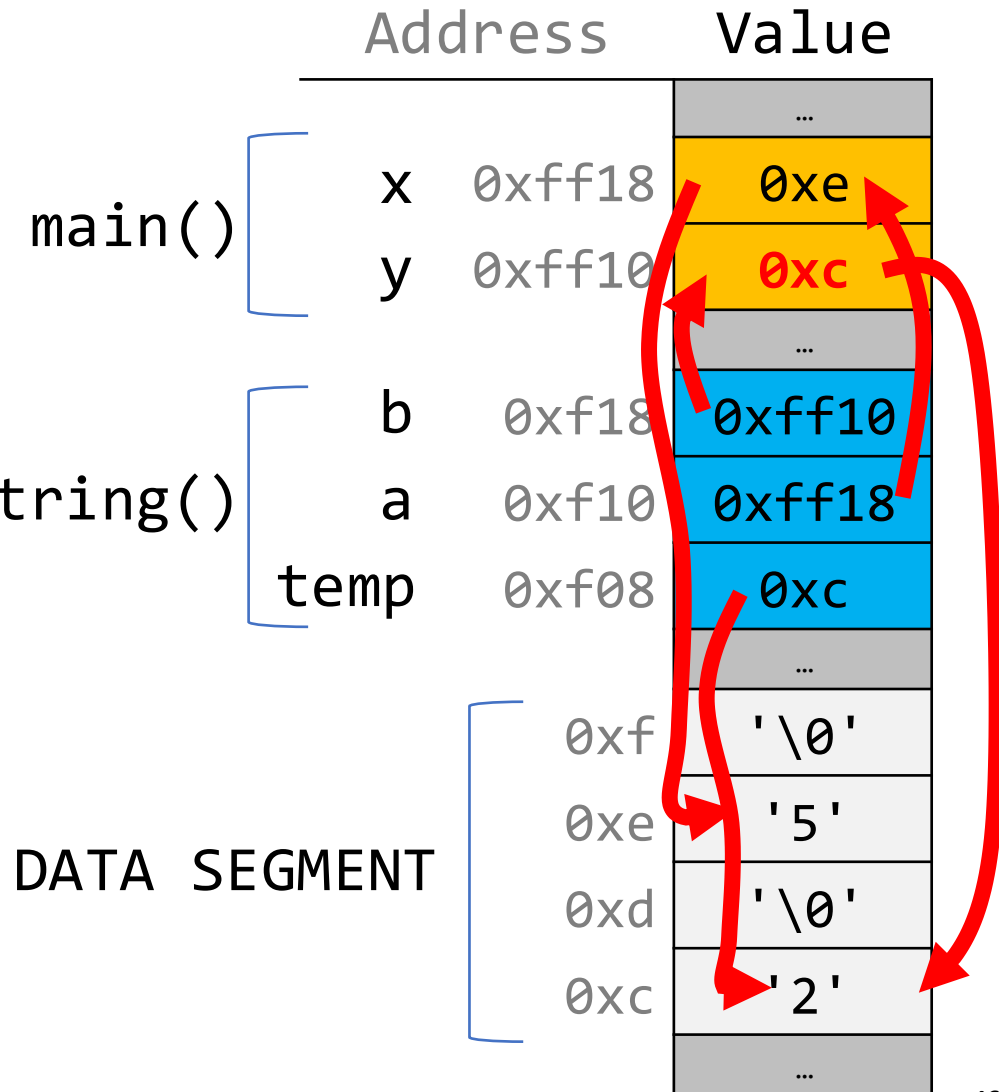
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

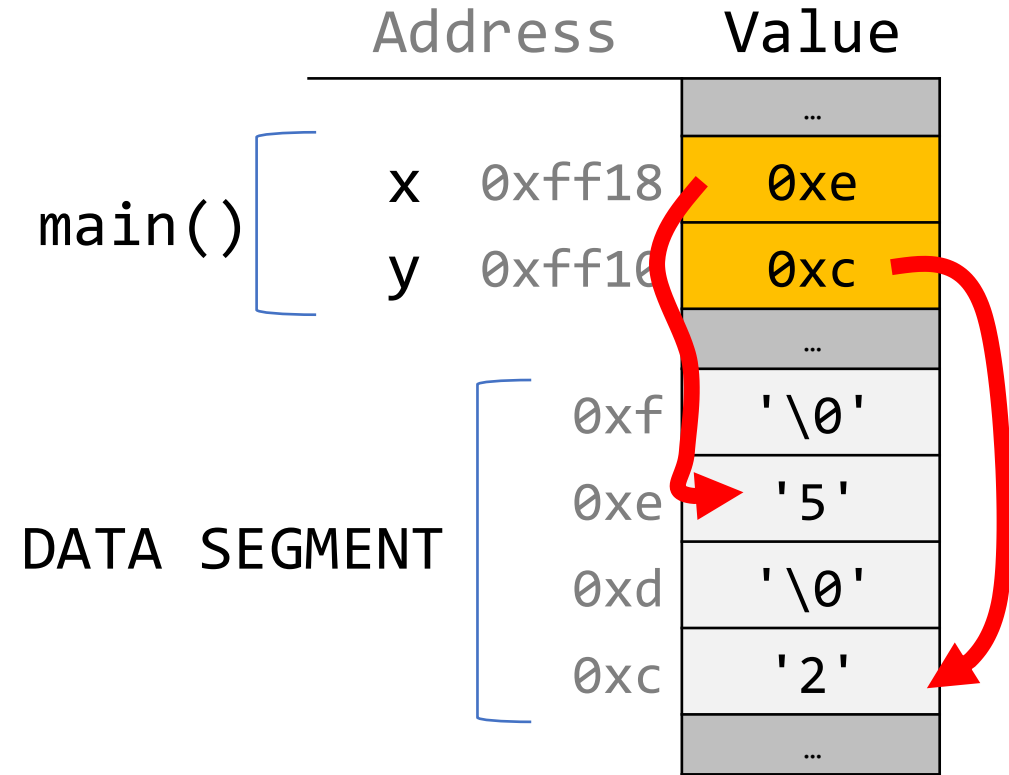
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

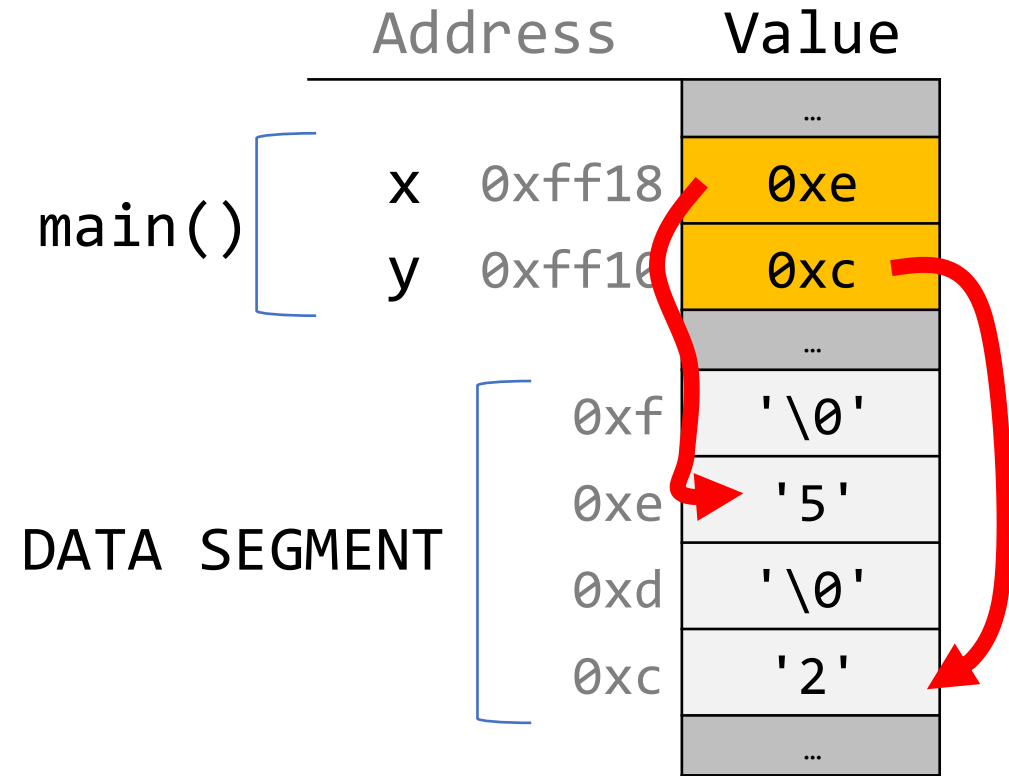
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

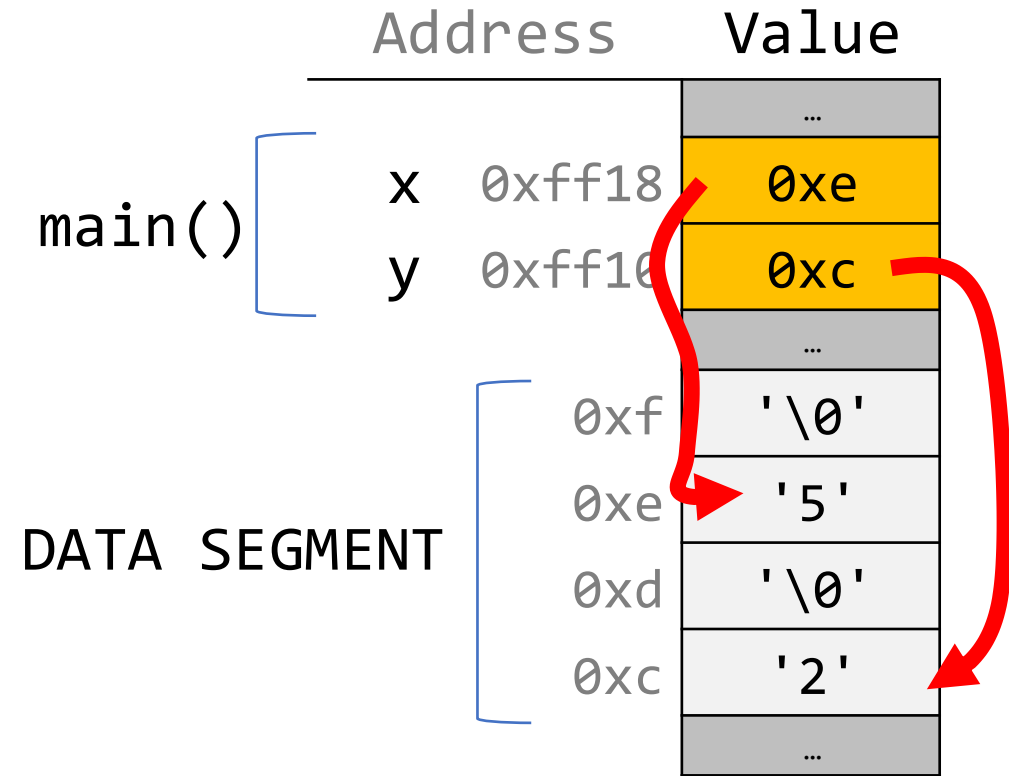
```
int main(int argc, char *argv[]) {  
    char *x = "2";  
    char *y = "5";  
    swap_string(&x, &y);  
    // want x = 5, y = 2  
    printf("x = %s, y = %s\n", x, y);  
    return 0;  
}
```



Swap

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

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



“Awesome! Thanks.”

“Awesome! Thanks. We also have 20 custom struct types. Could you write swap for those too?”



“Awesome! Thanks. We also have 20 custom struct types. Could you write swap for those too?”



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

Generic Swap

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

```
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

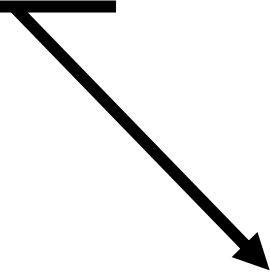
```
void swap(void *data1ptr, void *data2ptr, size_t nbytes) {  
    char temp[nbytes];  
    // store a copy of data1 in temporary storage  
    temp = *data1ptr; ???  
    // copy data2 to location of data1  
    // copy data in temporary storage to location of data2  
}
```

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);
```



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

memcpy

memcpy is a function that copies a specified amount of bytes at one address to another address.

```
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

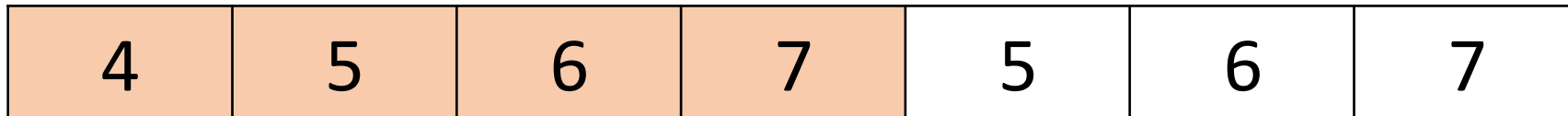
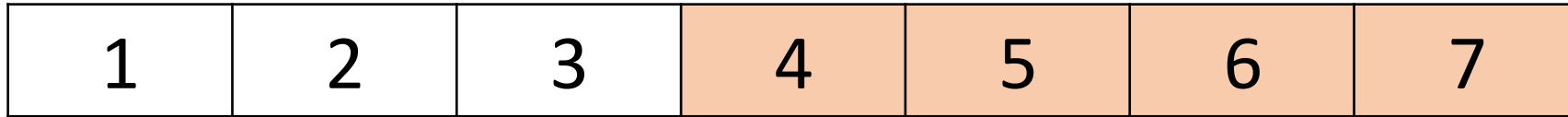
memmove is the same as `memcpy`, but supports overlapping regions of memory. (Unlike its name implies, it still “copies”).

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

It copies the next `n` bytes that `src` points to 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

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

void * Pitfalls

- `void *`s are powerful, but dangerous - C cannot do as much checking!
- E.g. with `int`, C would never let you swap *half* of an `int`. With `void *`s, this can happen! (*How? Let's find out!*)

Demo: void *s Gone Wrong



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!



Lecture Plan

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

Swap Ends

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

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

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

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

Swap Ends

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

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

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

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

Swap Ends

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

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

Pointer Arithmetic

```
arr + nelems - 1
```

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

If it's an array of...

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

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

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

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

Swap Ends

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

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

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

`(nelems - 1) * elem_bytes`

Swap Ends

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

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

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

`(nelems - 1) * elem_bytes`

Swap Ends

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

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

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

Swap Ends

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

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

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

`char *` pointers already add bytes!

Swap Ends

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]));
```

Demo: Void *s Gone Wrong



swap_ends.c

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!

```
int x = 0xffffffff;
int y = 0xeeeeeeeee;
swap(&x, &y, sizeof(short));
```

```
// now x = 0xffffeeee, y = 0xeeeeffff!
printf("x = 0x%x, y = 0x%x\n", x, y);
```

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 Array Rotation

Next time: *Function Pointers*