CS107, Lecture 12
Disclosure, Partiality, Generics and Void *

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

CS107 Topic 4

<u>How can we use our knowledge of memory and data representation to write code that works with any data type?</u>

Why is answering this question important?

- Writing code that works with any data type lets us write more generic, reusable code while understanding potential pitfalls (today)
- Allows us to learn how to pass functions as parameters, a core concept in many languages (next time)

assign4: implement your own version of the **ls** command, a function to generically find and insert elements into a sorted array, and a program using that function to sort the lines in a file like the **sort** command.

Learning Goals

- Learn about the potential harm from vulnerabilities, challenges to proper disclosure of vulnerabilities, and how we weigh competing interests
- Learn how to write C code that works with any data type.
- Learn about how to use void * and avoid potential pitfalls.

Lecture Plan

- Vulnerabilities, disclosure and partiality
- **Overview:** Generics
- Generic Swap
- Generics Pitfalls

```
cp -r /afs/ir/class/cs107/lecture-code/lect12 .
```

Lecture Plan

- **Vulnerabilities, disclosure and partiality**
- **Overview:** Generics
- Generic Swap
- Generics Pitfalls

```
cp -r /afs/ir/class/cs107/lecture-code/lect12 .
```

Use-After-Free

"Use-After-Free" is a bug where you continue to use heap memory after you have freed it.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);
strncpy(ptr, argv[1], 3);
```

We freed bytes but did not set ptr to NULL

❌ Memory at this address was already freed, but now we are using it!

<u>This is possible because **free()** doesn't change the pointer passed in, it just frees the memory it points to.</u>

Use-After-Free

- What happens when we have a use-after-free bug? ***Undefined Behavior / a memory error!***
  - Maybe the memory still has its original contents?
  - Maybe the memory is used to store some other heap data now?
- Use-after-free is not just a functionality issue; it can cause a range of unintended behavior, including accessing/modifying memory you shouldn't be accessing

It's our job as programmers to find and fix use-after-free and other bugs not just for the functional correctness of our programs, but to protect people who use and interact with our code.

Use-After Free as a Vulnerability

- Use After Free Vulnerabilities in CVE database
- Use-after-free in Chrome (2020)
- Google's attempts to reduce Chrome use-after-free vulnerabilities (2021)
- Use-after-free in iOS (2020)
- Google 2023 Chrome fixes include use-after-free vulnerability and heap buffer overflow (2023)
- Adobe Acrobat Reader use-after-free vulnerability enables arbitrary code execution (2023)

What should someone do if they find a vulnerability? How can we incentivize responsible disclosure?

Disclosure

Various roles in this process: **users** (those at risk), **makers** (e.g., software company), **security researchers** (who found the vulnerability), **bad actors** (who wish to exploit the issue to harm users), etc.

- Users want to be protected with secure software
- Makers want to make their software secure and not have it exploited – they probably want to have time to fix vulnerabilities before they are made public
- Security researchers want their issues to be fixed and be rewarded for finding them
- Bad actors want to learn about vulnerabilities before they are patched

Full Disclosure

One approach is to make vulnerabilities public as soon as they are found. Vulnerabilities unknown to the software maker before release are called “zero-day vulnerabilities” because they “have 0 days to fix the problem”.

- puts pressure on the maker to fix it quickly
- discloses the vulnerability to the public as soon as it’s found
- Leaves users vulnerable until the maker releases a patch

Few people now endorse this approach due to its drawbacks.

Responsible Disclosure

Another approach is to privately alert the software maker to the vulnerability to fix it in a reasonable amount of time before publicizing the vulnerability. This is called "responsible disclosure":

- Contacts the makers of the software
- Informs them about the vulnerability
- Negotiates a reasonable timeline for a patch or fix
- Considers a deadline extension if necessary

time passes while the developers fix the bug

- Works with the developers to add the vulnerability to CVE Details https://www.cvedetails.com/ , from which it is added to the National Vulnerability Database https://nvd.nist.gov/

Responsible Disclosure

Responsible disclosure is the most common approach, and it is recommended by the ACM code of ethics:

Responsible disclosure is the approach more consistent with the ACM Code of Ethics. By keeping the existence of the vulnerability secret for a longer amount of time, it reduces the chance of harm to others (Principle 1.2). It also supports more robust patching (Principles 2.1, 2.9, and 3.6), as the company can take more time to develop the patch and confirm that it will not induce unintended consequences. Full disclosure puts individuals at risk of harm sooner, and those harms may be irreversible and onerous (contravening Principles 1.2 and 3.1). As such, full disclosure should the exception and should only be used when attempts at responsible disclosure have failed. Furthermore, the individual committing to the full disclosure needs to consider carefully the risks that they are imposing on others and be willing to accept the moral and possibly legal consequences (Principles 2.3 and 2.5).

Vulnerability Commercialization

Various entities may want to financially reward people for finding and reporting vulnerabilities:

- Software makers want to know about vulnerabilities in their software
- Other entities want to know about unpatched vulnerabilities to exploit them

Bug Bounty Programs

Many companies now offer “Bug Bounties,” or rewards for responsible disclosure.

Good Version of a bug bounty process:

- Responsible disclosure process is followed
- Company is buying information & time to fix the bug

Bad version of a bug bounty process:

- Company does not fix the bug *or* notify the public.
- Not knowing what vulnerabilities exist makes it harder for users to calibrate trust
- Company is effectively buying silence

Vulnerabilities Equities Process

The US federal government is one of the largest discoverers and purchasers of 0-day vulnerabilities.

It follows a “Vulnerabilities Equities Process” (VEP) to determine which vulnerabilities to responsibly disclose and which to keep secret and use for espionage or intelligence gathering.

VEP claimed in 2017 that 90% of vulnerabilities are disclosed, but it is not clear what the impact or scope of the un-disclosed 10% of vulnerabilities are.

More reading here and here

Concerns with VEP

- Lack of transparency: little oversight as to whether the “bias towards responsible disclosure” is consistently upheld
- Harm of omission: withholding the opportunity to fix the vulnerability means that another actor could re-discover and use it
- Risk of stockpiling: Other people can hack into the stored 0-days and use them, as in the “Shadowbrokers” attack which led to serious ransomware attacks on hospitals and transportation systems
- Intended use: NSA’s intended use of vulnerabilities may be concerning, as in PRISM surveillance program.

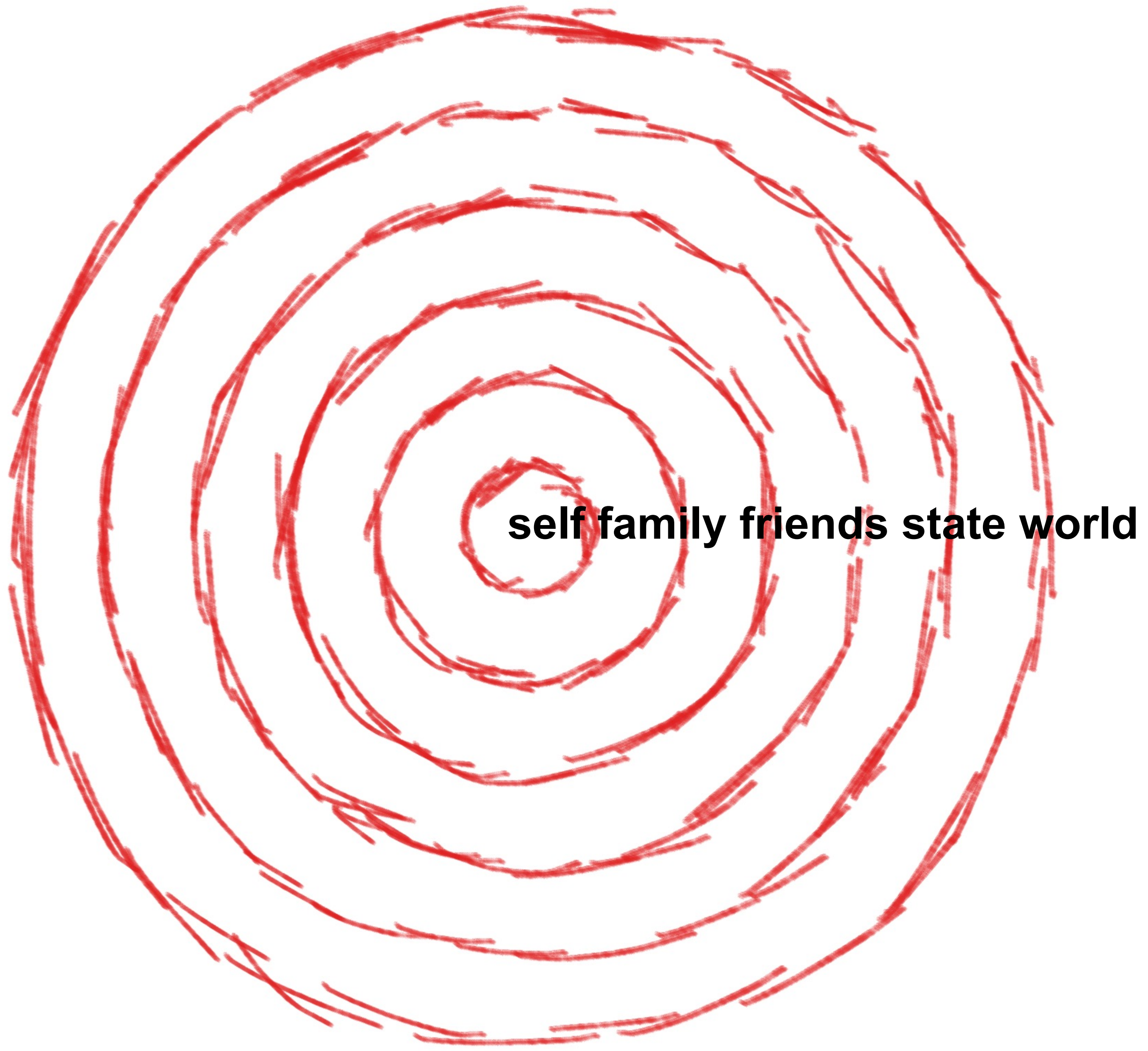
How do we weigh competing stakeholder interests here, such as country vs. individual?

Partiality

Partiality holds that it is acceptable to give preferential treatment to some people based on our relationships to them or shared group membership with them.

Impartiality, involves “acting from a position that acknowledges that all persons are ... equally entitled to fundamental conditions of well-being and respect.”

Partiality



Degrees of Partiality

Partiality: preference towards own family, friends, and state is morally acceptable or even required

Partial Cosmpolitanism: limited preference towards own state acceptable

Universal Care: preference towards family acceptable but not towards state

Impartial Benevolence: same moral responsibilities towards all people

Case Study: EternalBlue

2012-2017: NSA secretly stores the EternalBlue Microsoft vulnerability and uses it to spy on both US and non-US citizens.

early 2017: EternalBlue stolen by hacker group the ShadowBrokers. NSA discloses EternalBlue to Microsoft.

March 14, 2017: Microsoft releases a patch for the vulnerability.

May 12, 2017: EternalBlue is the basis of the WannaCry and other ransomware attacks, leading to downtime in critical hospital and city systems and over $1 billion of damages.

Microsoft’s Argument

“[T]his attack provides yet another example of why the **stockpiling of vulnerabilities** by governments is such a problem. ...

We need governments to consider the **damage to civilians** that comes from hoarding these vulnerabilities and the use of these exploits.

This is one reason we called in February for a new “Digital Geneva Convention” to govern these issues, including a **new requirement for governments to report vulnerabilities to vendors**, rather than stockpile, sell, or exploit them.

And it’s why we’ve pledged our support for **defending every customer everywher**e in the face of cyberattacks**, regardless of their nationality**.”

Full post here

Critical Questions

- Do we have special obligations to our own country and to protect our people? If so, what would this mean?
- If intentionally exploiting a vulnerability is wrong when done by a private citizen, is it equally wrong when done by the government?
- Should I be loyal to my country, a citizen of the world, or both?
- When should I give preference to my family members and when should I strive to treat all equally?

What you choose matters – the moral obligations you take on constitute who you are.

Revisiting EternalBlue

Federal Government ←

→ Microsoft

Partiality: preference towards own family, friends, and state is morally acceptable or even required

Partial Cosmpolitanism: limited preference towards own state acceptable

Universal Care: preference towards family acceptable but not towards state

Impartial Benevolence: same moral responsibilities towards all people

Partiality Takeaways

- Understanding partiality helps us understand how we balance cases of competing interests and where we may personally fall on this spectrum.
- In order to evaluate situations, it’s critical to understand the good and the bad that may come of it (e.g. EternalBlue). Better understanding privacy and privacy concerns is critical to this! (more later)

Lecture Plan

- Vulnerabilities, disclosure and partiality
- **Overview: Generics**
- Generic Swap
- Generics Pitfalls

```
cp -r /afs/ir/class/cs107/lecture-code/lect12 .
```

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? (Pointers are key!)

Lecture Plan

- Vulnerabilities, disclosure and partiality
- **Overview:** Generics
- **Generic Swap**
- Generics Pitfalls

```
cp -r /afs/ir/class/cs107/lecture-code/lect12 .
```

Swap

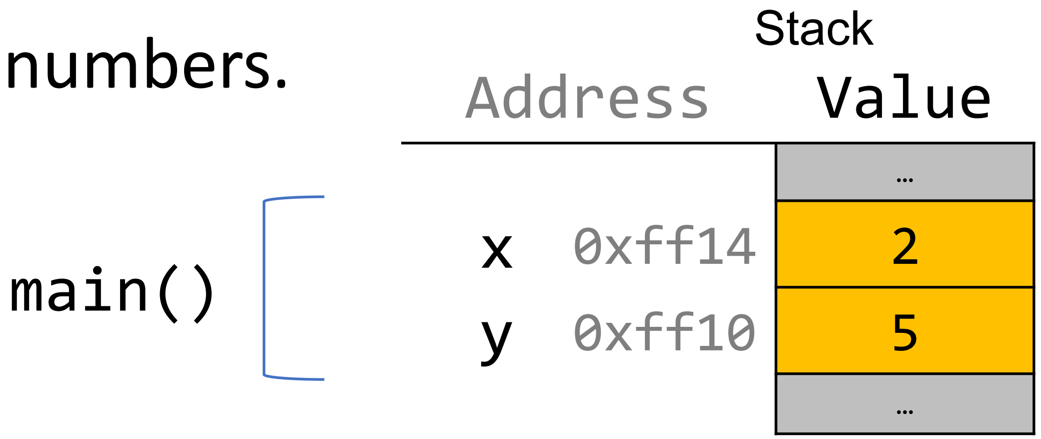
You’re asked to write a function that swaps two numbers.

```
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.

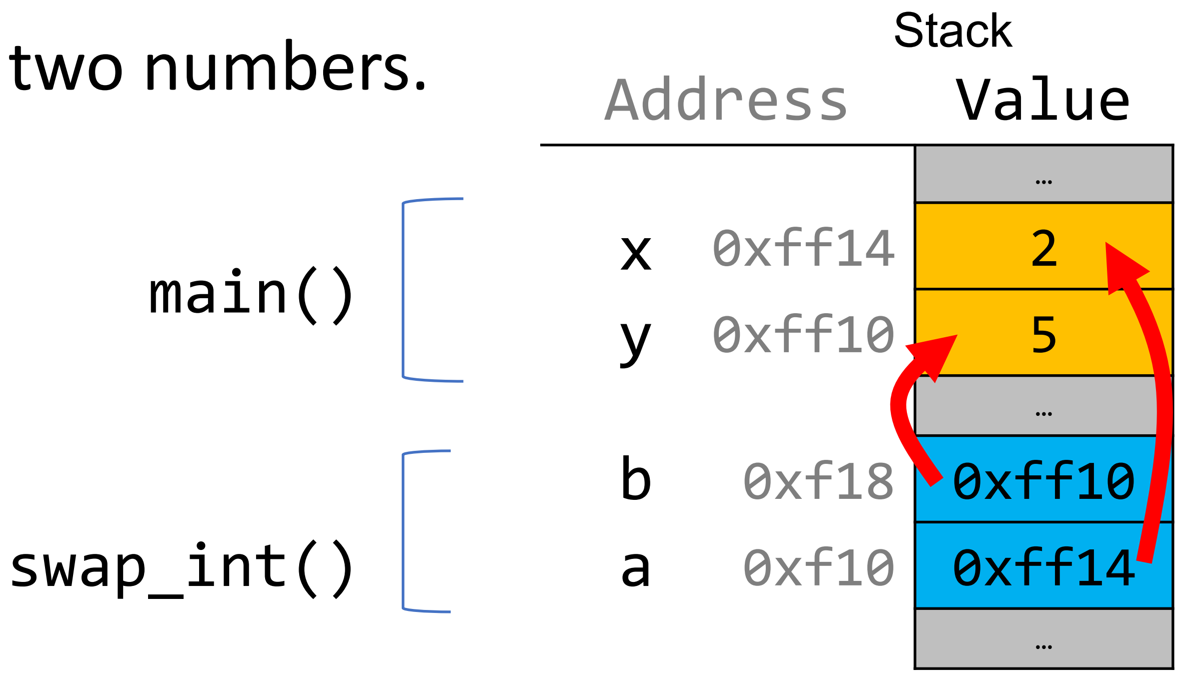
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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.

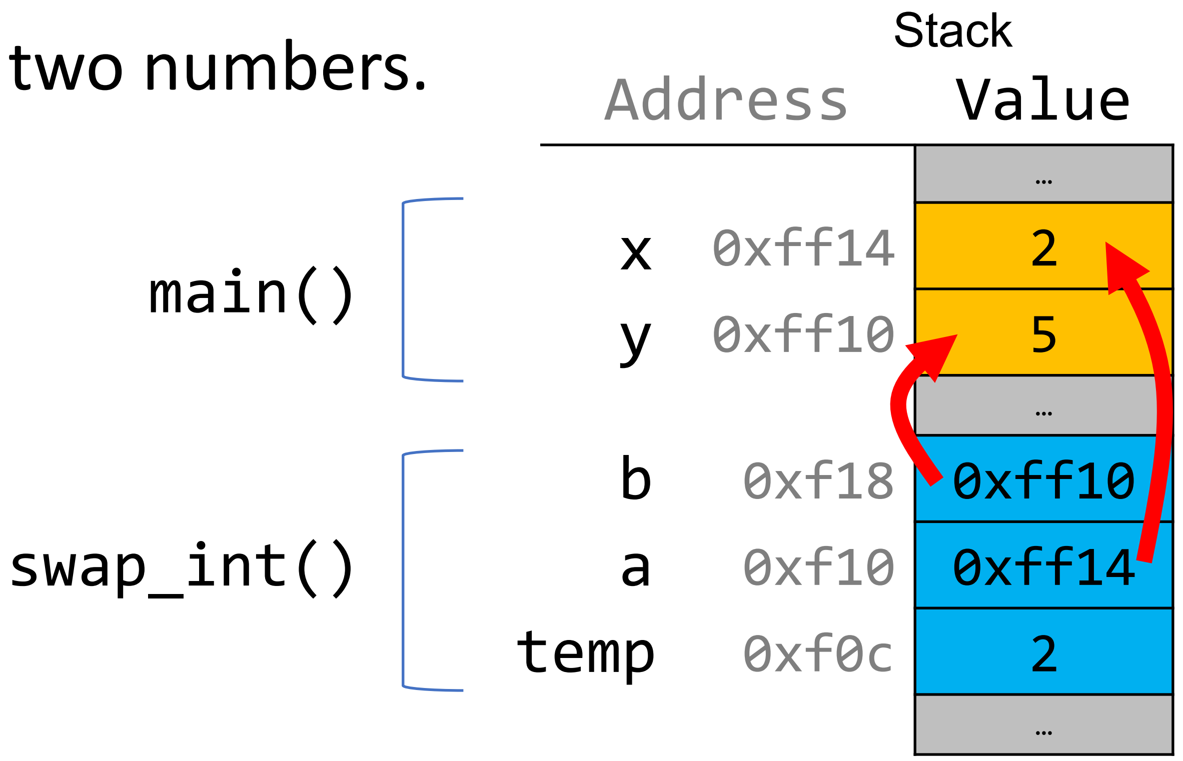
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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.

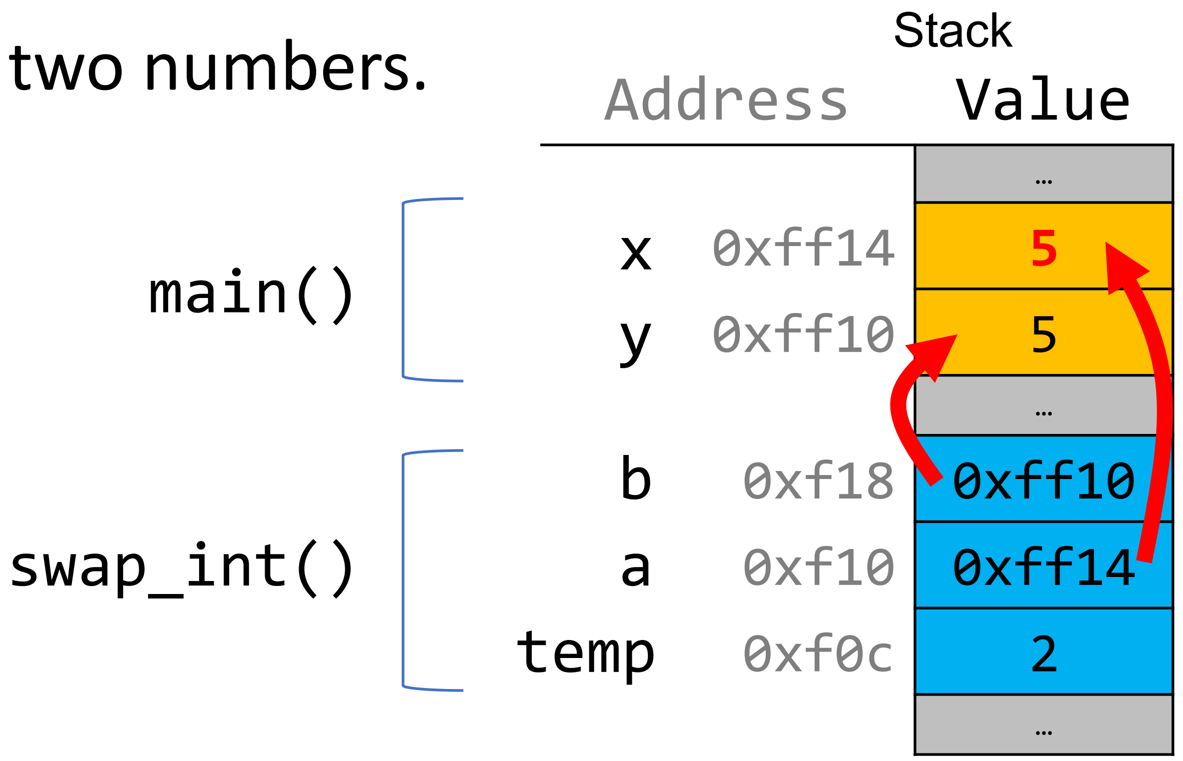
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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.

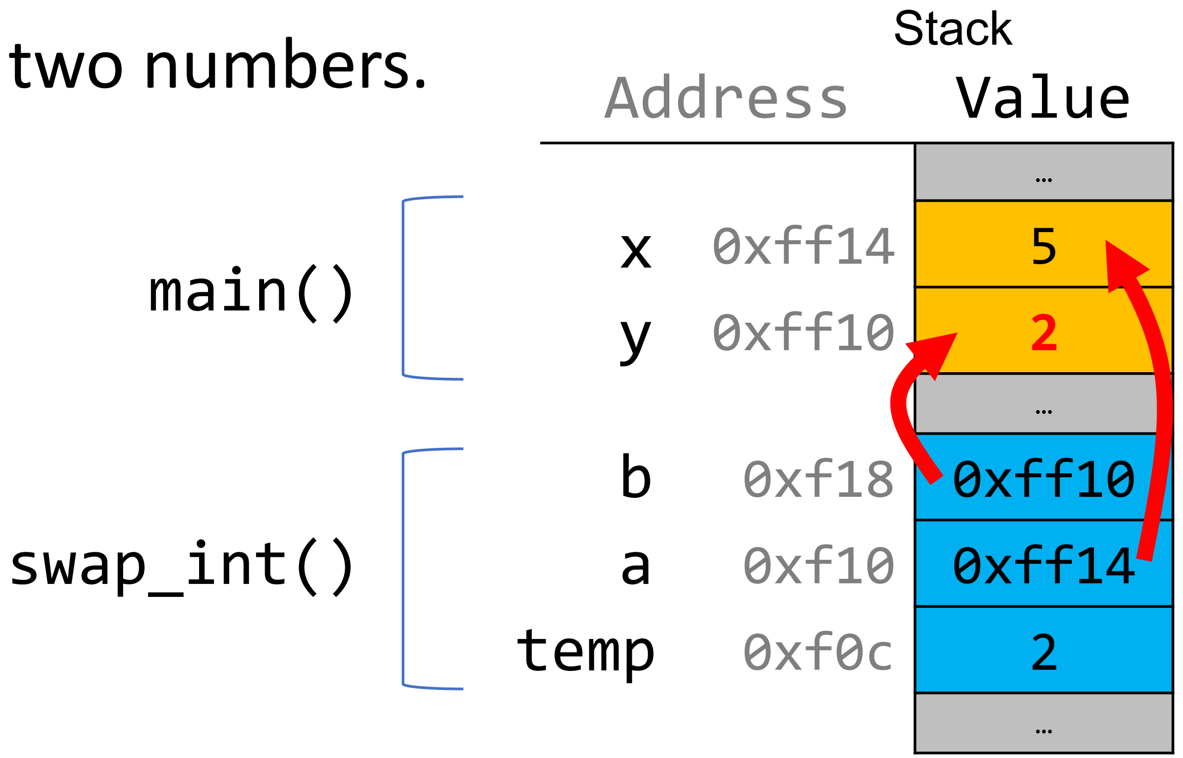
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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.

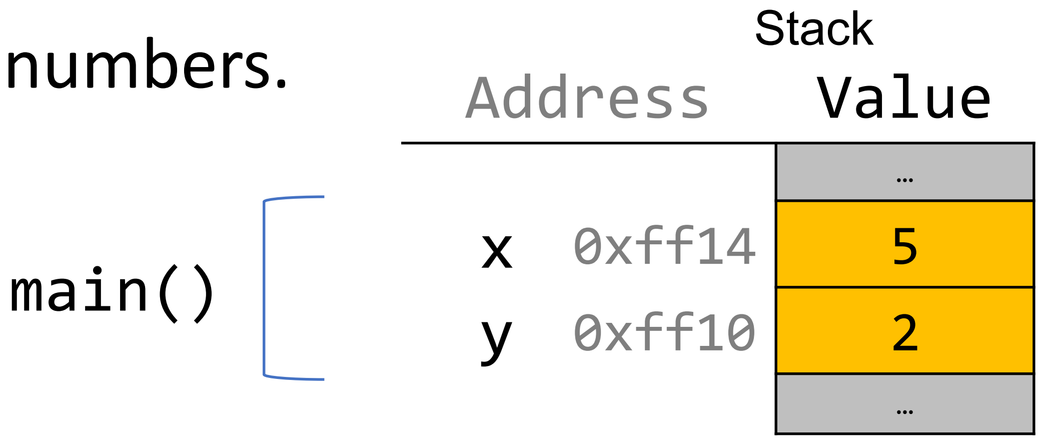
```
void swap_int(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

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;
}
```

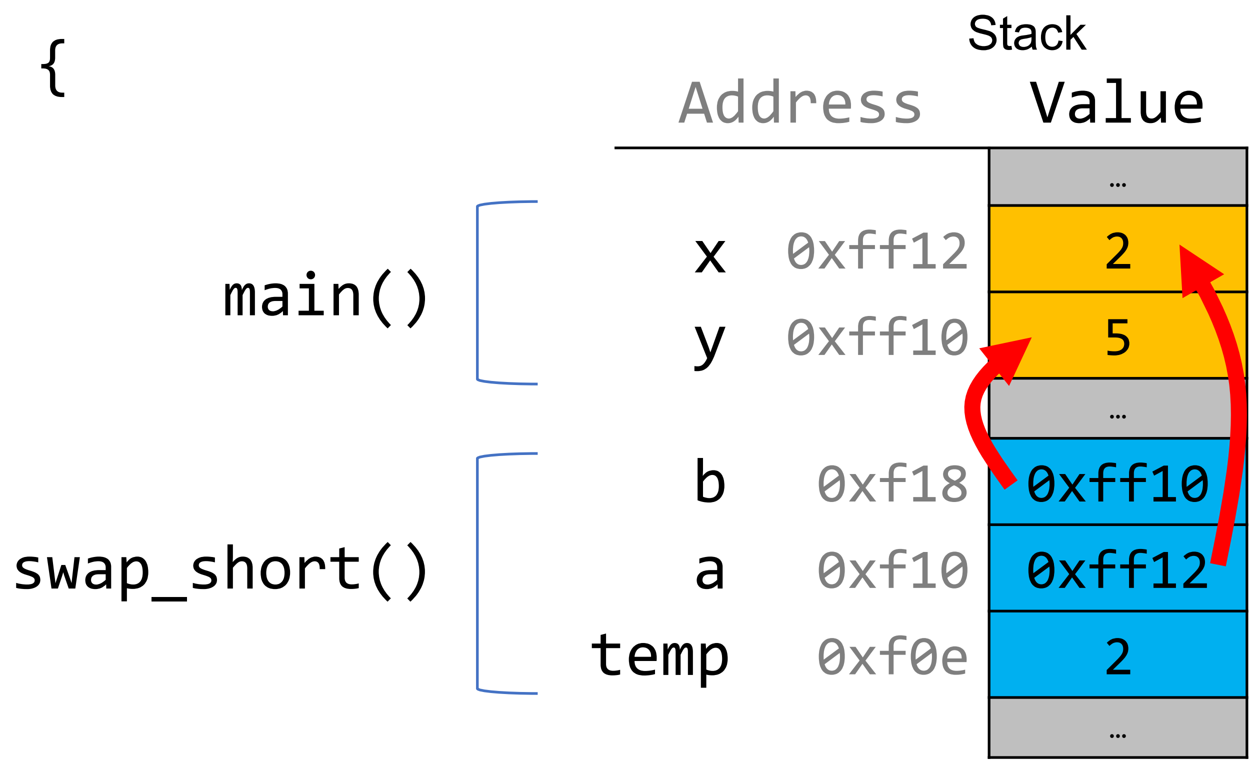


“Oh, when I said ‘numbers’ I meant shorts, not ints.”

😑

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;
}
```

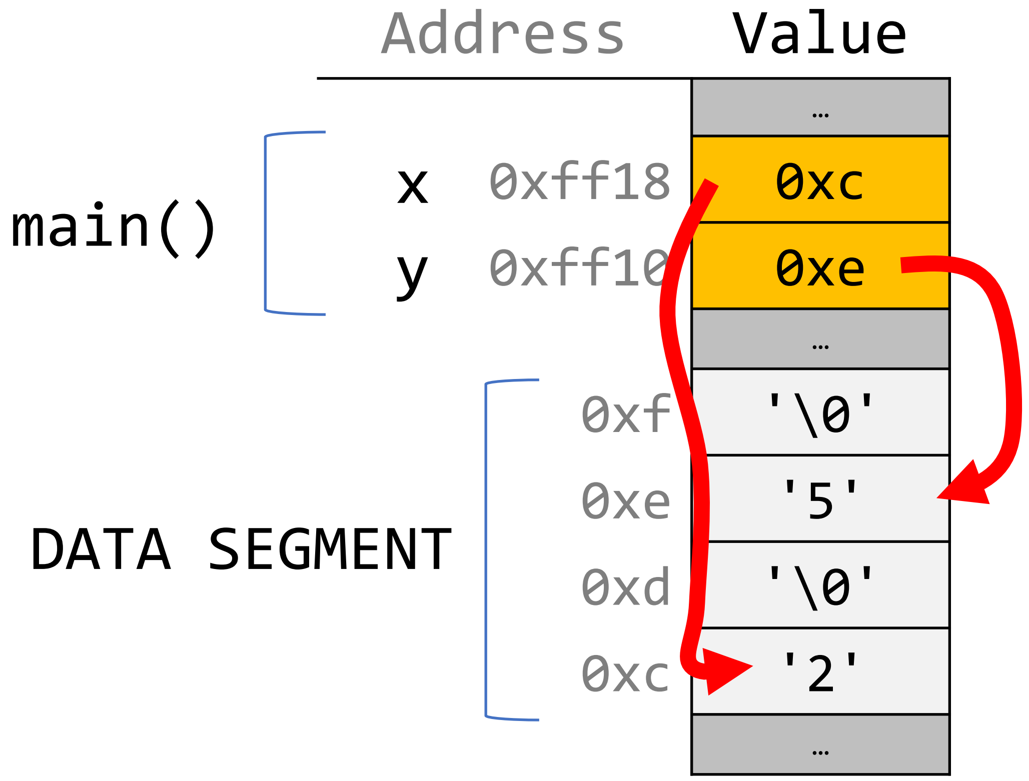


“You know what, I goofed. We’re going to use strings. Could you write something to swap those?”

😤

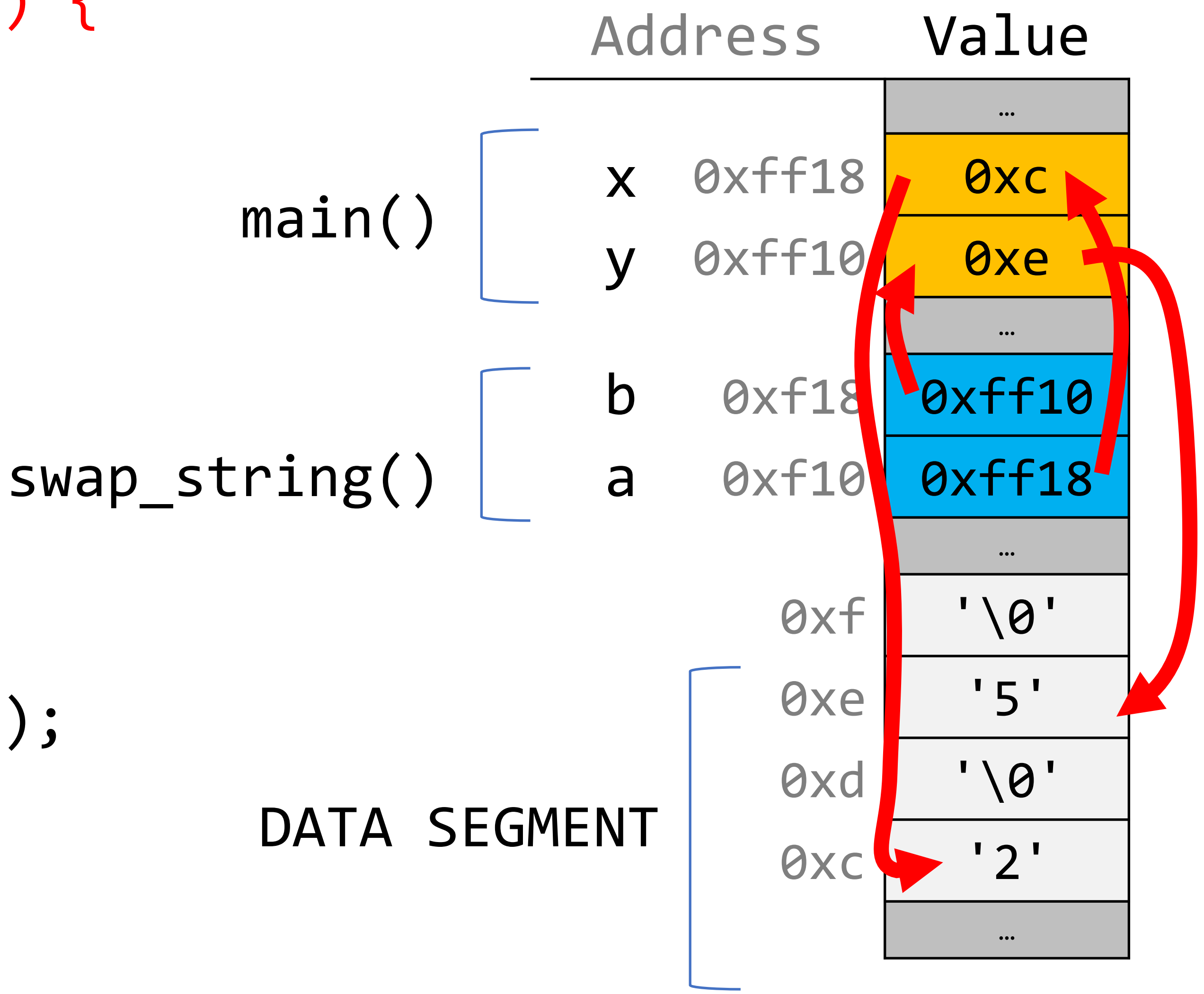
Swap

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

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



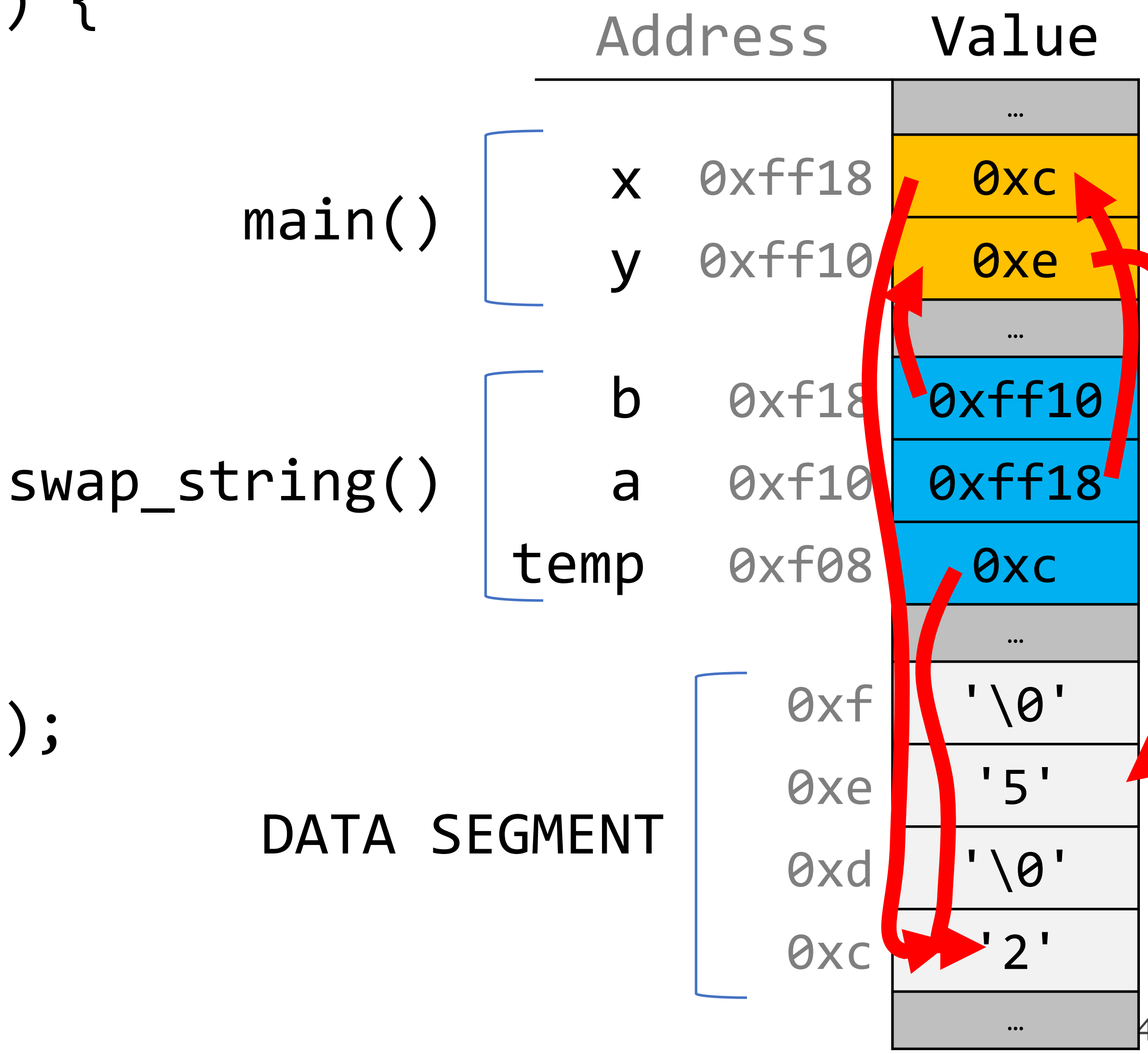
Swap

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

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



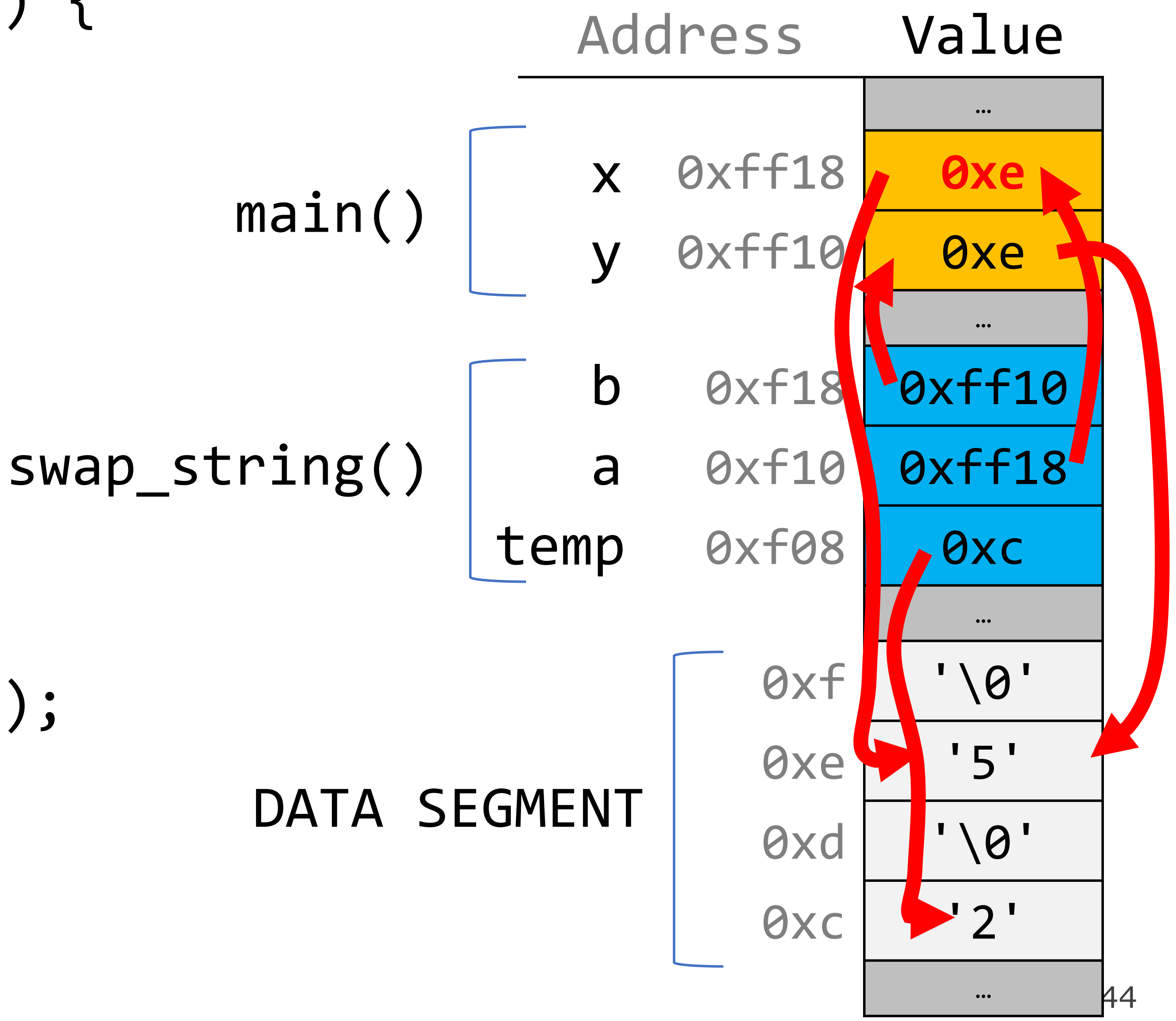
Swap

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

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



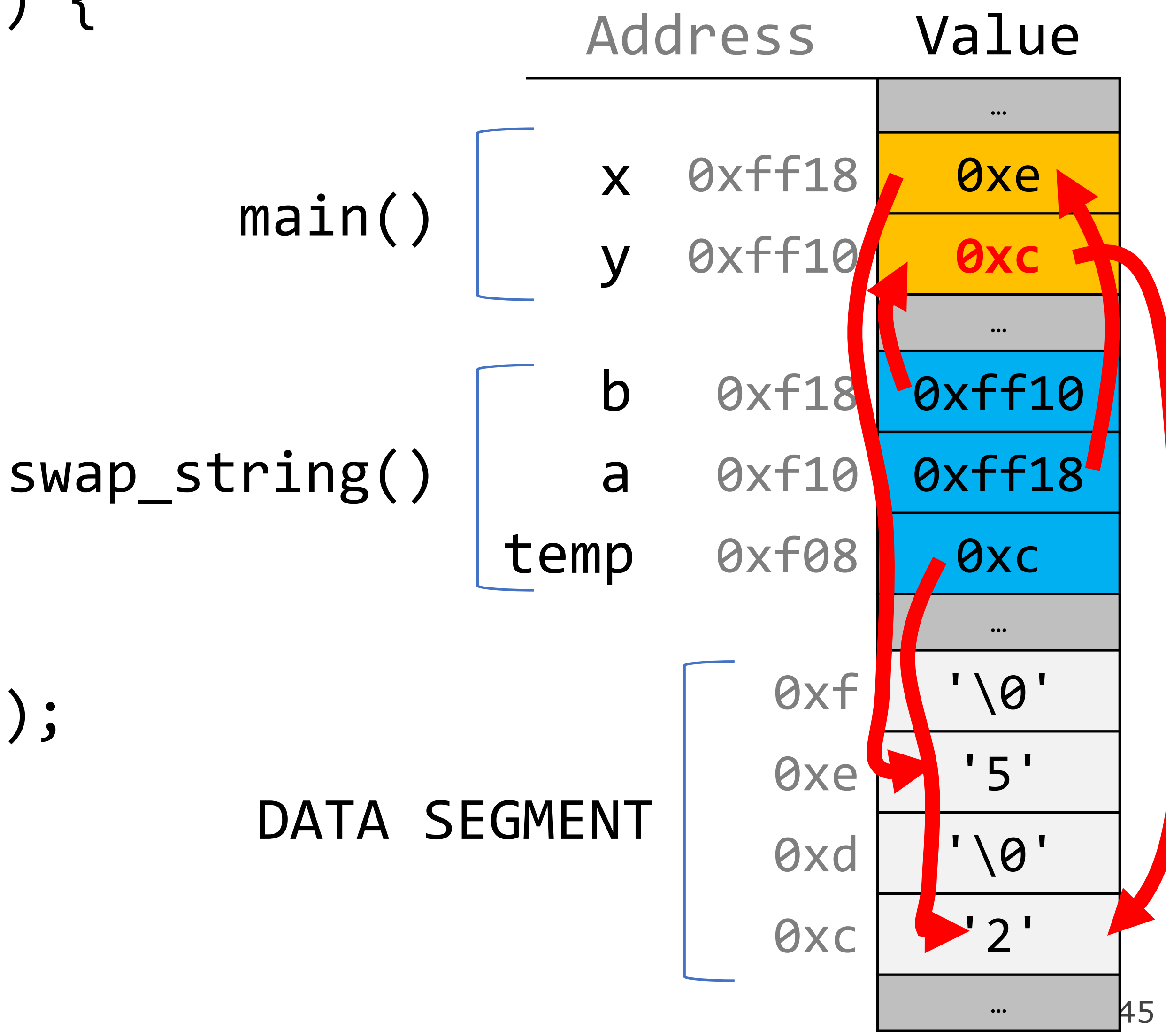
Swap

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

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



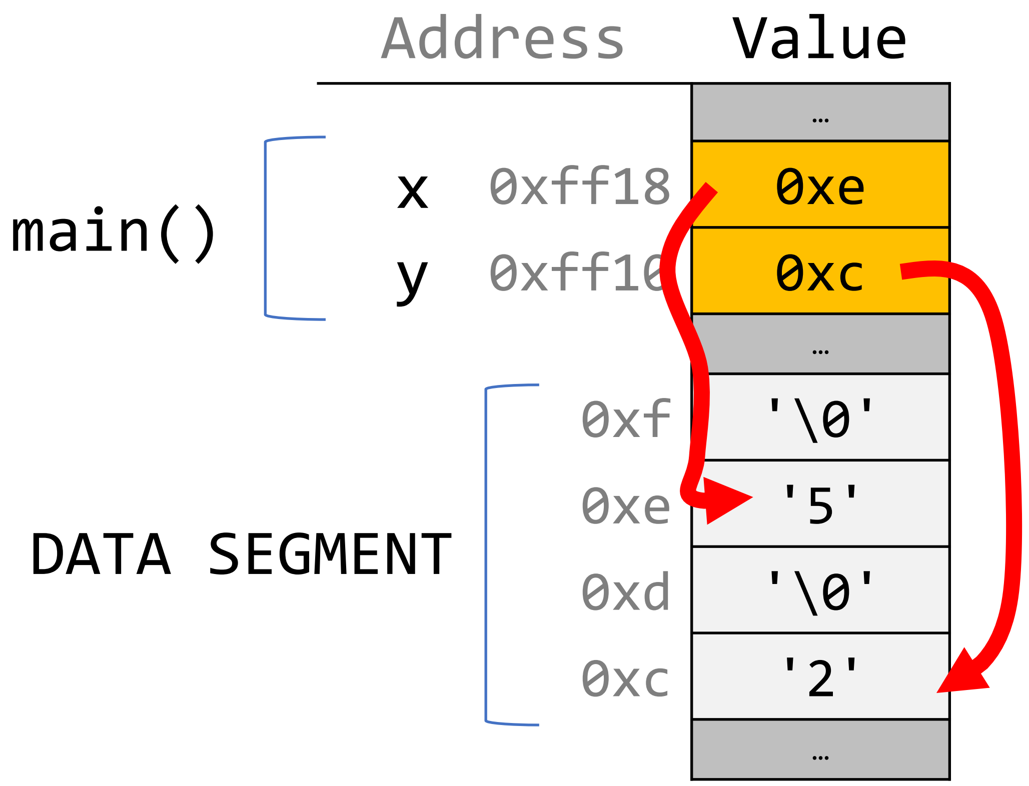
Swap

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

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



Swap

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

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



“Awesome! Thanks. 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

```c
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 number 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 pointed to by **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

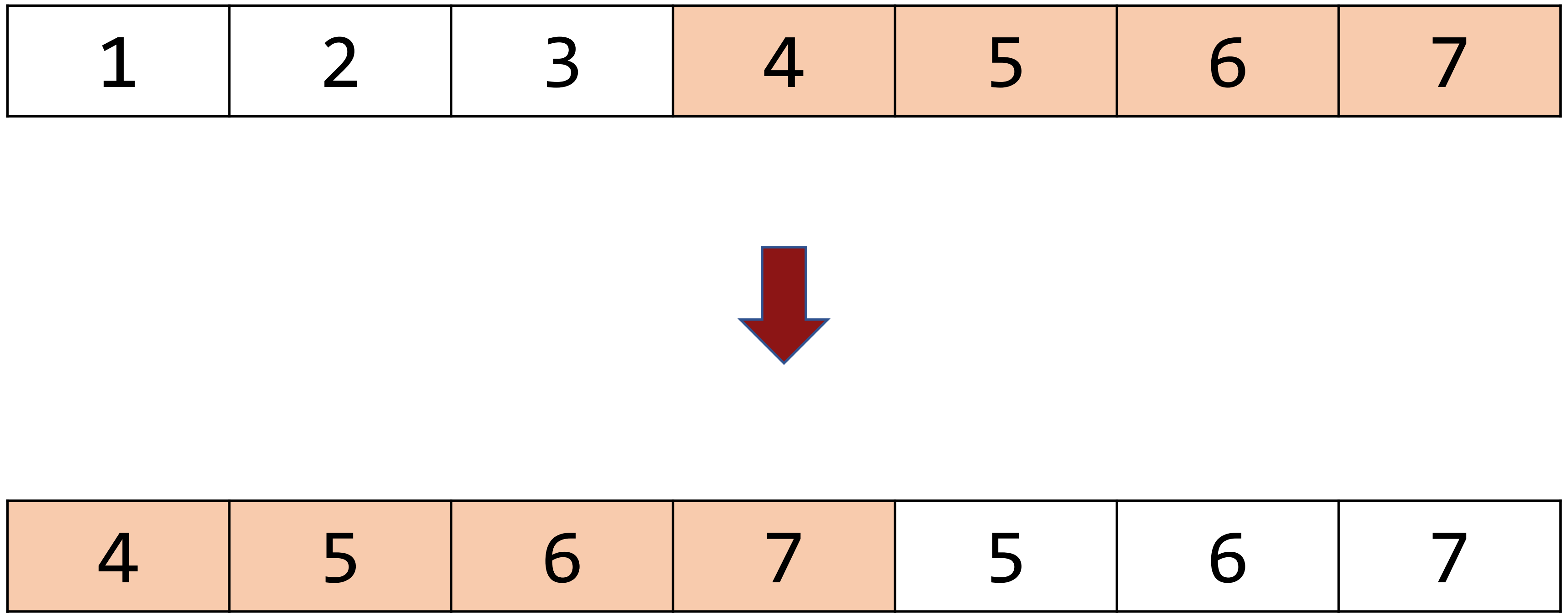
memmove is the same as memcpy but supports overlapping regions of memory. (Unlike its name implies, it still “copies”).

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

It copies the next **n** bytes that **src** points to to the location pointed to by **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
}
```

Assuming data to be swapped is not overlapping, how can **memcpy** help us here? **Respond with your thoughts on PollEv:** pollev.com/cs107 or text CS107 to 22333 once to join.

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

How can we store a copy of *data1* in temporary storage?

memcpy(data1ptr, temp, nbytes);
0%

memcpy(temp, data1ptr, nbytes);
0%

memcpy(temp, *data1ptr, nbytes);
0%

memcpy(*temp, data1ptr, nbytes);
0%

memcpy(*data1ptr, temp, nbytes);
0%

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

- Vulnerabilities, disclosure and partiality
- **Overview:** Generics
- Generic Swap
- **Generics Pitfalls**

```
cp -r /afs/ir/class/cs107/lecture-code/lect12 .
```

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

Recap

- Vulnerabilities, Disclosure and partiality
- **Overview:** Generics
- Generic Swap
- Generics Pitfalls

Next time: More Generics, and Function Pointers

Lecture 12 takeaway: Vulnerabilities should be responsibly disclosed, and partiality helps us better understand competing interests such as with vulnerability disclosure. We can use **void ***, **memcpy** and **memmove** to manipulate data even if we don't know its type. **void ***s have no type checking, so we must be vigilant!

Overflow Slides

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:

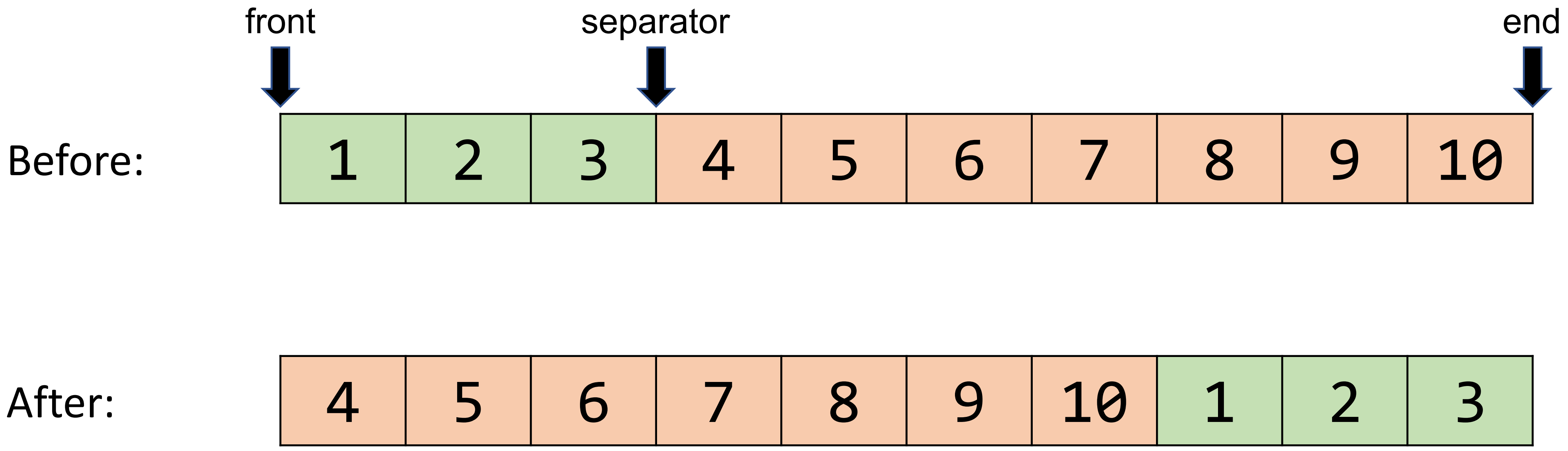
```
void rotate(void *front, void *separator, void *end);
```

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.

`rotate.c`

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

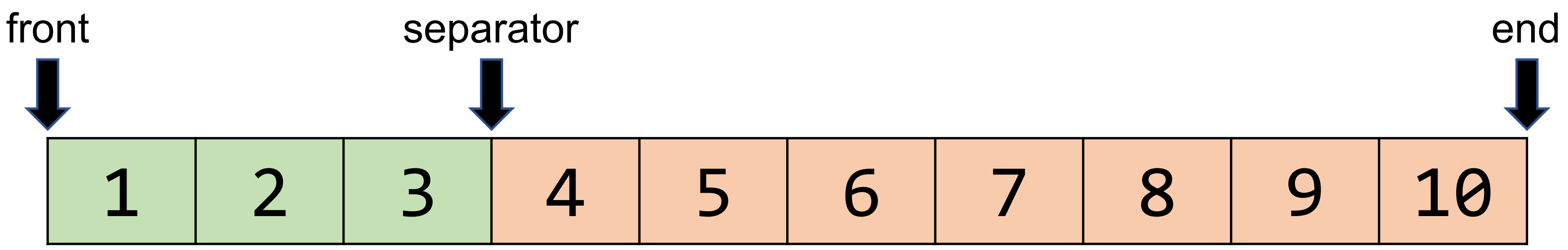
Exercise: Implement **rotate** to generate the provided output.

```
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;
}
```

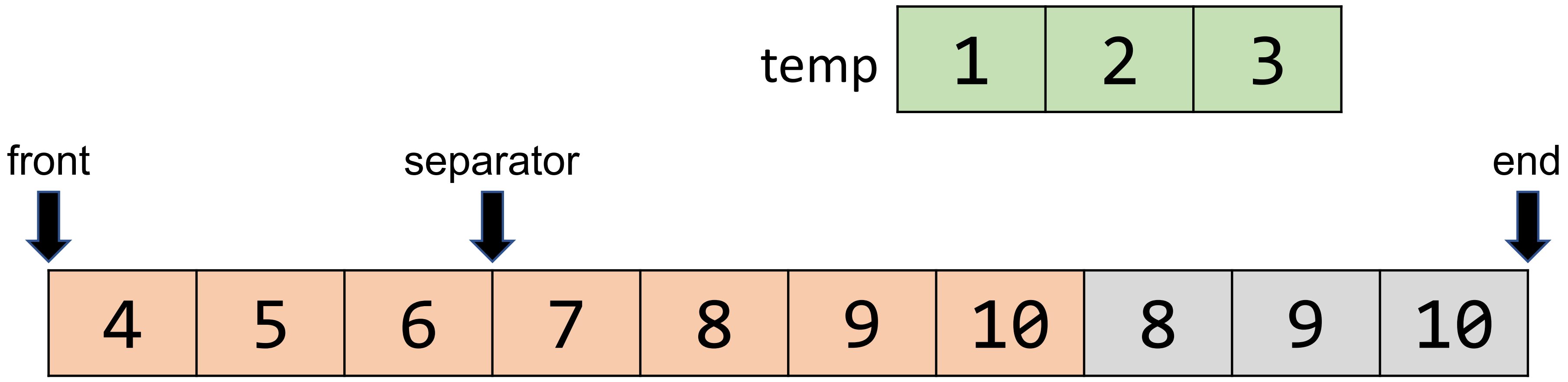
```
Output:
myth52:~/lect8$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
myth52:~/lect8$
```

The inner workings of rotate

Before rotate:



Before last step:



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);
}
```