photo by unsplash user @ servuspaul

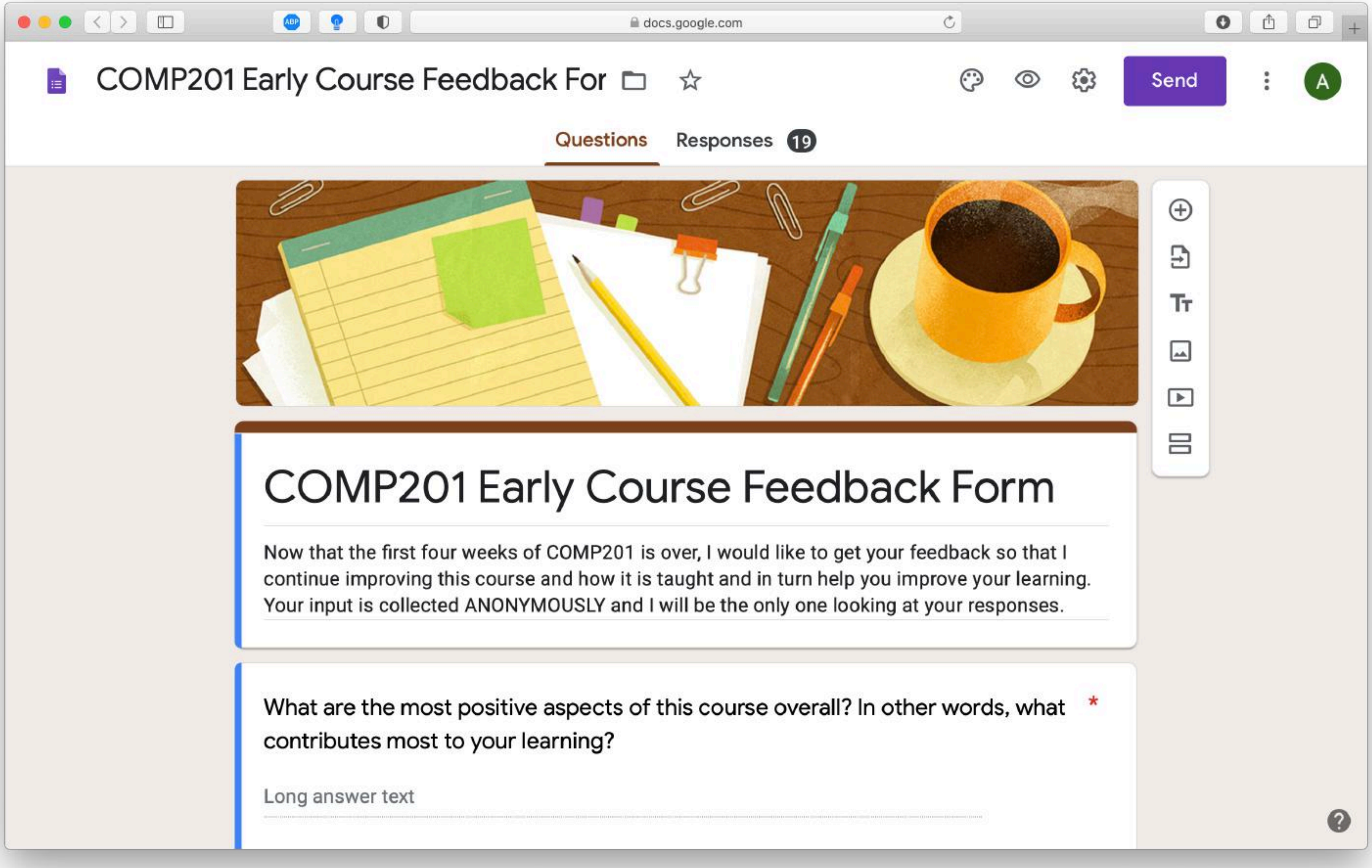
COMP201

Computer Systems & Programming

Lecture #12 – Other heap allocations, C Generics – `Void *`

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020



COMP201 Early Course Feedback Form

Now that the first four weeks of COMP201 is over, I would like to get your feedback so that I continue improving this course and how it is taught and in turn help you improve your learning. Your input is collected ANONYMOUSLY and I will be the only one looking at your responses.

What are the most positive aspects of this course overall? In other words, what contributes most to your learning? *

Long answer text

https://forms.gle/5d8LDWfH84pC33Fg7

Recap

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Plan for Today

- Other heap allocations
- **Overview:** Generics
- Generic Swap

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- Other heap allocations
- **Overview:** Generics
- Generic Swap

Recap: `malloc`

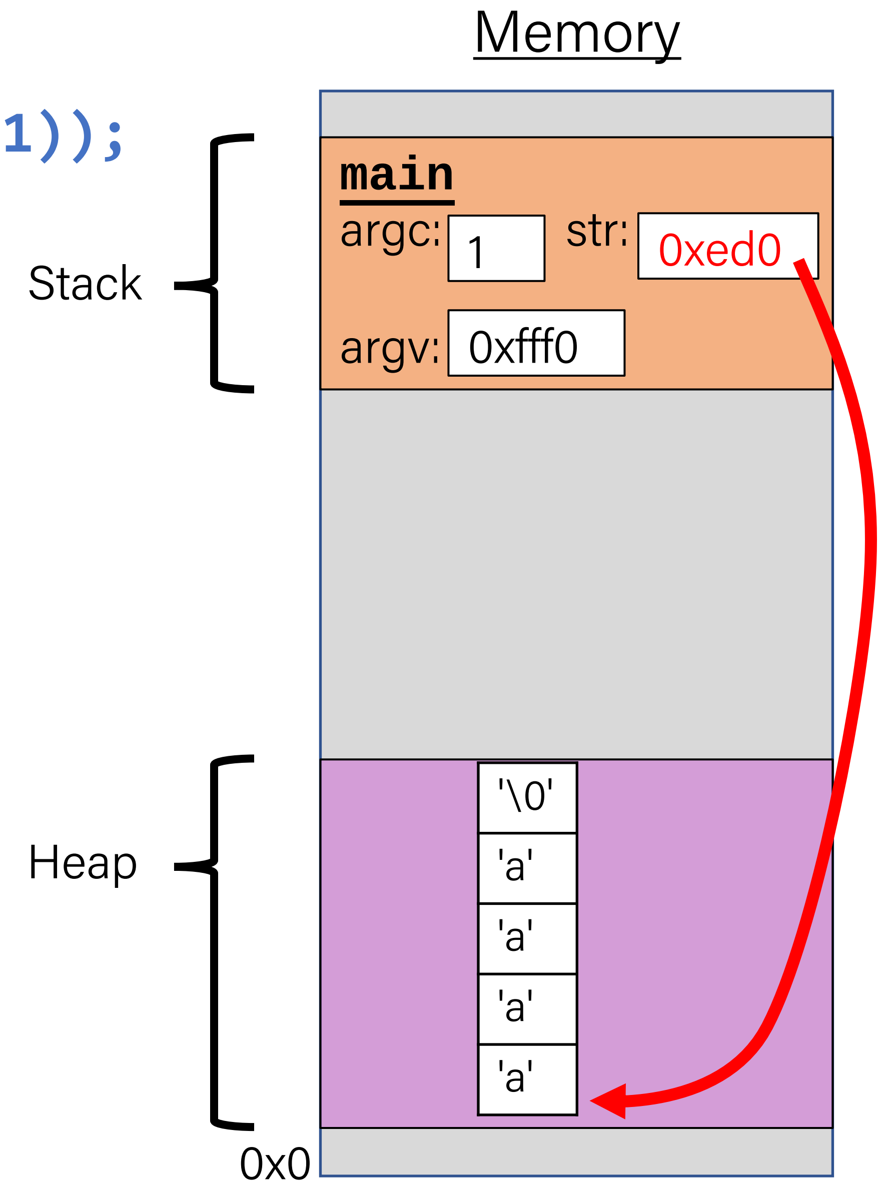
```
void *malloc(size_t size);
```

To allocate memory on the heap, use the **`malloc`** function (“memory allocate”) and specify the number of bytes you’d like.

- This function returns a pointer to *the **starting address** of the new memory*. It doesn’t know or care whether it will be used as an array, a single block of memory, etc.
- **`void *`**means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns `NULL`, then there wasn’t enough memory for this request.

Recap: malloc

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



Other heap allocations: calloc

```
void *calloc(size_t nmemb, size_t size);
```

calloc is like malloc that **zeros out** the memory for you—thanks, calloc!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (**nmemb** * **size**).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- calloc is more expensive than malloc because it zeros out memory. Use only when necessary!

Other heap allocations: `strdup`

```
char *strdup(char *s);
```

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself.

```
char *str = strdup("Hello, world!");  // on heap
str[0] = 'h';
```

Implementing strdup

How can we implement **strdup** using functions we’ve already seen?

```
char *myStrdup(char *str) {
    char *heapStr = malloc(strlen(str) + 1);
    assert(heapStr != NULL);
    strcpy(heapStr, str);
    return heapStr;
}
```

Cleaning Up with `free`

```
void free(void *ptr);
```

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

`free` details

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);   ⇦ ✅
…
free(ptr);     ⇦ ❌
```

❌ Memory at this address was already freed!

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);       ⇦ ✅
…
free(ptr + 1);     ⇦ ❌
```

Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *str = strdup("Hello!");
…
free(str);    // our responsibility to free!
```

Memory Leaks

- A memory leak is when you allocate memory on the heap, but do not free it.
- Your program should be responsible for cleaning up any memory it allocates but no longer needs.
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap!

However, memory leaks rarely (if ever) cause crashes.

- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.
- Valgrind is a very helpful tool for finding memory leaks!

free Practice

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
     assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
}
 printf("%s\n", str);
```

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
}
 printf("%s\n", str);
```

NEW Head for **www.slido.com** and enter your answers using the event code **#73165**

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```c
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
    free(num);
}
printf("%s\n", str);
free(str);
```

Demo: Pig Latin

pig_latin.c

realloc

```
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.
- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.
- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

realloc

- realloc only accepts pointers that were previously returned my malloc/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc` —the previous (smaller) one was already reclaimed by `realloc`.

```c
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Heap allocator analogy: A hotel

Request memory by size (`malloc`)

- Receive room key to first of connecting rooms

Need more room? (`realloc`)

- Extend into connecting room if available
- If not, trade for new digs, employee moves your stuff for you

Check out when done (`free`)

- You remember your room number though

Errors! What happens if you...

- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes...
- Return to room after checkout?

Demo: Pig Latin Part 2

`pig_latin.c`

Heap allocation interface: A summary

```
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);
```

Compare and contrast the heap memory functions we've learned about.

🤔

Heap allocation interface: A summary

```
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

Engineering principles: stack vs heap

Stack (“local variables”)

Heap (dynamic memory)

- **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

Engineering principles: stack vs 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)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.
- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Lecture Plan

- Heap allocations
- **Overview:** Generics
- Generic Swap

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

Learning Goals

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

Generics

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

Lecture Plan

- Heap allocations
- **Overview:** Generics
- Generic Swap

Swap

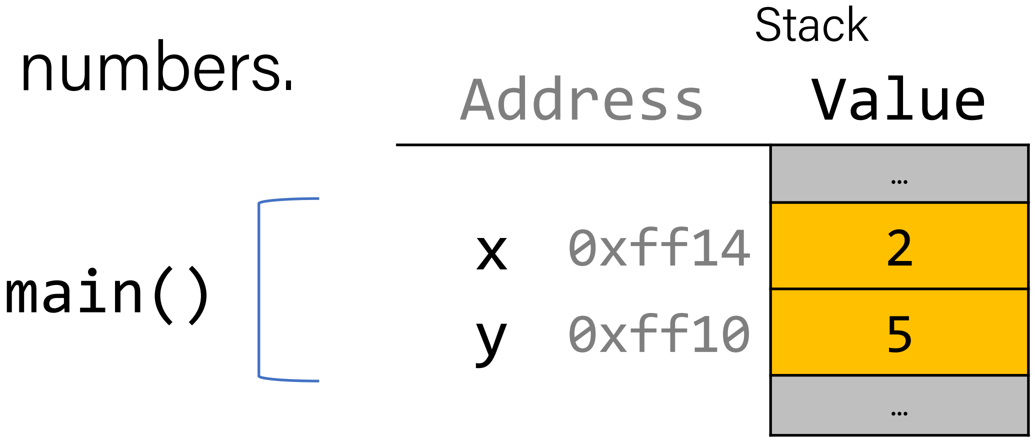
You’re asked to write a function that swaps two numbers.

```
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;
}
```



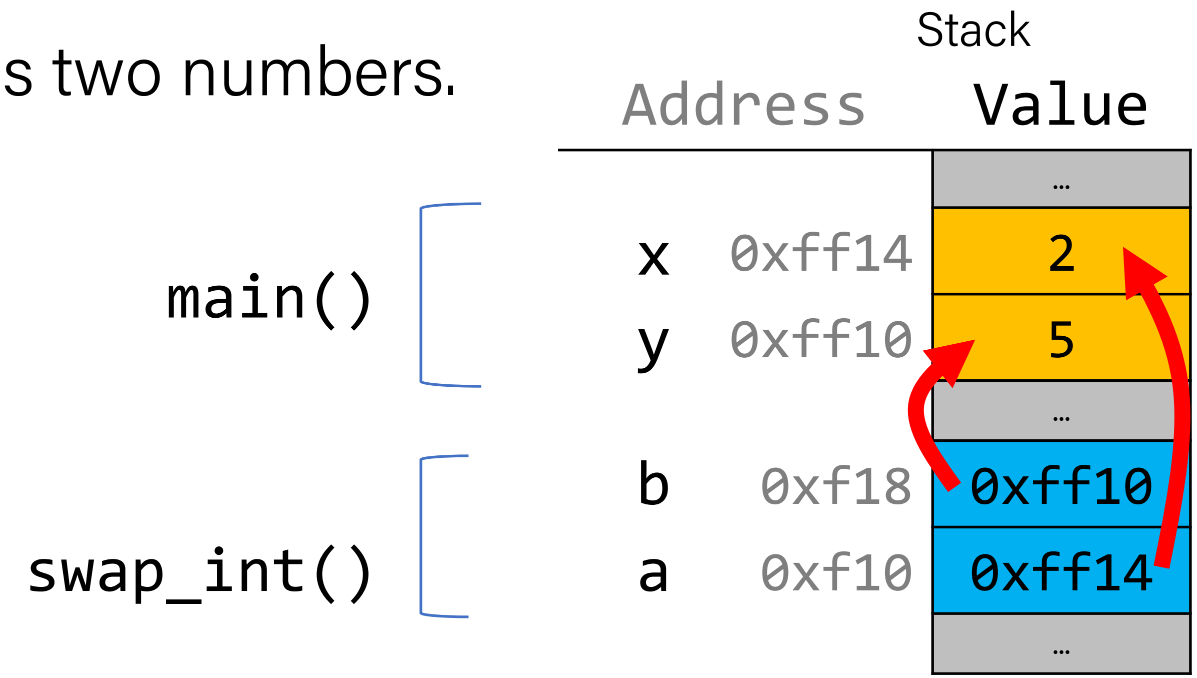
Stack

		Address	Value
			…
main()	x	0xff14	2
	y	0xff10	5
			…

Swap

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

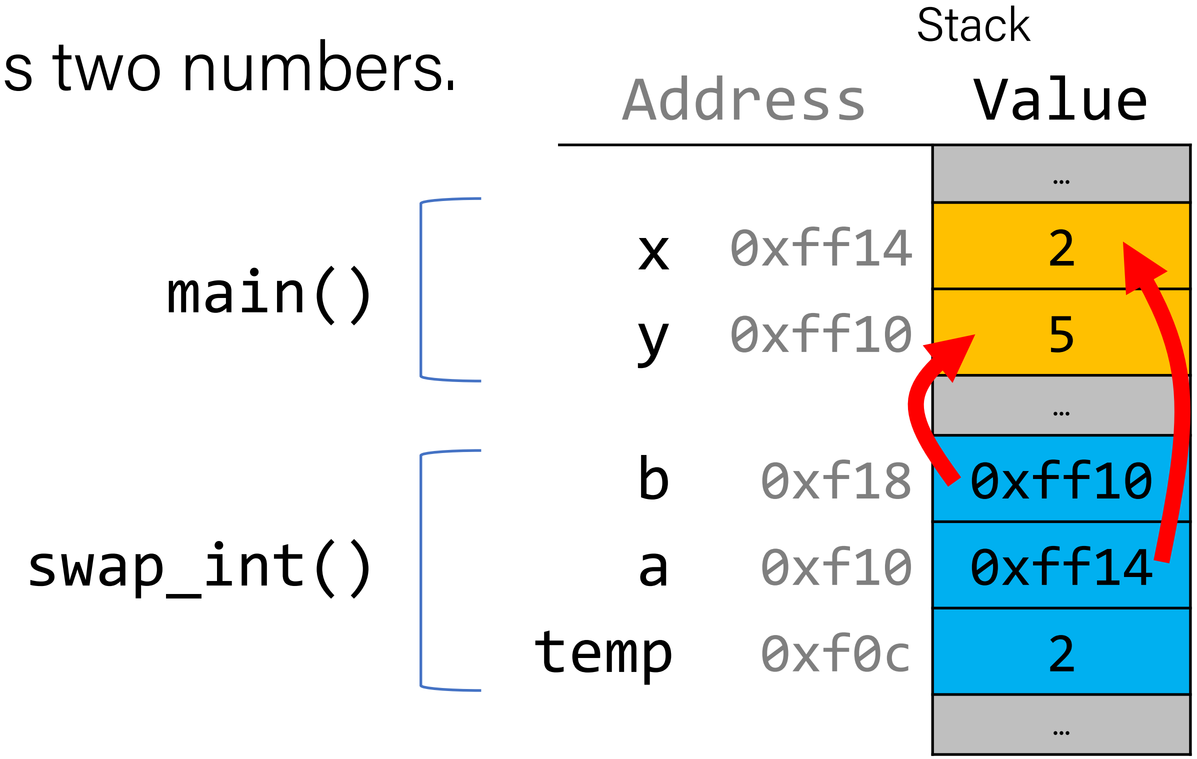
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```



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;
}
```



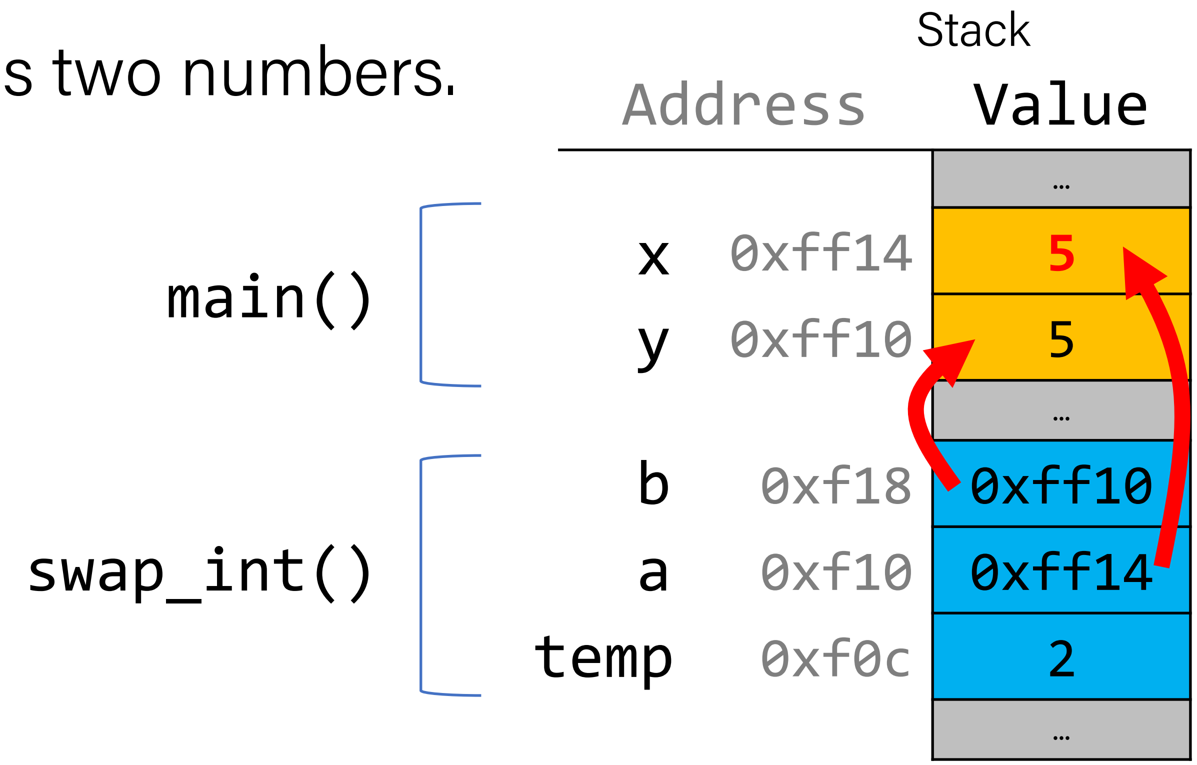
Stack

		Address	Value
			…
main()	x	0xff14	2
	y	0xff10	5
			…
swap_int()	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;
}
```

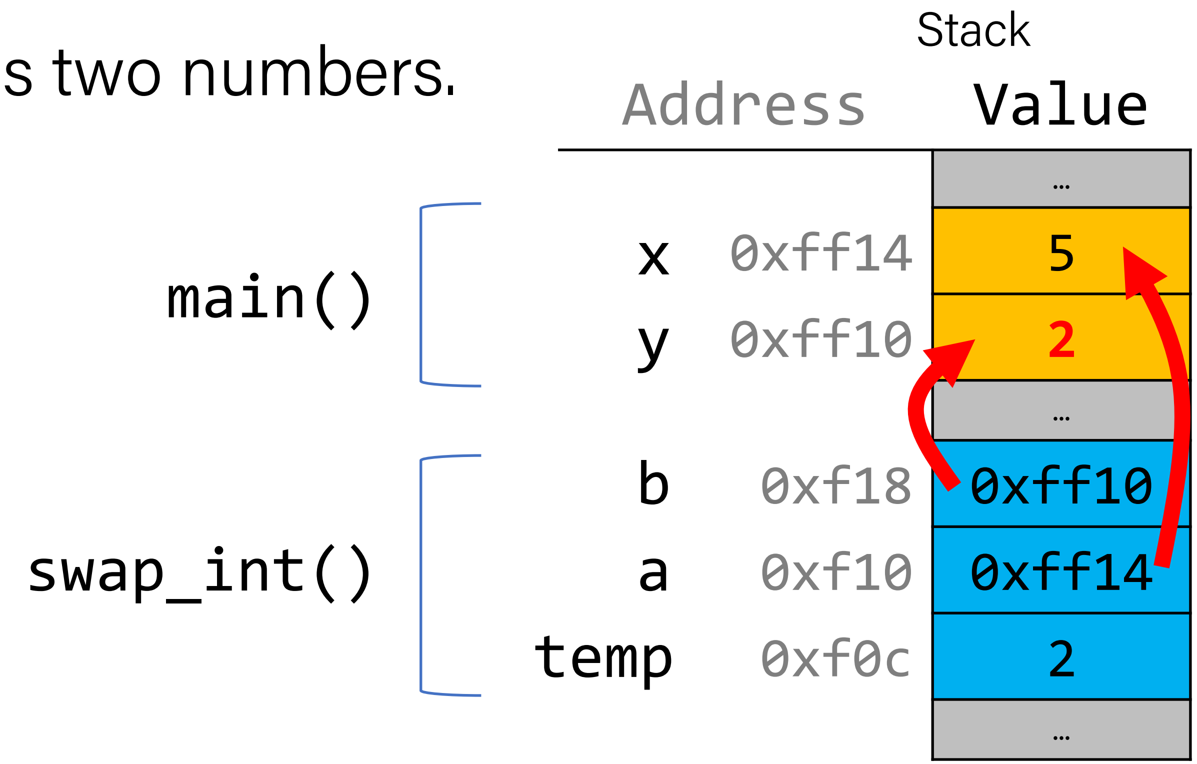


			Stack
		Address	Value
			…
main()	x	0xff14	5
	y	0xff10	5
			…
swap_int()	b	0xf18	0xff10
	a	0xf10	0xff14
	temp	0xf0c	2
			…

Swap

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

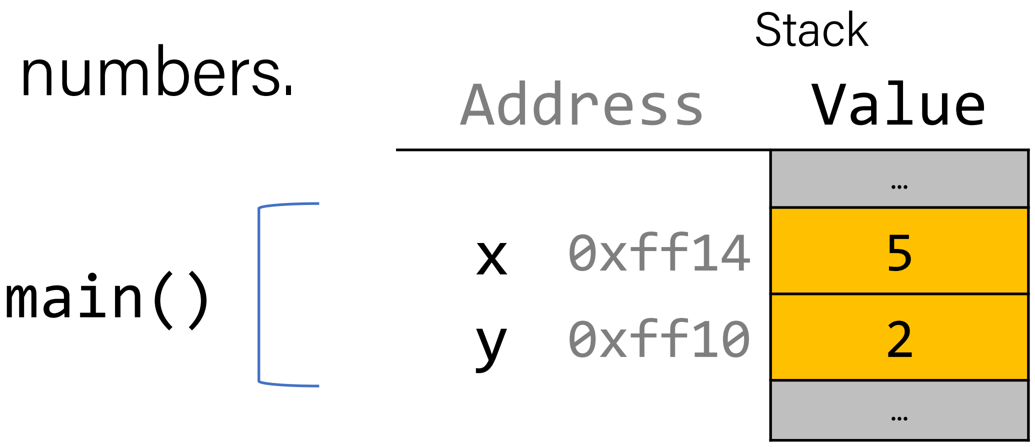
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```



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;
}
```



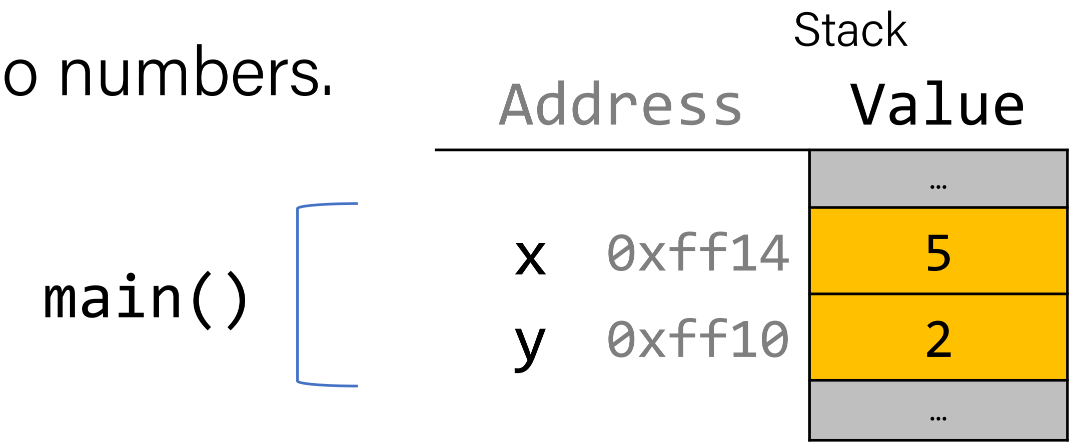
Stack

		Address	Value
			…
main()	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()

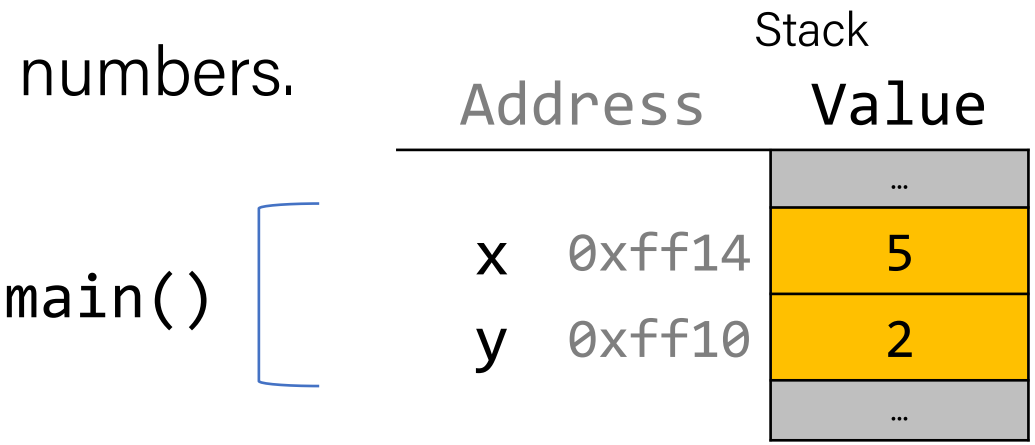
Stack

	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`.”

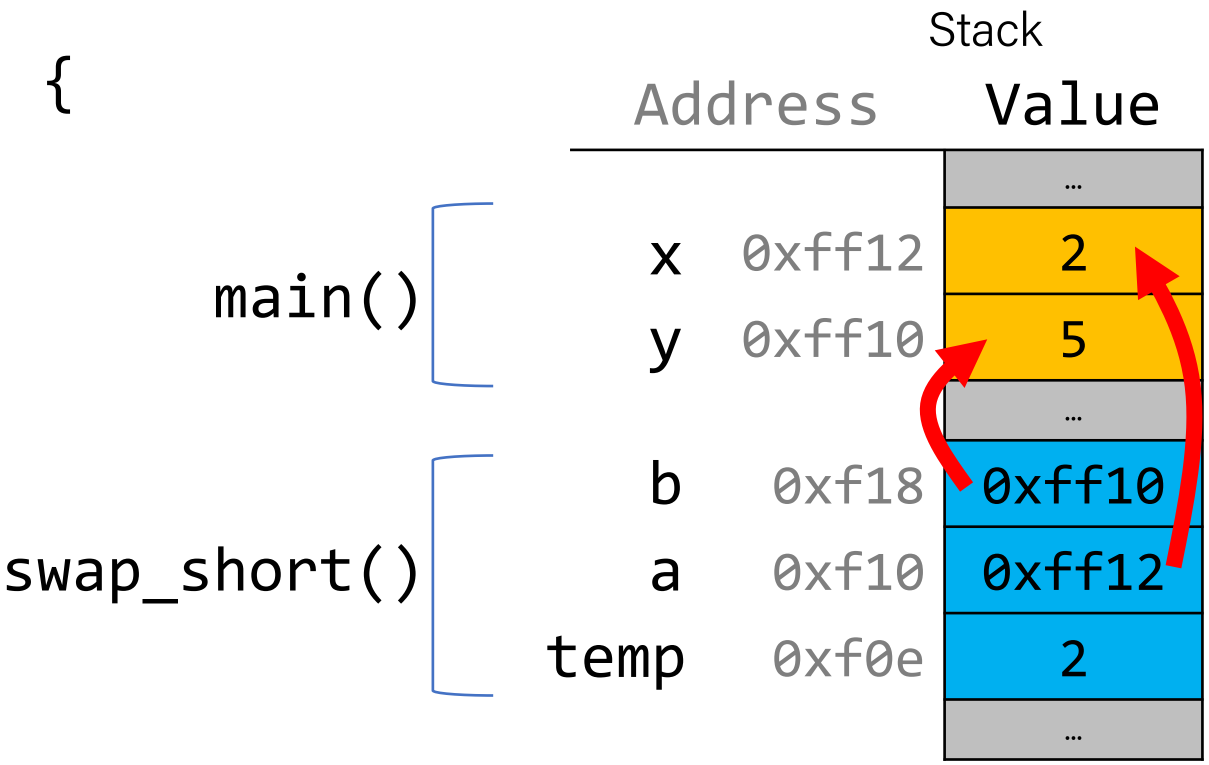
😑

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;
}
```



Stack

		Address	Value
			…
main()	x	0xff12	2
	y	0xff10	5
			…
swap_short()	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?”

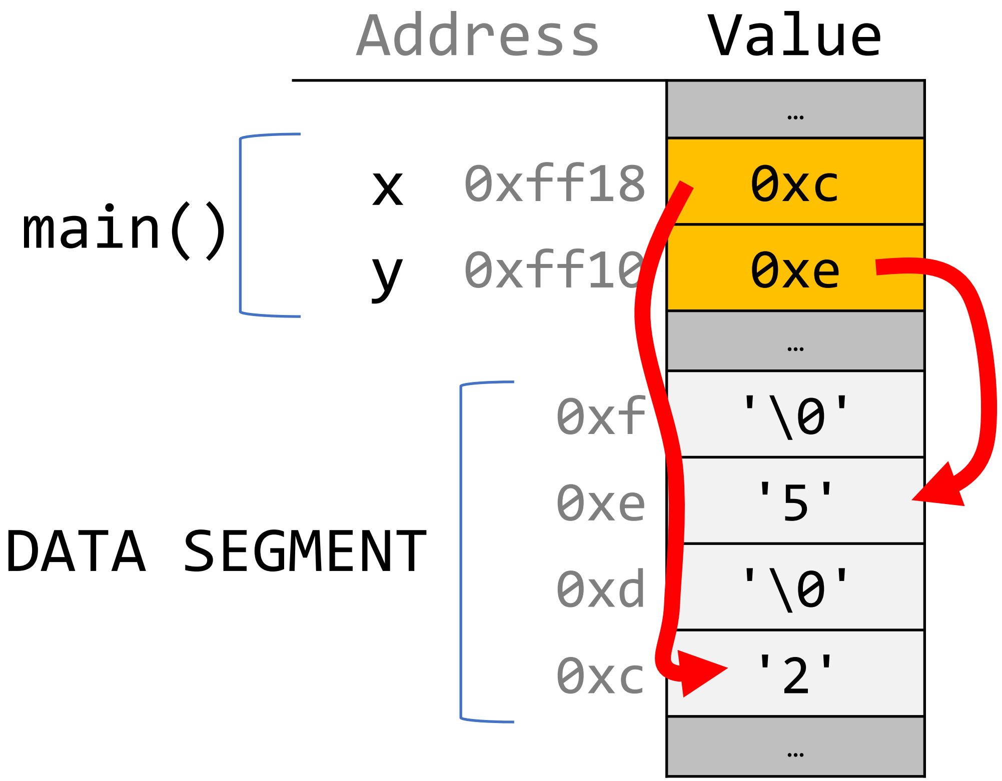
😤

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;
}
```

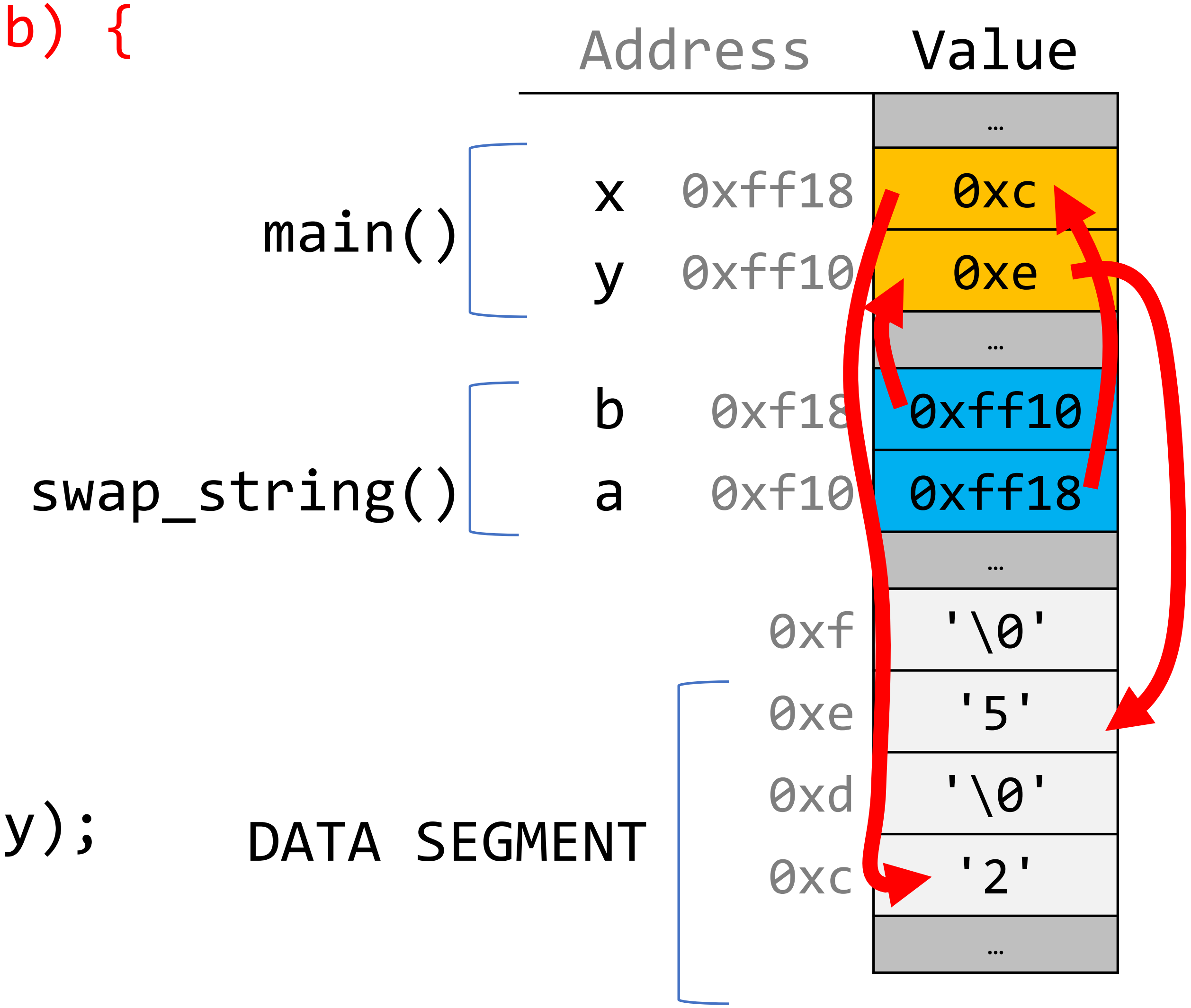
Swap

```
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;
}
```



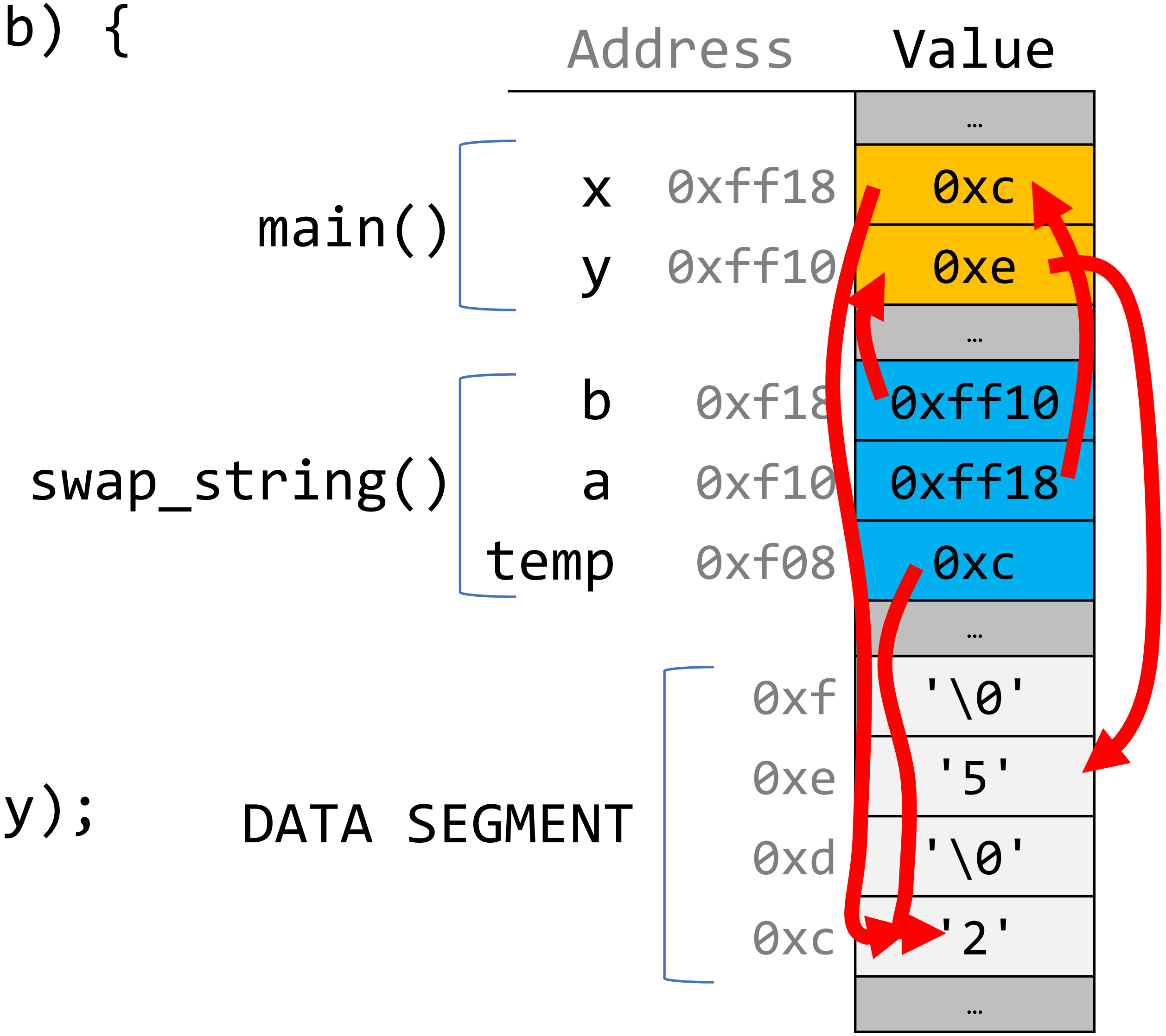
Swap

```
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

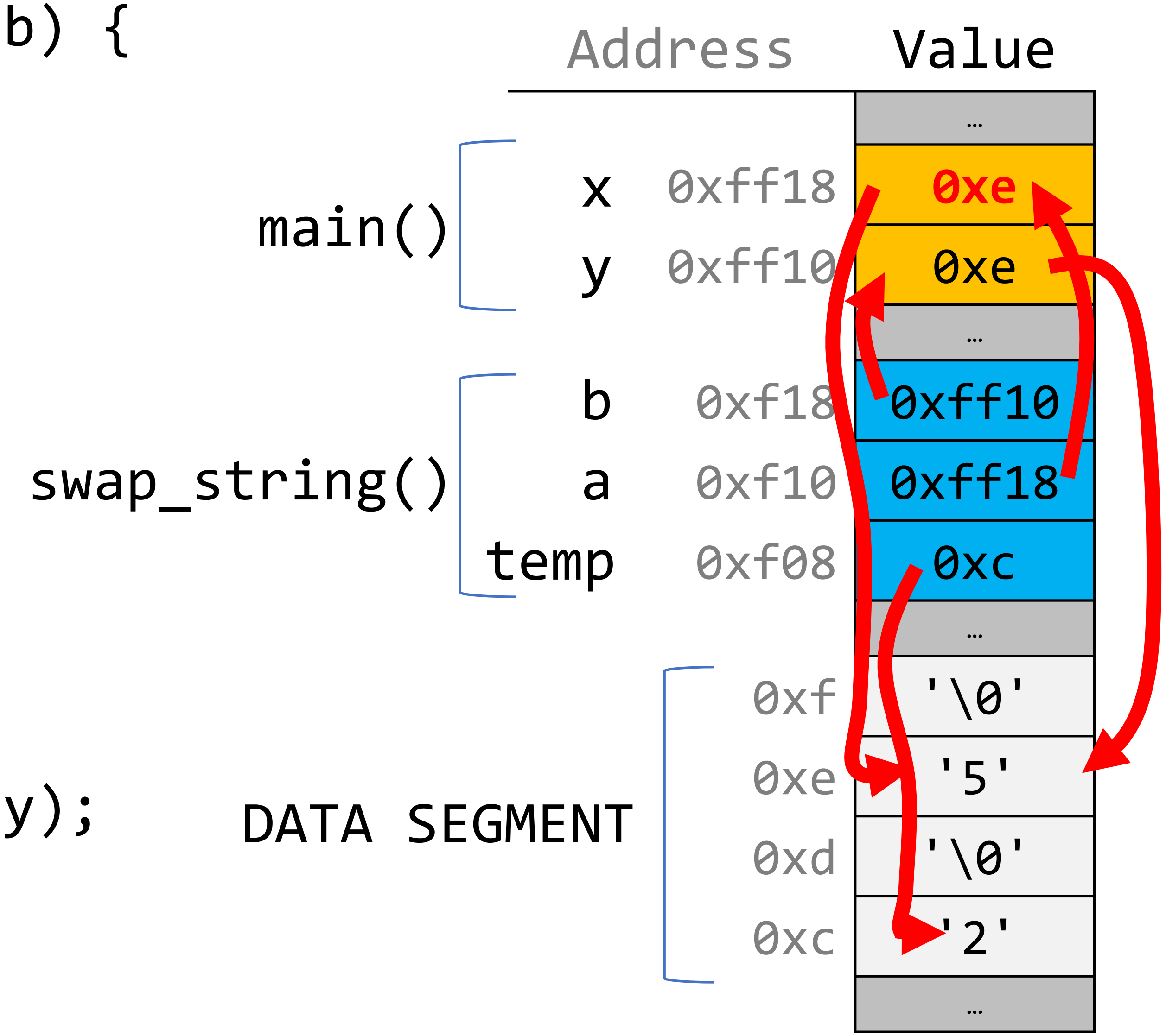
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```



		Address	Value
			…
main()	x	0xff18	0xc
	y	0xff10	0xe
			…
swap_string()	b	0xf18	0xff10
	a	0xf10	0xff18
	temp	0xf08	0xc
			…
DATA SEGMENT		0xf	'\0'
		0xe	'5'
		0xd	'\0'
		0xc	'2'
			…

Swap

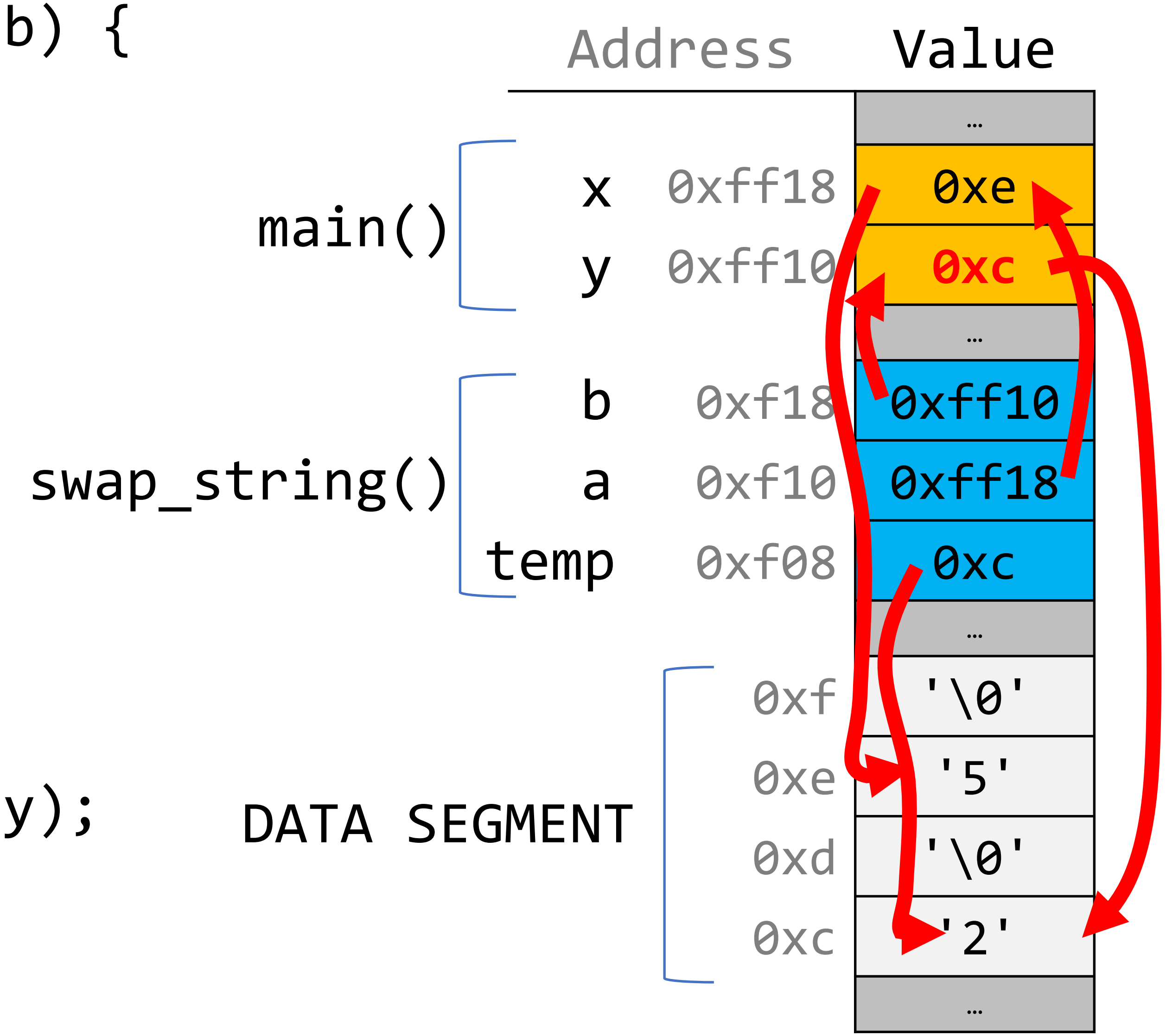
```
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```



		Address	Value
			…
main()	x	0xff18	0xe
	y	0xff10	0xe
			…
swap_string()	b	0xf18	0xff10
	a	0xf10	0xff18
	temp	0xf08	0xc
			…
DATA SEGMENT		0xf	'\0'
		0xe	'5'
		0xd	'\0'
		0xc	'2'
			…

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;
}
```

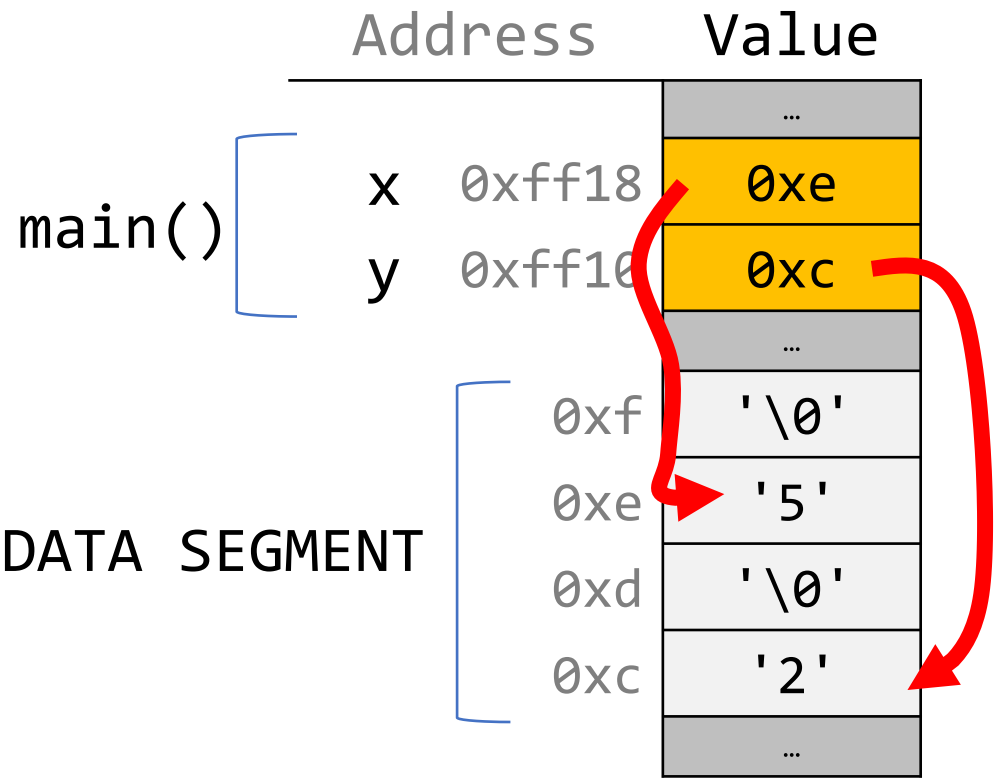


		Address	Value
			…
main()	x	0xff18	0xe
	y	0xff10	0xc
			…
swap_string()	b	0xf18	0xff10
	a	0xf10	0xff18
	temp	0xf08	0xc
			…
DATA SEGMENT		0xf	'\0'
		0xe	'5'
		0xd	'\0'
		0xc	'2'
			…

Swap

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

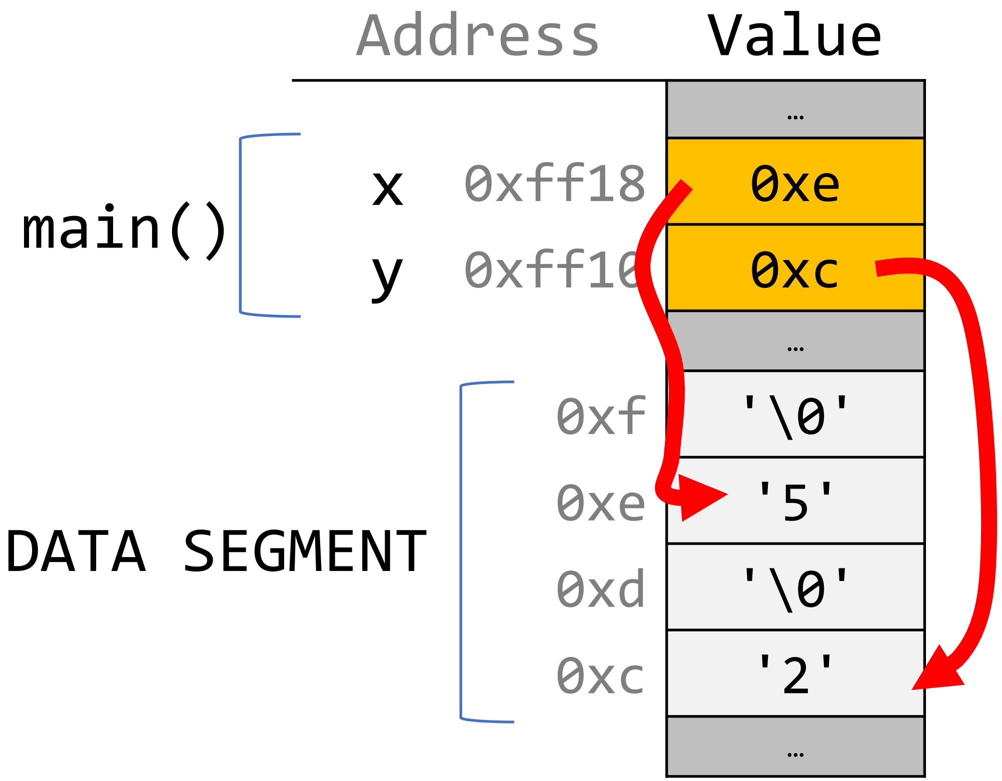
```
int main(int argc, char *argv[]) {
    char *x = "2";
    char *y = "5";
    swap_string(&x, &y);
    // want x = 5, y = 2
    printf("x = %s, y = %s\n", x, y);
    return 0;
}
```



		Address	Value
			…
main()	x	0xff18	0xe
	y	0xff10	0xc
			…
DATA SEGMENT		0xf	'\0'
		0xe	'5'
		0xd	'\0'
		0xc	'2'
			…

Swap

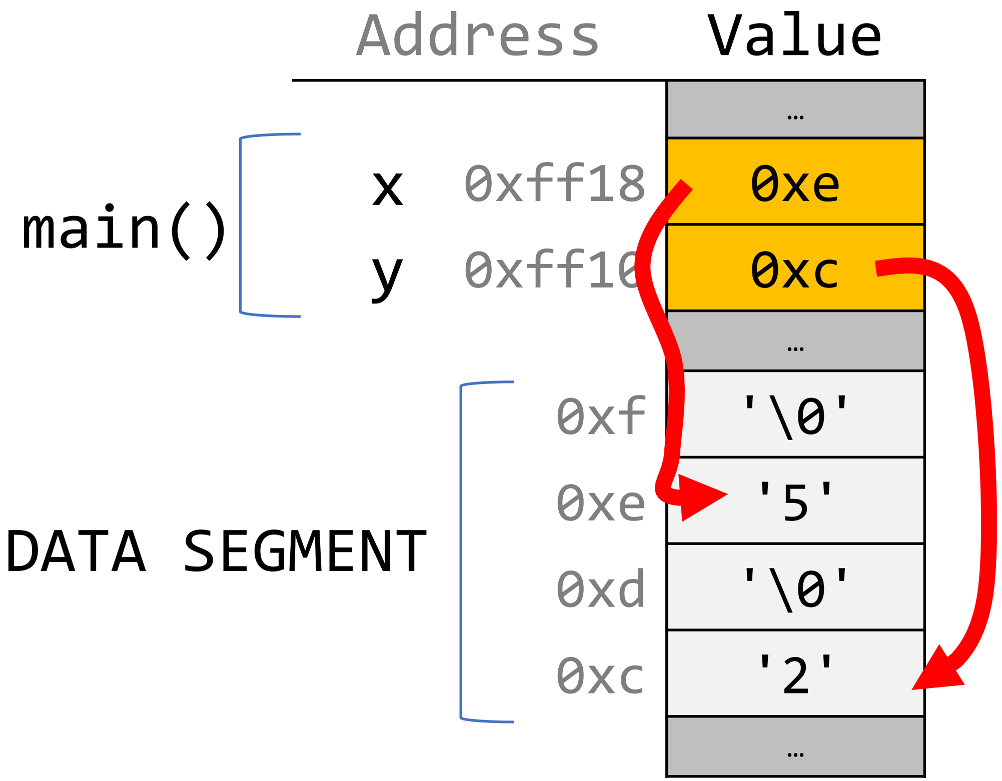
```c
void swap_string(char **a, char **b) {
    char *temp = *a;
    *a = *b;
    *b = temp;
}

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



		Address	Value
			…
main()	x	0xff18	0xe
	y	0xff10	0xc
			…
DATA SEGMENT		0xf	'\0'
		0xe	'5'
		0xd	'\0'
		0xc	'2'
			…

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?

Recap

- Heap allocations
- **Overview:** Generics
- Generic Swap

Next time: More Generics, and Function Pointers